Local Coordination for Interpersonal Communication Systems

von Dirk Kutscher

Dissertation

zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften
– Dr.-Ing. –

Vorgelegt im Fachbereich 3 (Mathematik & Informatik)
der Universität Bremen
im August 2003

Datum des Promotionskolloquiums: 22.10.2003

Gutachter: Prof. Dr-Ing. Ute Bormann (Universität Bremen)
Prof. Peter T. Kirstein (University College London)

Abstract

The decomposition of complex applications into modular units is an acknowledged design principle for creating robust systems and for enabling the flexible re-use of modules in new application contexts. Typically, *component frameworks* provide mechanisms and rules for developing software modules in the scope of a certain programming paradigm or programming language and a certain computing platform. For example, the JavaBeans framework is a component framework for the development of component-based systems — in the Java environment.

In this thesis, we present a light-weight, platform-independent approach that views a component-based application as a set of rather loosely coupled parallel processes that can be distributed on multiple hosts and are coordinated through a *protocol*. The core of our framework is the *Message Bus* (Mbus): an asynchronous, message-oriented coordination protocol that is based on Internet technologies and provides group communication between application components.

Based on this framework, we have developed a local coordination architecture for decomposed multimedia conferencing applications that is designed for endpoint and gateway applications. One element of this architecture is an Mbus-based protocol for the coordination of call control components in conferencing applications.

Acknowledgments

The research work that is described in this thesis has been performed at the *Universität Bremen* in the *Arbeitsgruppe Rechnernetze* created and headed by Prof. Dr.-Ing. Ute Bormann. Looking back over the last five years, essentially every member of that research group has been involved in some part of the work that has helped me to finally write this thesis. The central part of this thesis, the Mbus protocol, has evolved into a fundamental framework for developing component-based systems in our group, and a lot of people have contributed in different ways to make this happen: I remember numerous architecture discussions on endpoint and gateway coordination, several productive and rewarding interoperability testing events, and a lot of interesting pieces of software that have been written. This broad impact means that all the people that I mention in the following truly deserve to be acknowledged.

I sincerely appreciate the support I have received from Prof. Dr.-Ing. Ute Bormann during all phases of my research work. She has catalyzed my interest in the *concepts* of communications and especially in distributed systems when I was a student and was participating in the two-year student project *TKP* on synchronous telecooperation which Ute supervised. In these early years, I have learned to appreciate her ability to convey important concepts in a most precise, structured and understandable fashion, looking behind specific peculiarities and marketing terminology. During the last five years, I have been able to work in an extremely creative and enjoyable environment and have been able to learn a lot from her expertise, not only in the computer networks domain. After I have joined her research group in 1998, she has not only given me the opportunity to work in the telecooperation area by participating in a number of unique research projects but has also introduced me into international standardization work, namely in the IETF, which has significantly promoted my further research activities. In creating this thesis, she has supported me in multiple ways and has been of significant help for improving the quality of the final document. Not only for writing this thesis, I have enjoyed her strong sense for quality and have benefited from her advice.

My participation in different regional, national and international research projects and the collaboration with many different international partners has been an extremely rewarding experience and has given me the opportunity to validate my ideas in different application contexts and to take part in many interesting conceptual discussions and practical undertakings. In particular, I would like to mention the European projects MECCANO and 6WINIT that I had the pleasure to participate in. Both projects have played a substantial role in the development of the Mbus and our local coordination architecture: in MECCANO, we initially designed the Mbus protocol and applied it to the development of decomposed multimedia conferencing gateways. In 6WINIT, we advanced the Mbus framework and designed our gateway control architecture. Both of these projects have been directed by Professor Peter T. Kirstein, University College London (UCL). Professor Kirstein, recently awarded the Jonathan B. Postel Service Award, is

one of the pioneers of Internet and multimedia conferencing technologies related research and is the nestor of the Internet in the UK. In MECCANO and 6WINIT, I have enjoyed an extremely fruitful collaboration and have been able to benefit a lot from his tremendous knowledge and experience. I am especially grateful that he has agreed to be my advisor for this thesis, and his comments have indeed been very helpful. In fact, my research owes a lot the excellent collaboration between our research groups: The Mbus idea has initially been sketched by Dr.-Ing. Jörg Ott and Colin Perkins who has participated in the MECCANO project as a member of Professor Kirstein's group. The development and standardization of the Mbus protocol together with Jörg and Colin has been a great experience and I particularly acknowledge the work of Colin and other people at UCL for developing the Mbus-enabled audio tool RAT that has been an important element in many of our applications.

In addition, my thanks go to Prof. Dr. Ulrike Lechner and Prof. Dr. Jan Peleska for joining the examination committee and for proving their patience during the scheduling difficulties.

The foundation of my research work has been laid by Dr.-Ing. Jörg Ott, who originally conceived the *internal management service* for conferencing systems and the Mbus idea. Our common involvement in many different projects and the numerous conceptual discussions have been a major thrust for the development of our local coordination architecture. In particular, our participation in the DTI and FETA projects has been an extremely pleasant and rewarding experience: for the first time, we have had the opportunity to transfer the local coordination concept to existing IP telephony platforms and to advance the ad-hoc communication idea by developing the Dynamic Device Association mechanisms for Mbus devices. Together with Ute, we have supervised the Hausgeist project that has investigated employing the Mbus technology for large-scale ad-hoc communication in home networks, which has led to many interesting discussions that have also contributed to my research work. During all of our joint activities, I have benefited a lot from Jörg's deep knowledge of multimedia conferencing technologies, from his striving for quality, and from his remarkable ambition and reliability, e.g., for finishing papers at absurd times, despite an enormous workload. I would not want to miss the many conceptual comments he has provided for the Mbus work and related activities.

I would also like to thank Dr.-Ing. Carsten Bormann who has been the managing director of our group for the last five years and who has supported me in multiple ways. He has been an authoritative information source for many questions in several discussions and projects, and I have benefited a lot from his experience. I have especially appreciated Carsten's guidance for the preparation of this thesis and for other documents that we have authored.

This thesis and the research results that I describe therein owe a lot to the extraordinary productive and enjoyable environment I have worked in during the last five years. Professor Bormann's research group at the University of Bremen is one of the few associations of lecturers, researchers and students that combines interesting conceptual leading-edge research with its practical implementation and validation. This focus on application-oriented research is maintained by a group of researchers that I have got to know as challenging partners in discussions on concepts and architectures but also as creative and reliable partners in project work, in software development, and in technology transfer activities. Due to the infrastructure character of the Mbus framework, literally all of them have been exposed to the Mbus and have thus contributed to my work by participating in discussions and interoperability events, by contributing Mbus implementations, by developing Mbus applications and by providing me with feedback.

Stefan Prelle has developed the Java Mbus implementation and an Mbus-based H.323 call

control engine that we have used in our gateway systems. In particular, he contributed to the advancement of the Mbus Call Control Profile by pointing to H.323 requirements, thus helping to keep the profile generic. Olaf Bergmann developed the first C++ implementation of the Mbus Guidelines interaction mechanisms and also provided valuable feedback for advancing the C++ implementation and the Call Control profile, which he has used for developing an Mbus-based SIP server system.

Markus Germeier has developed AudioGate, our first Mbus-based gateway system, and he has also written the Perl Mbus implementation. Niels Pollem has developed the StarGate gateway and implemented a first version of an Mbus-based controller for gateway systems. The StarGate development has initiated the definition of the Mbus Call Control Profile.

Andreas Büsching has developed the media processor component that we have used in our Mbus-based media gateway. He has also done a lot of Mbus hacking and porting for the DTI and FETA projects. In particular, he has ported the C++ implementation to MS VC++ and has developed our first DDA server implementation. Dirk Meyer has implemented another C++ Mbus RPC implementation, has contributed to the Mbus Call Control Profile and has co-developed the Mbus-based *Wipone* endpoint in the Wiptel project. Eilert Brinkmann has advanced our gateway controller for the 6WINIT project and has provided valuable feedback for improving the C++ implementation.

Anne-Kathrin Stoll has won my personal “Bjarne Stroustrup award” for implementing the design of a C++ template based socket abstraction for IPv4 and IPv6 providing unified interfaces to socket operations without compromising run-time efficiency, which we have used for our gateway systems. Anja Prelle has developed an Mbus-based CPL module that can be used as a component in SIP proxies and H.323 gatekeepers. Kevin Loos has developed the Python Mbus implementation and a corresponding DDA client implementation that we have used on PDA systems for the DTI and FETA projects.

Quite a lot of people have developed Mbus implementations on their own and have thus helped by providing feedback with respect to the Mbus protocol specification, by contributing their developments and by participating in interoperability testing events. In addition to my own Mbus implementations, I am aware of the following implementations:

- UCL C implementation, developed by Colin Perkins as part of UCL’s commonlib that is used for RAT and other Mbone tools;
- University of Bremen Java implementation, developed by Stefan Prelle;
- University of Bremen Perl implementation, developed by Markus Germeier and later continued by members of the Hausgeist project;
- Hausgeist Python implementation, developed by Kevin Loos during the Hausgeist project, later used for the DTI and FETA projects; and the
- Hausgeist C# implementation, developed by Daniel Losch for MS .NET platforms.

I would not have been able to pursue this work without the tremendous support from my parents, my family, and my friends who have been an invaluable source of encouragement, especially during the last months. In particular, I am sincerely grateful to Christiane who has done an excellent job in managing our family during a challenging period and for being a wonderful mother to our sons Ben Fuman and Nick Otis — they are great kids!

Contents

1 Introduction **1**
1.1 Networks of Personal Devices . . . 3
1.2 Trends for Conferencing Systems . . . 5
1.3 A Message Bus for Local Coordination . . . 9
1.3.1 A Decomposed Multimedia Conferencing System . . . 9
1.3.2 Integration into Computing Environments . . . 10
1.4 Research Areas . . . 12
1.5 Structure of this Thesis . . . 14

2 Conferencing Architectures **15**
2.1 The H.323 Recommendations . . . 17
2.1.1 Conference Initiation . . . 18
2.1.2 Data Protocols for Multimedia Conferencing . . . 20
2.1.3 Conference Course Control . . . 22
2.2 Internet Multimedia Conferencing Architecture . . . 24
2.2.1 Conference Initiation . . . 25
2.2.1.1 Session Announcement . . . 26
2.2.1.2 Session Initiation . . . 27
2.2.2 Conference Course Control . . . 29
2.2.2.1 The Agreement Protocol . . . 29
2.2.2.2 CCCP . . . 31
2.2.2.3 SCCP . . . 33
2.2.3 Summary . . . 34
2.3 Lessons Learned . . . 35

3 Use Cases and Requirements **39**
3.1 Use Cases . . . 39
3.1.1 A Decomposed Multimedia Conferencing System . . . 39
3.1.2 In-Vehicle Networks . . . 45
3.1.2.1 From Cable Replacement to New Applications . . . 45
3.1.2.2 Real-time In-Vehicle Communication . . . 46
3.1.2.3 Multimedia In-Vehicle Communication . . . 48
3.1.2.4 In-Vehicle Communication in the Future . . . 49
3.1.2.5 Observations . . . 51
3.1.2.6 Redesign . . . 53
3.2 Requirements . . . 55

3.2.1 Requirement *Ad-hoc communication* . . . 55
3.2.2 Requirement *Scalability* . . . 56
3.2.3 Requirement *Group communication* . . . 56
3.2.4 Requirement *Intra- and Inter-host communication* . . . 57
3.2.5 Requirement *Efficiency* . . . 57
3.2.6 Requirement *Small Footprint* . . . 58
3.2.7 Requirement *Reliable Communication* . . . 58
3.2.8 Requirement *Message Ordering* . . . 59
3.2.9 Requirement *Standardized Interaction Schemes* . . . 59
3.2.10 Requirement *Security* . . . 59
3.3 Summary . . . 60

4 Foundations and Related Work **61**
4.1 Distributed Systems . . . 62
4.1.1 Remote Procedure Call Paradigm . . . 63
4.1.2 Group Communication and the ISIS Toolkit . . . 67
4.1.3 Lessons Learned . . . 71
4.2 Local Coordination (Vertical Control) . . . 72
4.2.1 LBL Conference Bus . . . 73
4.2.2 Pattern Matching Multicast . . . 74
4.2.3 Summary . . . 75
4.3 Coordination and Ad-Hoc Communication . . . 76
4.3.1 TIBCO Rendezvous . . . 77
4.3.1.1 Ordering and Synchronization . . . 77
4.3.1.2 Architecture and Communication . . . 78
4.3.1.3 Reliability . . . 79
4.3.1.4 Programming Model . . . 80
4.3.1.5 Summary . . . 80
4.3.2 Service Discovery and “Plug and Play” Solutions . . . 81
4.3.2.1 Service Location Protocol . . . 81
4.3.2.2 Jini . . . 83
4.3.2.3 Universal Plug and Play . . . 85
4.3.3 Lessons Learned . . . 88
4.4 Component Technologies . . . 90
4.4.1 Component Object Model (COM) . . . 91
4.4.2 JavaBeans . . . 91
4.4.3 Scripting Languages (Tcl) . . . 92
4.4.4 CORBA . . . 93
4.4.4.1 Remote Method Invocation Service . . . 94
4.4.4.2 Object Management Architecture . . . 94
4.4.4.3 Summary . . . 95
4.4.5 Lessons Learned . . . 95
4.5 Summary . . . 97

5 Architecture **100**
5.1 Basic Message Transport 101
5.2 Higher Layer Interactions 101
5.3 Device Association 102
5.4 Local Conference Control Architecture 102

6 The Mbus Framework **103**
6.1 Design Overview 103
6.2 Mbus Transport 105
6.2.1 Basic Transport Mechanisms 106
6.2.1.1 Addressing 106
6.2.1.2 Group Communication 107
6.2.1.3 Mapping to IP Unicast and IP Multicast 109
6.2.1.4 Unicast Communication 111
6.2.1.5 Reliable Communication 113
6.2.2 Membership Information Service 121
6.2.2.1 Hello Message Transmission Interval 122
6.2.3 Message Syntax 123
6.2.4 Security 125
6.2.4.1 Message Authentication 126
6.2.4.2 Message Encryption 127
6.2.4.3 Applying Authentication and Encryption to Messages 127
6.2.5 Configuration of Mbus Transport Parameters 129
6.2.6 Mandatory Mbus Commands 130
6.2.6.1 Commands for Membership Management 130
6.2.6.2 Commands for Synchronization 131
6.2.7 Sample Mbus Session 132
6.3 Higher Layer Interactions 132
6.3.1 Usage of Mbus Addresses 134
6.3.2 Interaction Models 138
6.3.2.1 Remote Commands 139
6.3.2.2 Mbus Remote Procedure Calls 140
6.3.2.3 Mbus Transactions 149
6.3.3 Mbus Control Models 151
6.3.3.1 Managing Control Relationships 153
6.3.3.2 No Control 154
6.3.3.3 Tight Control 155
6.3.3.4 Exclusive Tight Control 156
6.4 Mbus Bootstrapping 157
6.4.1 Sample Scenario 158
6.4.2 Service Location and Device Association 159
6.4.3 DDA System Design 160
6.4.3.1 Service and Device Discovery 161
6.4.3.2 Device Selection 163
6.4.3.3 Service and Device Association 163
6.4.3.4 Application Protocol Operation 164

6.4.3.5 Service and Device Dissociation . . . 165
6.4.4 DDA Implementation . . . 165
6.4.4.1 Service and Device Discovery . . . 166
6.4.4.2 Service and Device Association . . . 169
6.4.5 Summary . . . 170
6.5 Mbus and Ad-hoc Communication . . . 171
6.5.1 Requirements for Group Communication in Ad-hoc Environments . . . 172
6.5.2 DDA Extensions for Multiparty Peering . . . 173
6.5.2.1 Requirements for Multiparty Peering . . . 174
6.5.2.2 Modified DDA Association . . . 175
6.5.3 Mbus Extensions for Multiparty Peering . . . 177
6.5.3.1 Coordinator Concept . . . 177
6.5.3.2 Connectivity Discovery . . . 179
6.5.3.3 Visibility Reporting . . . 180
6.5.3.4 Message Transmission and Forwarding . . . 181
6.5.3.5 Change and Failure Handling . . . 182
6.5.4 Summary . . . 182
6.6 Summary . . . 184

7 Mbus Implementations **187**
7.1 Implementation Considerations . . . 187
7.2 Basic Implementation Requirements . . . 189
7.3 C++ Implementation . . . 191
7.3.1 Design . . . 191
7.3.2 Integration into Applications . . . 195
7.3.3 Mbus Guidelines Implementation . . . 197
7.3.4 Summary . . . 198
7.4 Java Implementation . . . 200
7.5 Mbus Implementations for Scripting Environments . . . 201
7.6 Summary . . . 203

8 Evaluation **205**
8.1 Basic Message Transport and Security Mechanisms . . . 206
8.2 RPC Communication . . . 208
8.3 Fault Tolerance . . . 212
8.4 Summary . . . 215

9 Mbus in Conferencing Systems **216**
9.1 System Model . . . 216
9.1.1 The Internal Management Concept . . . 217
9.1.2 Mbus Coordination Model . . . 218
9.1.3 Local Coordination in End Systems . . . 220
9.1.4 Local Coordination in Call-Signaling Gateways . . . 222
9.1.5 Summary . . . 223
9.2 Session Description . . . 225
9.2.1 SDPng System Model . . . 226

9.2.2 SDPng Design 229
9.2.2.1 The SDPng Negotiation Process 229
9.2.3 SDPng in the Local Coordination Architecture 232
9.2.4 Summary 233
9.3 Mbus Call Control 234
9.3.1 Concepts 236
9.3.2 The Mbus Call Control Profile 237
9.3.2.1 Mbus Parameter Type Definitions 237
9.3.2.2 Mbus Addressing Scheme 239
9.3.2.3 Mbus Commands 240
9.3.3 Basic Services 241
9.3.3.1 Initialization 241
9.3.3.2 Call Setup 242
9.3.4 Lessons Learned 245
9.4 Summary 248

10 Mbus in Projects **251**
10.1 Endpoint Decomposition 252
10.1.1 CONTRABAND 252
10.1.2 Mbus-based Application Entities 255
10.1.3 Bonephone 258
10.2 Mbus in Desk Area Environments 259
10.2.1 Desktop Telephony Integration 259
10.2.1.1 Concepts 260
10.2.1.2 Automated Conference Creation 261
10.2.1.3 Implementation 262
10.2.2 Functional Enhancements using external Telephone Applications 263
10.2.2.1 Integration of Multimedia Clients on a Workstation 264
10.2.2.2 Endpoint-based Conferencing 265
10.2.2.3 Implementation 267
10.2.3 Lessons Learned 267
10.3 Gateways 269
10.3.1 IPv6 Wireless Internet Initiative 270
10.3.2 Gateway Architecture 270
10.3.3 Implementation 272
10.3.4 Lessons Learned 275
10.4 Mbus for Device Coordination 276
10.5 Summary 277

11 Conclusions **279**
11.1 Conceptual Achievements 280
11.2 Engineering Results 281
11.3 Mbus Compared to Other Approaches 283
11.4 Open Issues and Next Steps 285
11.4.1 Mbus Protocol 285
11.4.2 Conferencing Architecture 289

11.4.3 New Application Areas 290
11.4.4 Lessons Learned 291

Bibliography **293**

A Colophon **308**

List of Figures

1.1 Decomposed IP telephone 9
1.2 Integration of external functional components 11

2.1 Basic H225.0-Q.931 call setup process 19
2.2 The T.120 framework 21
2.3 An MCS domain 22
2.4 GCC components in an MCS domain 23
2.5 The Internet Real-time Multimedia Conferencing Architecture 25
2.6 Basic SIP call setup process 28
2.7 Conceptual overview of CCCP 32
2.8 Generalized local endpoint architecture 37

3.1 Modular conferencing endpoint 40
3.2 Inter-system communication 44
3.3 A sample in-vehicle network 47
3.4 AMI-C network transport layer 51
3.5 Addressing components in an in-vehicle network 54

4.1 Interaction for a Remote Procedure Call 64
4.2 Sample PMM message 74

5.1 Mbus Protocol Layers 100

6.1 Sample Mbus address 107
6.2 Algorithm for an Mbus address-matching predicate 108
6.3 Receiver-based filtering by an Mbus implementation (simplified) 108
6.4 Algorithm for determining the uniqueness of an Mbus address 112
6.5 Implementation of an algorithm for determining the uniqueness of an Mbus address 112
6.6 A bridged local network (WLAN and Ethernet) 114
6.7 The hidden-terminal problem 116
6.8 Acknowledging a reliable message 118
6.9 Acknowledging multiple reliable messages 118
6.10 Retransmitting a complete list of acknowledgments 119
6.11 Acknowledgment piggybacking and retention in request/response scenarios 121
6.12 Mbus message syntax 123
6.13 Sample Mbus header 124

6.14 Sample Mbus commands . . . 125
6.15 Applying the Mbus security functions . . . 129
6.16 Sample Mbus configuration file . . . 130
6.17 Sample Mbus message exchange with synchronization . . . 133
6.18 Sample fully qualified Mbus address and three partly qualified Mbus addresses 135
6.19 C++ function to aggregate a set of Mbus addresses into a single Mbus address 137
6.20 C++ function to expand an Mbus group address into a set of corresponding fully qualified addresses . . . 138
6.21 Sample Mbus remote command interaction . . . 139
6.22 Sample Mbus RPC interaction . . . 141
6.23 Sample Anycast interaction . . . 145
6.24 Sample coordinated RPC interaction . . . 148
6.25 Sample Mbus transaction . . . 150
6.26 Canceled Mbus transaction . . . 152
6.27 Usage of default destination addresses . . . 155
6.28 Registration of a controller with an entity implementing tight control . . . 156
6.29 Registration of a controller with an entity implementing exclusive tight control 157
6.30 Overview of the DDA process . . . 160
6.31 Example for a DDA service announcement . . . 167
6.32 Example of a DDA session description . . . 169
6.33 The complete DDA process . . . 170
6.34 Sample multiparty peering process . . . 173
6.35 DDA in invitation mode . . . 176
6.36 Three multiparty peering scenarios . . . 178
6.37 Mbus-based connectivity discovery . . . 180
6.38 Visibility reports and relaying of mbus.hello messages . . . 181

7.1 Layers in the C++ implementation . . . 191
7.2 Layer interaction in the C++ implementation . . . 193
7.3 An Mbus C++ application . . . 195
7.4 Mbus RPC server application class . . . 198
7.5 Mbus RPC client application class . . . 199
7.6 Sample Java Mbus application . . . 201

8.1 RPC interaction without acknowledgment piggybacking . . . 209
8.2 RPC Interaction with acknowledgment piggybacking . . . 209
8.3 Effect of packet loss on RPC delivery performance . . . 213

9.1 Internal management and wide-area communication . . . 217
9.2 Mbus coordination model . . . 218
9.3 Signaling gateway . . . 222
9.4 The SDP offer/answer model . . . 226
9.5 Conferencing system model . . . 228
9.6 Mbus address for a controller . . . 239
9.7 Mbus address for a call signaling engine . . . 240
9.8 Call setup model . . . 243

10.1 The CONTRABAND system architecture 253
10.2 The wearable CONTRABAND conferencing endpoint 254
10.3 RAT initialization communication 256
10.4 RAT runtime communication 257
10.5 PDA GUI for dynamic device association 260
10.6 Automated conference setup 262
10.7 Mbus Call Control for conference creation 263
10.8 FETA application integration scenario 264
10.9 FETA endpoint-based conferencing scenario 266
10.10 System architecture of the Mbus-based TZI-Gateway 271
10.11 SIP-based signaling component 272
10.12 Incoming call state diagram 274

List of Tables

2.1 Services and protocol mechanisms for H.323 and IMCA 35

4.1 Comparison of the LBL Conference Bus and PMM 75

6.1 Multicast scopes for IPv6 and IPv4 110
6.2 Mandatory Mbus commands 131

8.1 Roundtrip-times (1) 206
8.2 Roundtrip-times (2) 206
8.3 RPC interaction times 210

9.1 Overview of the call control parameter types 239

Chapter 1

Introduction

The term *pervasive computing* has been a popular metaphor in the computer science research community for the recent years. The term refers to the trend of increasingly *ubiquitous*, networked computer systems that perform specific functions and become a natural part of users' daily-life activities. One important aspect of pervasive computing devices is their ability to communicate with each other autonomously, e.g., in order to perform coordination and to exchange data. Early research on pervasive computing (the term *ubiquitous computing* is a synonym) has been performed in 1987 by researchers of the Palo Alto Research Center (PARC) [Weiser99].

There are many different usage scenarios for ubiquitous computing, ranging from devices in a personal (body) area network to pervasive computing in home and office environments. The ubiquitous computing community is studying different aspects of this trend, including philosophical questions [Gold95], user interface issues [Ark99] and device communication [Zimmermann99]. In this thesis, we focus on the *communication aspects* for such environments, in particular on the *coordination* aspects.

Coordination-based protocols and frameworks have evolved from traditional distributed computing technologies, such as RPC communication [Birell84] [RFC1831] and group communication [Birmann87b], and focus on a rather loose coupling of communication peers, considering dynamic association of peers and communication between referentially and temporally uncoupled entities. Examples for corresponding technologies are *Universal Plug and Play* (UPnP [UPnP03]), *Jini* [Sun01] and *TIBCO Rendezvous* [TIBCO02]. In general, these protocols forego the static binding of peers and elaborate mechanisms for guaranteeing consistency and fault tolerance in favor of a high degree of flexibility and the possibility to establish communication sessions dynamically.

An application and research domain that has a longer history of coordination-based architectures and protocols, is the field of *multimedia conferencing*, i.e., synchronous interpersonal communication through networked computer systems. The initiation of multimedia conferences and the control of their operation requires a coordination between endpoints that is known as *call control* and *conference course control* [Handley00] [Ott97]. Moreover, conferencing architectures promote a local endpoint architecture that is highly modular and has a strong demand for the coordination of the different elements in an endpoint system: there are multiple application entities that implement different applications for the multimedia conferencing service, e.g., an interactive audio application entity, a video entity and one or more shared application entities.

In fact, due to the standardized protocols that are employed for the communication *be-*

tween endpoints in a multimedia conference, the functionality of application entities and other functional entities in different conferencing endpoints can be generalized, and often different systems provide a similar logical structure, even though they are heterogeneous with respect to their individual capabilities and implementation details. Later, a detailed analysis will reveal functional components such as call control entities that provide the inter-endpoint session establishment, the mentioned application entities, and dedicated coordinating entities that orchestrate these different modules. Due to the application independence of call control protocols such as SIP [RFC3261] and H.245 [ITU95c],[1] some of these components can actually be generic and be used for multimedia configurations as well as for more limited endpoints such as audio-only systems.

The overall similarity of functionality, the generality of endpoint architectures, and especially the predominant modularity, suggest a *component-based approach* for the development of conferencing endpoint systems. As we will discuss later in detail, component-based approaches are typically motivated by the desire to *re-use* existing components in new contexts in order to construct new applications by assembling existing building blocks. Requirements for the deployment of component technologies are standardized interfaces, the establishment of recognized architectures and emerged *best current practices* to solve particular problems in a certain domain. However, while appropriate architectures and best current practices for the development of conferencing endpoints have generally won recognition during the recent years, an acknowledged component-based approach for this application is still missing. This can be ascribed to several factors:

- Platforms for conferencing endpoints such as IP telephones are often quite light-weight devices with limited computational power and storage capabilities. Such devices have to meet certain power consumption limits, do not provide persistent storage except for flash ROMs and can in general not be compared to desktop computers. As a consequence, they are less suitable platforms for running applications that are developed relying on a component-based framework such as CORBA or JavaBeans.

- Moreover, alternative light-weight approaches have not been developed due to the vast heterogeneity of platforms and programming languages.

Hence we often see proprietary, platform-specific solutions for coordinating components in conferencing endpoints, which leads to architectures where components cannot be re-used without manual adaptation. Instead they must be integrated by extending and changing the software and by developing adapters. Obviously, the integration of such components is a static integration that cannot be done at runtime, but is a manual operation at the development-time.

In this thesis, we describe a coordination framework that is designed for coordinating components of light-weight systems, including but not limited to specialized conferencing endpoints. In contrast to other developments, we provide a completely platform-independent approach that is based on a light-weight *protocol* for local coordination. The *Message Bus* (Mbus) is a message-oriented coordination protocol for the coordination of process groups, whereas the individual processes can be distributed on different hosts on a local network. Unlike typical

[1] H.245 is the call control protocol of the H.323 family of recommendations.

component frameworks, our coordination-based approach allows for dynamic associations of components in order to extend systems at runtime without the need for manual configuration.

In Section 1.1, we have a closer look at networked devices in pervasive computing environments, and in Section 1.2, we present a specific application example: the integration of conferencing endpoints with a desktop environment. In Section 1.3, we provide a first overview of our Mbus-based approach. We define the scope of this in Section 1.4 and provide an overview of the thesis in Section 1.5.

1.1 Networks of Personal Devices

While some years ago the typical computer-based working environment consisted of a single workstation or PC (later more often complemented by a laptop computer), we can today state an increasing deployment of information technology in working environments. Many users not only make use of workstations and laptop computers but also of systems such as personal digital assistants (PDAs), electronic organizers and (smart) mobile phones. Moreover, special purpose devices such as IP telephones, Web-radios, networked MP3-players, and digital video cameras with integrated computers and communication facilities are (though still slowly) proliferating in and around the desk area environment.

Interestingly, many of these devices provide some kind of network interface. Besides the workstation and the laptop computers many devices are already equipped with an Ethernet or Wireless LAN (802.11) interface and with an IP stack. Other devices, such as mobile phones and low-end PDAs provide connectivity using other technologies such as Bluetooth and Infrared. Recently, the term *Internet Appliance* (or *Networked Appliance*) has become popular for referring to devices that perform a specific function by the use of Internet technologies and for referring to devices that can be configured or controlled using Internet protocols. Moyer et al. [Moyer01] describe a *Networked Appliance* as "*a dedicated function consumer device containing at least one networked processor*" with the requirement

> "*to be able to communicate, at an application level, with a device within a local domain, such as a home network, from outside of that domain. The device may optionally use IP and may use an arbitrary command set, the details of which are not necessarily known outside of the domain.*"
>
> —Moyer et al.[Moyer01]

Gillet et al. provide a taxonomy of *Internet Appliances* and distinguish devices with respect to their flexibility and tailorability to specific users' needs. In this classification, the most powerful Internet Appliances are described as appliances that

> "*perpetuate the general-purpose functionality of today's PC because they can be composed together in different ways to support a wide array of tasks. Intelligence (protocols and the software that implements them) will be needed to assist the user in automatically negotiating and configuring the appropriate array of devices to support particular tasks.*"
>
> —Gillet et al. [Gillet00]

Gillet et al. expect these devices

> "*to be a generation of devices that develops when today's computer is decomposed into what the user sees as its constituent functional elements: display, keyboard, speakers, microphone, pointing device etc. These various components are currently organized by the operating system within a standardized PC architecture that is collected into a single box. With connection via LANs (wired or wireless) and the Internet, it will no longer be necessary to keep all of the cooperating components together in close physical proximity.*"
>
> —Gillet et al. [Gillet00]

In order to implement the necessary communication services for operating such decomposed and distributed devices, different communication paradigms and mechanisms are required:

- Device federations can be established dynamically and can provide dynamic membership after they have been established. Hence, services such as device and service discovery, device association and membership control must be provided.
- Devices may interact in different ways, beyond the acknowledged client-server RPC paradigm. For example, information may be shared between a group of devices, devices operate in parallel and may require asynchronous communication, and intermittent connectivity must be taken into account.

Saif and Greaves have investigated communication primitives for ubiquitous systems and conclude that

> *"... what is needed is a set of primitives that allow the decoupling of messages in either direction, both in space i.e. to a different host or process with independent properties, and in time, alleviating from the synchronous blocking semantics. The primitives should be generic, allowing for interoperability, and type checking should be done at runtime. All in all, it should provide a unifying, efficient abstraction for all the ubiquitous device control functions..."*
>
> —Saif and Greaves [Saif2002]

Later, we will analyze such scenarios in more detail and describe the properties of *coordination-based* protocols that are intended to provide a more appropriate and efficient way to control devices in ubiquitous computing environments. Instead of strict client-server-communication based control, coordination-based typically provide mechanisms for implementing control service for a group of entities of an application context that take dynamic membership, group communication, and a comparatively loose coupling of entities into account.

There is a vast heterogeneity in the hardware and software architectures of the different personal devices: A workstation and a laptop computer will typically be a PC-system with a desktop operating system such as Microsoft Windows XP, MacOS X or Linux. An IP telephone will be based on an embedded system, e.g., running Windriver's VxWorks or an embedded Linux variant. Typical PDAs are also embedded systems with a corresponding operating system such as PalmOS, Microsoft Windows CE or Lineo Embedix. Roughly the same holds for *Internet Appliances*, such as networked MP3 players. Naturally, these systems differ vastly in

their computing capabilities, display resolution and user input interfaces. Due to the different hardware architecture and operating systems they also differ in binary compatibility standards, operating system APIs, available programming languages and so on. For the more elaborate devices, the common denominator is probably the IP stack, i.e., the ability to communicate using Internet protocols.

In summary, we can currently observe the increasing deployment of multiple different computing devices (some of them are general purpose devices, some of them provide a dedicated service) into the personal (both work and home) environment of computer users. More and more of these devices provide a network interface; many of them implement Internet protocols. For establishing federations between device components in ubiquitous computing environments, a coordination-based approach instead of client-service RPC-oriented communication alone is needed.

1.2 Trends for Conferencing Systems

A lot of academic research and international standardization work has contributed to the fact that we are now at a stage where Internet-based *synchronous* telecooperation, i.e., multimedia conferencing, is gaining significant momentum. The techniques for real-time media transport over packet networks, i.e., the Internet, are well understood. Call-signaling and call-control protocols have been developed — starting off with a traditional telephony mind-set (H.323 [ITU96a]) but later followed by a more general, flexible approach that is targeted at session initiation in general (SIP [RFC3261]). The audio-centric variant of real-time media transport and conference control has become popular under the term *Voice over IP* (VoIP).[2]

It is now these base technologies for synchronous telecooperation that are currently revolutionizing the telephony industry by providing more efficient and more flexible transport mechanisms for audio communications. In corporate environments, we already see early signs of a displacement of traditional telephones (and corresponding infrastructure systems such as PBXs) by the introduction of IP telephones. Thus, the new technology does not only change the paradigm for long-haul voice transport, but also introduces new end user devices with the potential for offering new services besides voice conversation. For example, even the first generation of IP telephones offered multi-party voice conferences that are implemented by local mixing in the telephone itself. Instant messaging and personal presence state distribution has become part of some software telephones, e.g., Microsoft Messenger. Other examples for new services are the integration of new applications (new media types) into multimedia conferences[3]

[2]It should be noted that *Voice over IP* and *IP Telephony* are marketing terms that designate a voice-only specialization of *multimedia conferencing* (which can include communication using other media types besides audio).

[3]The increasing deployment of IP telephony sheds a new light on former approaches in the field of conferencing such as *Desktop Multimedia Conferencing* (DMC). Desktop Multimedia Conferencing systems have been designed with the idea of providing an *integrated* conferencing application that run on a user's personal computer, supporting multiple applications such as video and audio conferencing, shared whiteboard and other collaborative applications [Ott97]. DMC systems were intended as "next-generation telephones" with the objective to replace the traditional telephone.

Interestingly however, these so called *software phones* such as DMC systems (and prototypes of IP telephone software that run on desktop computers) have not been able to replace the traditional telephone — quite the contrary: in the majority of cases, IP telephony today is deployed by the use of hardware phones. This is apparently

or intelligent call routing based on user location.

Compared to the public telephone system's *intelligent network* approach, the introduction of new services for Internet-based conferencing system is much easier, because it generally does not require any changes to the network itself, but is merely a function of end systems. This extensibility is enabled by two factors:

1. The Internet service model provides an application-independent end-to-end packet-forwarding service that is implemented by the set of cooperating IP routers. These routers know nothing about transport protocols and applications and do not have to be changed in order to introduce new user applications.

2. An application independent and extensible framework for multimedia conferencing allows heterogeneous end systems to interoperate and to add new applications to conferencing systems. For example, the Session Initiation Protocol (SIP, [RFC3261]) has been designed for initiating sessions independent of the specific media types, which is accomplished by separating the call signaling and call control functions from application semantics for configuring specific application sessions.[4] In addition, SIP itself can be extended by new methods, thus enabling new services on the call signaling layer without requiring changes to central network elements.

One example of the development of new services and new applications for Internet telephony and conferencing systems is the *integration of telephony equipment into a user's work environment*, e.g., by controlling a telephone from a user's computer in order to relieve the user of switching from one device to another when initiating voice conversations. In principle, this concept of *Computer Telephony Integration* (CTI) is orthogonal to the network technology of the phone, i.e., it is not a genuine VoIP feature.

The CTI approach was developed to provide phone control and call management on PC systems in order to increase productivity in office and especially in call center environments. The basic idea is that the personal computer that is in many cases used for parallel tasks such as access to information systems, address book lookups and writing of call minutes anyway, is used as a *remote control* to the telephone in order to remove the necessity to permanently switch and move information between two devices. CTI has originally been invented for PSTN/ISDN telephones that are connected to a PBX (*Private Branch Exchange* — a telephone switching system) and are logically linked to a co-located computer.

Hence, the integration of telephones with other computing systems is not a particular new idea. However, the implicit availability of IP connectivity in IP telephones (and the fact that the corresponding functionality is comparatively easy to implement on their usually more capable system platforms) makes this technology a good candidate for providing the integration with a user's computing environment.

---

due to user expectations and user experience: The telephone as a dedicated device for audio communication is providing a certain degree of availability and robustness and is still the most popular communication device (in office and home environments). Another reasons for its popularity is the fact that it provides established usage and control metaphors such as lifting and putting down the receiver that users have adopted over time.

[4]SIP messages can carry a session description that describes conference configurations and endpoint capabilities.

With the advent of IP telephony, the communication infrastructure in enterprises is simplified significantly: separate lines required before for telephony vanish and all devices may employ the available (fixed or wireless) IP connectivity for all kinds of interactions. In particular, IP-based telephone sets are enabled to communicate directly with PCs, laptops, PDAs, and so forth — unlike CTI where a dedicated component (the user's PC) was needed to be hooked up with the PBX to gain indirect control of the user's telephone.[5]

Some of these more flexible approaches allow for direct interaction between PC and phone by means of phone APIs: the *Pingtel expressa* SIP phone, for example, offers a Java-based networked API to control the phone remotely providing a set of functions (sometimes described as *Desktop CTI*) such as *click-to-dial* (the possibility to initiate calls by activating control elements on a desktop PC with a graphical user interface, e.g., by selecting an entry in an address book or by activating a link traversal in a Web browser). Instead of the traditional CTI approach this example provides a direct peer-to-peer coupling between a feature-rich IP telephone and a co-located desktop PC. The availability of a common network infrastructure and common transport and application layer protocols facilitates the creation of new services that extend the functionality of the phone itself.

However, the integration of telephony and conferencing systems into users' working environments (and into corporate network environments) and the development of other new applications go much further than these examples for remote-controlling phones. One notable example in this regard is the recently introduced *OpenScape* architecture by Siemens and Microsoft [Siemens03a] (July 2003). OpenScape is a software platform for real-time communication that intends to increase the productivity of communication and collaboration processes by a concept called *presence-based, real-time multi-resource communications* [Siemens03b]. By *presence-based* the authors refer to supporting synchronous communication by distributing personal presence information, i.e., information about a user's current location and her current reachability status. The goal of the OpenScape platform is to enhance traditional voice-oriented synchronous communication (as provided by an IP phone) by utilizing additional functional components in a user's environment, e.g,, instant messaging and video applications. The integration of additional components can be used to implement multimedia conferences, to provide alternative methods of reachability and to gather personal presence information in order to determine the current presence and availability of the user. Voice and multimedia conferences can be supported by conference control applications that run on a user's desktop computer: the *workgroup collaboration application* allows users to initiate and manage conferences. The OpenScape platform pursues a central, server-based design, i.e., user devices have to register with a central server and the server manages central resources, such as conference servers (MCUs) and media gateways.

Summary of Observations

We have seen how the development of Internet-based conferencing technologies has created new perspectives for extensibility and for integrating conferencing systems into computing

[5]Nevertheless, this traditional CTI model has been taken up by some vendors of IP telephony equipment to implement similar services: Cisco's Call Manager, just one of numerous examples, is an IP-based central replacement for the PBX and provides a CTI-style interface to allow user PCs to indirectly control their phones.

In addition, traditional client-server APIs such as TAPI have been extended for controlling IP telephony devices [Microsoft99].

environments. Because conferencing systems have become stand-alone computing systems themselves they can be integrated directly into a user's office environment without necessarily requiring the service of central infrastructure components.

Considering the (emerging) pervasiveness of networked appliances that we have described in Section 1.1, it is interesting to see that conferencing applications are no longer tied to a single PC or to a closed special-purpose device such as a video telephone, but begin to appear as distributed systems that rely on the services of different devices and services in user's environment.

Internet-based synchronous telecommunication systems have been research topics for many years, because they can provide new, interesting features that go beyond the services of the traditional telephone system. Earlier *desktop multimedia conferencing* approaches have focused on *multimedia* conferencing as their main feature and have pursued the idea of *integrated* conferencing systems that were considered as *next-generation* telephones. Internet-based conferencing systems have additionally provided *large-scale* multi-party conferencing, relying on IP multicast as an efficient distribution network. Today, we can observe the adoption of IP telephony systems for office and corporate environments, whereas we have identified the possibility of *integrating* telephony and conferencing systems into personal and corporate environments as one emerging interesting characteristic. We have looked at early approaches such as the Openscape architecture that provides the concept of integrating stand-alone IP telephones into multimedia conferencing applications and the concept of disseminating telephone related presence information about a user into computing environments.

In the future, we expect an increasing deployment of digital, networked devices in personal, home and office environments, enabling new possibilities for device interaction and user-defined services. PDAs can be used to control IP telephones, laptop computers can offer video conferencing capabilities and workstations can offer complex shared applications such as shared CAD editors. The integration of conferencing endpoints into an organizational context, e.g., into an enterprise OpenScape system, can rely on client-server communication, because, in general, there will be a static configuration of server systems, e.g., the OpenScape presence management server, and client systems, e.g., the endpoints of individual users. However, the integration of multiple components within a user's domain, i.e., different networked devices that can perform dedicated functions within a conferencing application context, relies on rather dynamic associations of entities and provides *peer-to-peer* communication relationships, because there are no statically defined client and server systems. In the following section, we will look at a sample decomposed endpoint system.

In order to integrate all these different personal devices into a single coherent application, some form of *coordination* is required. In this thesis, we discuss requirements for such a coordination framework and present an approach that follows the concept of message-oriented, asynchronous coordination of application components. This framework is intended to provide coordination services not only for inter-device scenarios as described above but also for intra-device communication, e.g., for a decomposed conferencing system. The modular structure and the strict separation of functionality that we have observed for conferencing systems make these devices an ideal candidate for a decomposed, distributed approach. Although current implementations exhibit a modular structure, they do not follow a component-based design, as they rely on proprietary, platform-specific interfaces, which requires manual adaptation and impedes component re-use.

In this thesis, we discuss the development of a component-based approach for the design

of conferencing systems that relies on the message-oriented coordination framework. In this context, we present an architecture that enables the construction of a conferencing system of existing application components and allows a conferencing system to take advantage of additional components that may be offered by external devices, e.g., in a pervasive computing environment as described above. In the following, we will briefly highlight the main ideas of this thesis, followed by a classification of this subject with respect to the relevant research and engineering areas.

1.3 A Message Bus for Local Coordination

Having briefly looked at some aspects of implementing conferencing applications focusing on IP telephony endpoints and their integration into corporate and personal infrastructures, one might ask: "Given the growing pervasiveness of computing devices in a user's working environment and given the observed trends for the implementation of conferencing systems and their integration into corporate and personal environments, what could a suitable architecture look like and what requirements for communications protocols would that bring about?"

As an example, we will sketch an architecture for the component-based implementation of conferencing applications that takes into account the need for integration into existing infrastructures.

1.3.1 A Decomposed Multimedia Conferencing System

As we have mentioned in the beginning, multimedia conferencing systems such as IP telephones are software systems that can conceptually and (usually) technically be divided into several components. Figure 1.1 is an example of a model with different components for the user interface (display and keypad) and a phone application module that provides the main functionality.

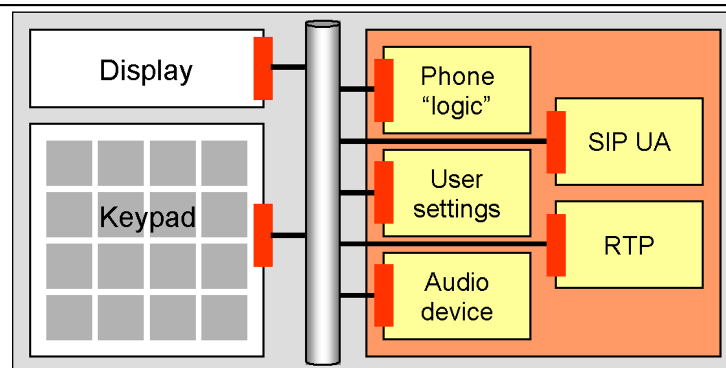

Figure 1.1: Decomposed IP telephone

When we increase the level of detail, the phone application model exhibits subcomponents that can be identified. For example, an IP telephone will provide an implementation of a call-signaling protocol (`SIP UA` in Figure 1.1) and a media engine (`RTP`) that receives and transmits

media streams. In many cases there will also be a controlling module (`phone logic`) that manages the overall system state and coordinates the other components.

Different control relationships and information flows between modules can be identified. For example, it is conceivable that a control component coordinates modules by invoking procedures with well-defined semantics, which have been specified in an interface description. Furthermore, information may be distributed between components using a group communication model that provides for event-notification. For example, events such as (`off-hook/on-hook`) may be generated by a user interface component and be distributed to one or more components within the telephone system.

There are different approaches towards implementing the exchange of control and status information. For example, if all components are implemented as modules of one application instance, one will probably rely on API functions, i.e. deploy a model of decomposed software modules of a single application. If the decoupling of components is more sophisticated, there might be components that are represented by independent program contexts (i.e., processes, threads). In order to coordinate these components, a communication infrastructure, i.e., an interprocess-communication facility, will be employed, that could be implemented by platform-specific means, such as UNIX pipes, message/event queues or socket communication.

In this thesis, we propose to employ a *distributed* approach, where components appear as rather independent entities that are coordinated through a *coordination protocol*. In this model, components are not linked into a single program (as object code modules) but are considered independent processes that may run in parallel and do not even have to run on a single processor.

1.3.2 Integration into Computing Environments

The distributed coordination approach is also the key to enlarging the scope of a conferencing endpoint beyond the borders of a single computing system. The Mbus coordination protocol that we describe in this thesis is not only targeted at intra-system coordination but provides inter-system coordination mechanisms as well. The idea is to coordinate application components of distributed applications, where the components are distributed in a domain, i.e., a local computing environment, for example, a user's desk area environment. One sample application of inter-system coordination is the *integration* of a conferencing system into a user's environment.

The integration of a conferencing system into its environment that we have described in Section 1.2 can be illustrated by the following example: the coupling of a VoIP phone with multiple components on a user's workstation. The capabilities of the audio conferencing system, i.e., the phone, are extended to other applications such as video conferencing and application sharing by employing functional components that reside *outside* of the phone system. Figure 1.2 depicts a conferencing system that spans multiple hosts: The IP telephone's components on the left are coupled with additional application components on a co-located workstation (such as a media engine, a shared application and a user interface component). In addition, there is a PDA system that provides an instant messaging application and an additional user interface component.

In a deployment scenario, the phone system would locate the available additional modules and integrate them into its application context. During this integration process, the phone would learn the capabilities of the new components and configure them appropriately. When initiating new conferences, the phone, representing the complete conferencing system, is now able to ad-

vertise new applications to other endpoints, e.g., the video conferencing application. During the initiation of the conference, the phone's call signaling component would negotiate capabilities and exchange transport addresses with other endpoints. After this initiation has completed, the controlling module on the phone configures the media engines corresponding to the outcome of the conference initiation process. I.e., it configures its inbuilt audio engine and the external video and shared application engines. During the conference additional communication between the modules will take place, e.g., for implementing conference control functions. When the conference is terminated, the call signaling engine of the phone, i.e., its control module informs all the other modules, e.g., to terminate their media sessions.

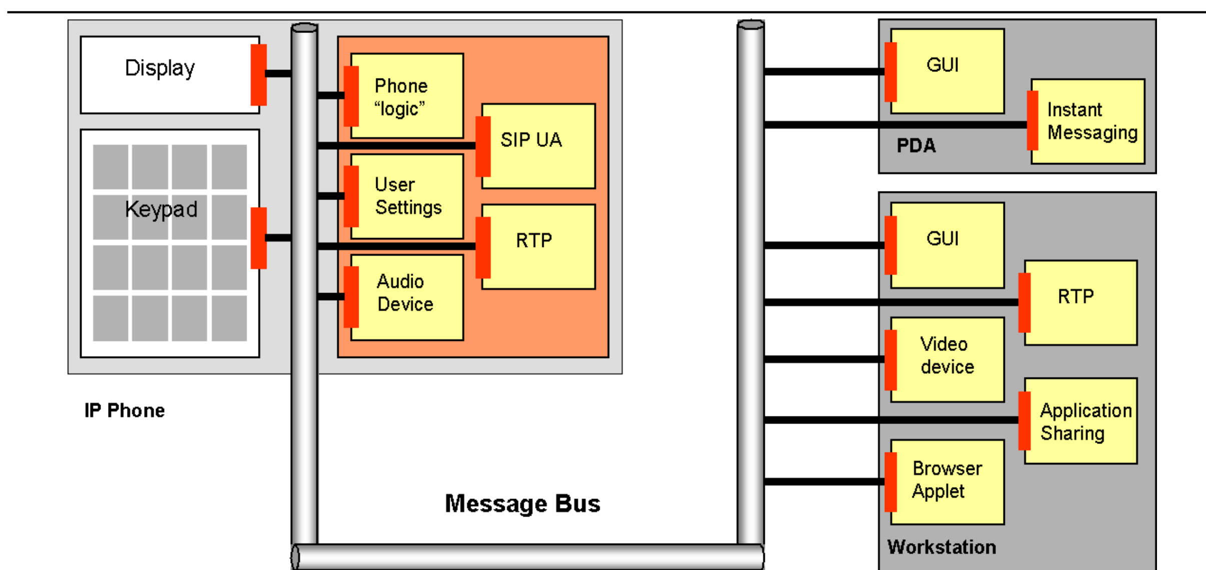

Figure 1.2: Integration of external functional components

It is conceivable that this group of components provides dynamic membership, i.e., not all components join the communication session at the same time. For example, the PDA could be turned on at some stage and the corresponding components would have to be integrated into the application. Analogously, the PDA could also be turned off (or simply leave the "geographic scope" of the application) at some time. This scenario suggests that the "classical", system-local approaches for decomposing conferencing endpoints into modules, such as the software library approach or the local inter-process-communication approach, are not applicable for extending the scope of conferencing systems. Instead, a *distributed approach* must be chosen. Such an approach must be component-based, i.e., third-party deployment of components by user-defined applications must be possible, and it must accommodate distributed conferencing endpoints and allow for their integration into pervasive computing environments as we have described in Section 1.2.

Furthermore, it is obvious that the independent deployment of modules regardless of their implementation details, such as platform and programming language, requires an *acknowledged* if not *standardized* coordination mechanism that guarantees interoperability between the different components and thus allows systems to locate and integrate components from third-party manufacturers.

In this thesis, we therefore propose to *generalize* and *standardize* the interfaces between intra-system and inter-system components. The main part of this thesis discusses the development of a general-purpose message-oriented coordination mechanism that is designed to be applicable as a foundation for a distributed, component-based system design for conferencing systems. The heart of this framework is the *Message Bus* (Mbus) — a light-weight protocol for coordinating distributed applications in local networks. In our approach, application components provide well-defined Mbus interfaces and use the Mbus protocol to communicate with other functional components independent of their host affiliation.

1.4 Research Areas

The discussion in the previous sections has already shown that the work we describe in this thesis is related to the general area of distributed systems and to the field of multimedia conferencing. In detail, we touch upon the following research issues:

Distributed Systems: In the area of distributed systems, significant research results have been produced for the development of fault-tolerant communication, group communication, and abstractions such as the RPC paradigm that we have already mentioned.

The solutions that have been developed in this area can be viewed as *building blocks* for the definition of an application layer coordination mechanism. Later in this thesis we will therefore describe relevant existing algorithms and protocols that have influenced the Mbus design.

In addition, there are different design alternatives for distributed systems. We will later contrast traditional mechanisms such as RPC communication with group communication systems and try to match the solutions against our requirements.

In this thesis, we essentially propose an alternative design for a group communication system that is based on a loose coupling of communication peers and relies on asynchronous, message-oriented communication.

Multimedia Conferencing: To some extent the application area of multimedia conferencing can be considered as a specialization of the distributed systems area. Multimedia conferencing relies on technologies for real-time transport for multimedia information, reliable group communication, representation of media data, security and many others.

The specific solution that we present in this thesis aims to provide a new design for conferencing endpoint architectures and is thus strongly related to the ”wide-area” conferencing architecture and the relevant technologies that implement this architecture. For example, in order to define interfaces for media engines in a component-based conferencing system, the concept of individual media streams and media engines within a conference has to be considered, and the characteristics of the employed transport protocol, e.g., the *Real-Time-Transport-Protocol* (RTP), will influence the design of communication interactions and service interfaces.

A specific element of multimedia conferencing system is *conference control*, the aspect of initiating and terminating conferences, but also *conference-course control*, i.e., protocols for controlling the behavior of application endpoints *during* a conference.

Different conference control protocols and architectures have been defined. In this thesis we build upon the existing work for local coordination architectures and take care that our solution fits well into the existing conferencing architecture.

In this thesis we try to maintain a *systems view*, considering requirements and mechanisms for different protocol layers *and* application areas. This means that we do not restrict ourselves to isolated research topics such as group communication but consider architectural implications including implementation and integration aspects, too. Particular architectures of interest are the Internet architecture and the Internet Multimedia Conferencing architecture.

In some parts of this thesis, we rely on theoretical work in the different research areas and describe a new protocol design, corresponding protocol specifications, implementations and applications. The focus lies on the design of the Mbus protocol and its application to different application areas — first of all to the multimedia conferencing domain. The protocol itself and the derived specifications and implementations have been deployed in multiple projects, which will be described at the end of this thesis. The Mbus protocol specification itself has been published as RFC 3259 [RFC3259].

The research and engineering results that are described in this thesis are based on work done by the author that was partly conducted in cooperation with national and international cooperation partners and has been reviewed and discussed in international standardization bodies, namely the IETF. In the following, we briefly list the main original results. Based on the conceptual work that is described here, e.g., the Mbus transport specification, additional implementations have been created independently. In this thesis, we name the contributors in those sections where we describe the respective developments.

Mbus Transport Specification: The Mbus transport specification that is described in Section 6.2 has been developed by the author in cooperation with Jörg Ott and Colin Perkins. The specification has been reviewed by the MMUSIC working group of the IETF and has been published as RFC 3259.

In particular, the author has contributed the security concept, the interval calculation for the `mbus.hello` timer, the specification of the scoping mechanisms and the Mbus parameter configuration.

Mbus Higher Layer Interactions: The specification of the Mbus Higher Layer Interactions that are described in Section 6.3 has been created by the author.

Dynamic Device Association: The Dynamic Device Association concept (Section 6.4) has been developed by the author in cooperation with Jörg Ott.

Mbus and Ad-hoc Communication: The Mbus Ad-hoc Communication concept (Section 6.5) has been developed by the author in cooperation with Jörg Ott.

Local Coordination Architecture: The Local Endpoint Coordination Architecture (Section 9.1) has been developed by the author, supported by numerous discussions on endpoint and gateway architecture in our research group.

Session Description and Capability Negotiation: The work on SDPng has been performed by the author in cooperation with Jörg Ott and Carsten Bormann.

Implementations: The author has developed a C++ and a C implementation of the Mbus transport protocol. The C++ implementation and additional infrastructure components developed by the author have been used as fundamental building blocks in many Mbus based applications.

1.5 Structure of this Thesis

This rest of this thesis is structured as follows:

- Chapter 2, *Conferencing Architectures* provides a discussion of multimedia conferencing technologies — context technologies for our coordination framework — with a focus on conference control architectures;
- Chapter 3, *Use Cases and Requirements* presents two detailed use cases and infers a set of requirements for the Mbus protocol;
- Chapter 4, *Foundations and Related Work* analyzes fundamental work, e.g., conceptual building blocks for coordination of distributed systems, and investigates related work, e.g., existing coordination protocols for conferencing systems;
- Chapter 5, *Architecture* introduces the Mbus framework architecture;
- Chapter 6, *The Mbus Framework* is the main part of this thesis and describes the Mbus protocol, additional functions such as bootstrapping and enhancements for the deployment of the Mbus in special environments such as ad-hoc communication scenarios;
- Chapter 7, *Mbus Implementations* describes available Mbus implementations and reports on deployment experiences;
- Chapter 8, *Evaluation* provides an evaluation of the Mbus transport mechanisms and describes simulation and test results;
- Chapter 9, *Mbus in Conferencing Systems* presents Mbus application semantics that we have defined for call-control and other applications;
- in Chapter 10, *Mbus in Projects* we look at the deployment of the Mbus protocol and the corresponding application semantics in real-world projects; and
- in Chapter 11, *Conclusions* we recapitulate the main results and summarize the lessons that we have learned in developing and deploying the Mbus protocol.

Chapter 2

Conferencing Architectures

In this chapter, we describe the different architectures for multimedia conferencing, with a focus on *conference control* functions. In [Kirstein93], Kirstein et al. provide a first definition of *multimedia conferencing* and state that it implied the sharing of voice, video and computer data amongst geographically distributed groups of people, whereas the *sharing of computer data* refers to shared applications such as shared workspaces. The evolution of this concept and the evolution of telephony-based conferencing approaches has led to the development of *architectures* for multimedia conferencing, such as the H.323 family of recommendations and the Internet Multimedia Conferencing Architecture.

For multimedia conferencing on the Internet, both architectures rely on RTP, the Real-Time Transport Protocol [RFC3550] [Perkins03], for the transmission of real-time media data, but differ significantly with respect to the provided control services. In general, the term *conference control* refers to control services that are available for establishing and during the course of a conference, i.e., *conference course control services*. Some examples of control services are:

Conference initiation: Conference initiation refers to all the services that are required to establish a multimedia conference, such as distributing conference configurations that specify the application types of a conference and their parameters.

Depending on the type of the conference, the initiation may also involve *call signaling* and *call control* services, including user location and call setup procedures.

Membership management: There are different aspects to membership management: a basic service is the maintenance of membership lists, which can be realized on different levels. In loosely coupled sessions, membership information is typically gathered from the membership information that is conveyed as part of the media session communication, e.g., relying on RTP mechanisms.

Conference course control protocols can provide a more precise management of membership information, e.g., by providing unique identifiers for participants that are mapped to the RTP SSRC identifiers. These services allow to decouple the conference membership from the membership of media sessions, i.e., a participant can be a member of the conference without having to join all media sessions. The membership management for conference course control has typically higher requirements with respect to consistency and reliable communication than the soft-state-based media session membership information.

In addition, some architectures that provide a very tight coupling rely on membership control, i.e., by controlling admittance to conferences and by offering services such as excluding members. A conference control architecture can enforce such measures by relying on cryptographic methods, e.g., on group security mechanisms for membership control in multi-party conferences. A central group coordinator could be employed to control the admittance to the conference and to enforce the implementation of decisions that affect group membership.

Floor control: Controlling the right to speak (or to be active in the conference by other means) is another example of a conference course control service. In loosely coupled conferences, there is usually no way to regulate participants' activities in the conference, except for informal, receiver-based mechanisms such as filtering of audio sources in an audio session.

Conference course control mechanisms can be used to implement the concept of floor control, e.g., by coordinating the application instances at all participants.

Ordered termination of conferences: Whereas loosely coupled conferences that are represented by one or more media sessions do not require dedicated mechanisms for implementing conference termination, this can be different for tightly coupled conferences. The conference can be conducted on some kind of formal policy that limits its duration (similar to a scheduled meeting) or the conference could have allocated resources that need to be released after the conference has ended. In multimedia conferences that are implemented by multiple application instances at the respective participants' endpoints, an ordered termination mechanism clearly is a useful mechanism for automating the usage of the conferencing system.

Obviously, there are different possibilities for implementing these services. However, we can note that services such as precise group membership management and enforced floor control typically require a reliable group communication mechanism that can be used to convey the corresponding control information.

Scheifler, Gettys et al. have described the concept of *separating mechanisms and policy* [Scheifler92], which has become an acknowledged design principle for creating larger (software) architectures.[1] This design principle is also applied to conference control by different conference control specifications: the conference control services can be viewed as basic building blocks that a specific conferencing system can rely on to implement a certain *conference policy*. For example, different types of conferences could be defined, such as audio-video broadcast, lecture and workgroup discussion, each of which requiring different rules with respect to membership management, floor control and other services. A *conference policy* can be defined in order to specify the usage of basic conference control mechanisms to implement this conference type.

The way how these services are implemented depends on the general conferencing architecture that is used. Two main standardized approaches can be distinguished today: The ITU-T's H.323 family of standards and the IETF's multimedia conferencing architecture. Both provide

[1] A *mechanism* determines *how* to implement a certain functions, whereas *policies* decide *what* will be done, relying on available mechanisms.

some similarities — e.g., H.323 has adopted the IETF's *Real-time Streaming Protocol* (RTP, [RFC3550]) for real-time media transport over packet-based networks — but there are a number of differences.

The H.323 approach is often characterized as an architecture for *tightly coupled conferences*, i.e., conferences with a certain degree of membership management and other control services. To some extent this notion also stems from the originally centralized architecture of ITU-based conferencing systems that employed centralized models of control. The IETF-evolved protocols are largely based on IP multicast and its service model, which has led to a rather *loosely coupled* approach that has shown to be more manageable and scalable for wide-area multi-party conferences. It should be noted that the evolution of protocols and architectures from both domains has blurred the distinction based on the tightness of coupling to some extent, as H.323 has adopted some acknowledged IETF protocols such as RTP and the IETF domain has also put forth some efforts that address conference course control.

We will present the two architectures in Section 2.1 and Section 2.2 focusing on conference control services. For a complete description, the reader is referred to the corresponding specifications that are named in the respective sections. In Section 2.3 we summarize the lessons we have learned from the different approaches.

2.1 The H.323 Recommendations

The *International Telecommunication Union* (ITU) has released a set of standards for audio- and video-conferencing over different networks. The H.320 standard suite (1990, [ITU93a]) specifies conferencing in ISDN narrow-band networks and a variant has been adopted as H.321 (1995, [ITU95a]) for broadband ISDN. As ISDN-based technologies, H.320 and H.321 relied on a fixed-bit-rate model, in general using G.711 [ITU88] and G.723.1 [ITU96e] as audio codecs such as and H.261 [ITU93b] as a video codec (H.261 provides fixed-bit-rate video coding, which is scalable in terms of bit-rate and realized quality.) The conferencing model can be characterized as *tightly coupled*, i.e., the set of members in a conference can be quite thoroughly controlled due to the nature of the ISDN telephone system. In the connection-oriented ISDN-environment, multi-party conferencing with H.320/H.321 has usually been achieved by relying on so-called *Multi-Point Communication Units* (MCUs), central devices in the network that perform call signaling and media distribution, including services such as audio-mixing and video-switching.

The H.323 umbrella standard has been released in 1996 and specified multimedia conferencing over packet-based networks without guaranteed quality of service, such as the Internet. The H.323 standard (as well as the other H.32x standards) contains several other specification components:

- H.225.0 [ITU00] is H.323's *call signaling* protocol, used to initiate conferences. H.225.0 relies on Q.931, the call signaling protocol for ISDN, and provides mechanisms for setting up call relationships between endpoints. For example, it defines messages such as *call set-up*, *proceeding* and *connect*. H.225.0 is described in more detail in Section 2.1.1.

- H.245 [ITU95c] defines *call-control* services, i.e., services for managing the multimedia communication once the fundamental call relationship has been established through

H.225.0 communication. The main function of H.245 is the exchange of *terminal capabilities*, i.e., capability descriptions of endpoints and their application entities, and the management of so-called *logical channels*, i.e., communication channels for conference applications, e.g., an RTP-based audio session. H.245 is also described in more detail in Section 2.1.1.

- In addition to audio and video conferencing, H.323 also provides the so-called *data-conferencing* services that are specified in the T.120 suite of standards [ITU96b]. The T.120 suite specifies mechanisms for shared applications such as shared whiteboards and shared desktops.

  An important element of T.120 is the conference control functionality specified as *Generic Conference Control* (GCC) in the recommendation T.124 [ITU98b]. We will analyze the conference control in more detail in Section 2.1.3.

  For the control and data conferencing communication, T.120 provides the concept of *channels* as an abstraction for multi-point communication facilities. Channels allow addressing different groups of receivers in a conference. The *Multipoint Communication Service* (MCS) that is specified in T.122 [ITU98a] and T.125 [ITU98c] provides the multi-point communication that can be realized using different mechanisms. The most common MCS implementation relies on multiple connections between endpoints that can form trees for routing the data. MCS is also discussed in detail in Section 2.1.3.

- H.235 [ITU98f] defines security mechanisms for H.323 and other H.245-based multimedia terminals.

For audio and video transport, H.323 has adopted the *Real-time Transport Protocol* (RTP, [RFC3550]) that has been standardized by the IETF. RTP provides the possibility to use native multicast services of the underlying network instead of using centralized MCUs for real-time data transport. However, control functions, such as video stream selection are still located in the MCUs. It should be noted that, in the following sections, we will not describe fundamental technologies such as IP, IP multicast and RTP in detail. In [Perkins03], Perkins provides a detailed description of RTP and its underlying concepts such as the *Light-Weight Session Model* (LWS, [Mccanne98]) and the principle of *application layer framing* (ALF, [Clark90]).

2.1.1 Conference Initiation

As indicated above, conference initiation in H.323 is separated into *call signaling* (H.225.0) and *call control* (H.245), and the call control communication takes place *after* the call setup (through call signaling communication) has completed.

In fact, the H.225.0 signaling services for the communication between endpoints are augmented by the H.225.0-RAS protocol (*Registration, Admission, and Status Protocol*), which is used to obtain services from so-called H.323 *gatekeepers*, intermediate systems for call-routing and authorization services. For example, for inter-domain H.323 communication, H.323 endpoints typically contact Gatekeepers first (trough H.225.0-RAS communication) before initiating calls trough H.225.0-Q.931 (the actual call signaling protocol). In this thesis, we cannot describe H.225.0-RAS in detail. In short, the H.225.0-RAS communication between terminals and gatekeepers provides services such as user registration, admission control and address

resolution. In addition, it is possible to route calls, i.e., the H.225.0 communication between endpoints, via gatekeepers, which means that gatekeepers act as proxy servers that provide call routing and user location services. This model is called the *gatekeeper-routed call-signaling model*.

H.225.0-Q.931 — the actual call signaling protocol — provides messages such as `Call Setup`, `Call Proceeding`, `Alerting`, and `Connect` to support the establishment of call relationships. Figure 2.1 depicts the H.225.0-Q.931 call setup process. The `Call Setup` message is sent from a caller to a callee and provides H.323 address information, e.g., an E.164 address, for both source (caller) and destination. In addition, it provides an H.245 address — a transport address that specifies how the calling endpoint wishes to establish the H.245 communication. Furthermore, the `Call Setup` message specifies the *conference goal*, i.e., the type of the call setup message, e.g., `create` for starting a new conference, `invite` for inviting a party to a new conference or `join` for joining an existing conference.

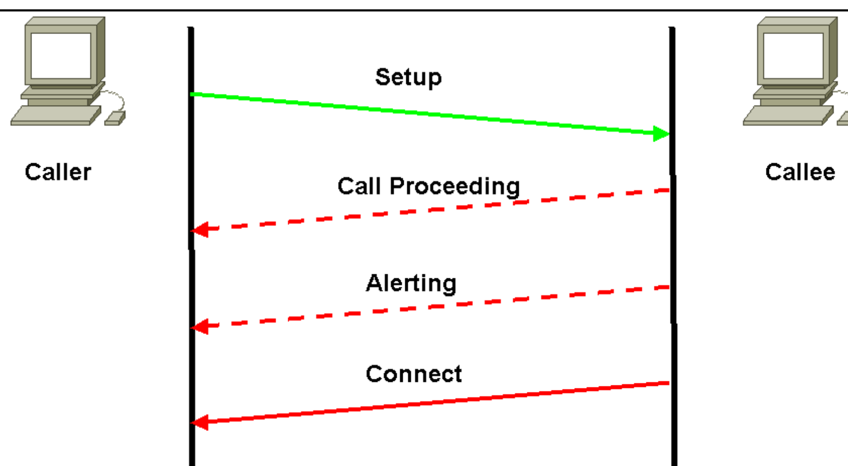

Figure 2.1: Basic H225.0-Q.931 call setup process

In essence, the main goal of the H.225.0 communication is the establishment of a H.245 communication channel that can be used for the further call control communication, which consists of *master-slave determination*, capability exchange, and opening of logical channels. The master-slave determination is a process to assign each of the two parties a well-defined role (master or slave) for the further H.245 communication, which is necessary, as H.245 communication is asymmetric and requires a *master* that coordinates resource conflicts. H.245 is limited to a fixed set of capability types such as audio capabilities and video capabilities, distinguishing *receive capabilities* (describing eligible configurations for receiving application data) and *transmit capabilities* (describing configurations that an endpoint supports for sending application data).

Logically, an H.323 conference consists of different sessions, e.g., an audio communication session and a video communication session, which are denoted *application sessions*. For each application session, a logical channel must be opened. Opening a logical channel implies resource allocation for the specific application session and the configuration of application entities. H.245 also specifies a set of control messages for logical channels pertaining to video sessions, e.g., update requests for H.261 video streams. In principle, the capability exchange

and the opening of logical channels takes places after the H.225.0 call setup, and each application session can only be used after the corresponding logical channel has been opened.

Because this sequential procedure can lead to significant delay between the time an initial H.225.0 setup message is sent and the time media sessions become functional (e.g., audio packets are sent for a phone call), H.323 version 2 provides an optimization called *Fast Start*. For the *Fast Start* optimization, logical channels are essentially opened before the H.225.0 connection setup has completed. Therefore the H.225.0 `call setup` is extended by a `fastStart` field, providing logical channel data structures, which describe the caller configuration for a given application session, e.g., an RTP audio session. If the callee accepts this configuration, it can use the configuration directly, without going through the regular H.245 logical channel setup, and can send media immediately. The Fast Start optimization has been developed for accommodating IP telephony scenarios, where long connection setup times are not acceptable — however, it is an optional enhancement that is not necessarily supported by all endpoints.

In summary, the H.323 conference initiation service is divided into call signaling and call control functions, whereas this division has shown to be inefficient for fast call setup. The H.225.0 call signaling functions can be divided into the actual call signaling communication (H.225.0-Q.931) and the gatekeeper communication (H.225.0-RAS). The H.225.0-Q.931 protocol is an adopted version of the ISDN-Q.931 signaling protocol and thus provides telephony inspired messages such as `Call Setup` and `Proceeding`. H.225.0-RAS is intended for the communication with gatekeepers that can provide authorization, user registration and user location services. H.225.0-Q.931 signaling can also be routed via gatekeepers, which act as application layer gateways and perform call routing functions.

2.1.2 Data Protocols for Multimedia Conferencing

In this section, we describe the services provided by T.124, the H.323 *Generic Conference Control* service (GCC) and the underlying *Multipoint Communication Service* (MCS) in more detail. T.124 is part of the T.120 family of protocols that has has originally been defined for H.320 (ISDN-conferencing) and later been adopted for H.323.

In the T.120 model, a conference consists of multiple *application protocol sessions* in which peer *application protocol entities* communicate. An *application protocol entity* is the instantiation of an application protocol in a terminal or in an MCU and is deployed by a *user application* (it is not the application itself). *User applications* and *application protocol entities* reside on *nodes* that are conferencing endpoints (terminals or MCUs). Each node provides a *node controller* that deals with the control aspects of the conference that apply to an entire node.

Figure 2.2 depicts this model for a T.120 node graphically. T.126 is the *Multipoint Still Image and Annotation Conferencing* protocol specification, T.127 is the *Multipoint Binary File Transfer* specification, and T.128 is the *Multipoint Application Sharing* specification [ITU98d]. The latter three are standardized application protocols within the T.120 suite. In addition, a conference may provide *application protocol sessions* using non-standard application protocols. The user applications and the node controller deploy a control agent that provides the T.124 control services and is thus called *GCC provider*. These services are specified in recommendation T.124. It should be noted that local coordination between user applications and node controllers is not covered by T.124. Figure 2.2 shows that the node controller interacts with the GCC provider which in turn employs an MCS access point for the group control communication within the conference. The *application protocol entities* also make use of the MCS for the

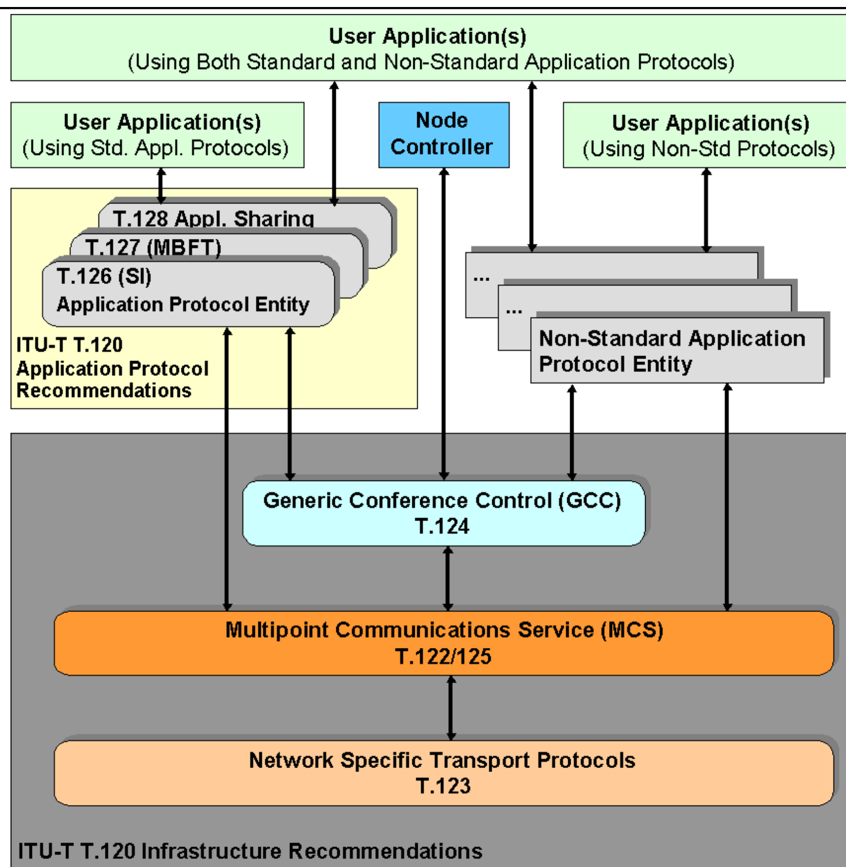

Figure 2.2: The T.120 framework [ITU96b]

group communication in their corresponding application sessions.

MCS provides the concept of *MCS domains* — a tree of MCS connections among *MCS providers*. An *MCS domain* maps to a conference and sets the boundary for multi-point communication between application protocol entities within a conference. In analogy to GCC, an *MCS provider* interacts with application entities on a node and underlying transport services and communicates with peer *MCS Providers*. Each MCS domain provides a *Top MCS Provider* that manages the channels, user identities and token resources for the domain. Figure 2.3 depicts an MCS domain with an MCU as the Top MCS Provider. In this figure there is only one node with multiple MCS connections (the Top MCS Provider). However, it is also possible to have multiple nodes and arrange them hierarchically in a tree structure.

MCS provides the concept of channels that can be used for individual application protocol sessions or the GCC communication. There are multicast channels that can be used to send data to all or a subset of the clients in a domain, and there are single-member-channels that can be used to address messages to single users. Two types of message transport are provided by the MCS:

Simple Send: The *simple send service* provides reliable one-to-many message transport without global ordering. In an MCS-tree-structure the messages are delivered directly, without being relayed over the Top MCS Provider.

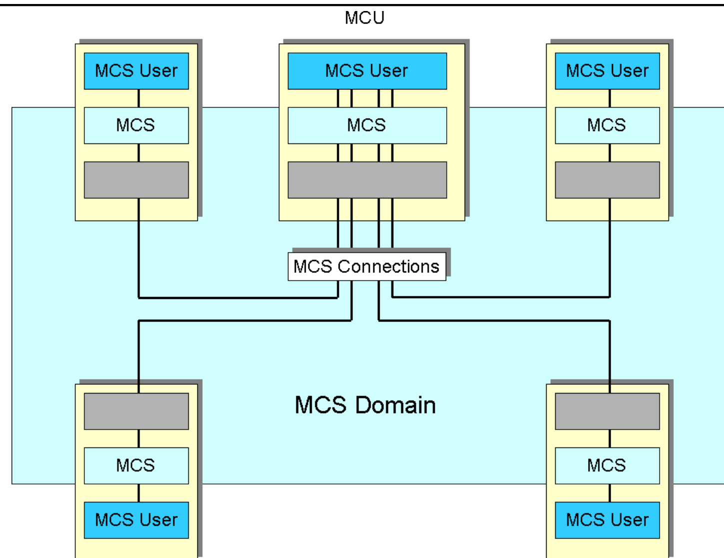

Figure 2.3: An MCS domain

Uniformly Sequenced Data Transfer: This transport class offers global ordering by routing all messages to the central Top MCS Provider first. The Top MCS Provider then forwards the messages to subordinate MCS nodes. This ensures that two messages that are sent by different MCS users arrive at all destination MCS users in the same order.

The *token concept* is the basis for realizing control functions, e.g., mutually exclusive access to resources: a token can be associated with a resource, e.g., the floor (the right to speak) in a conference, and MCS provides mechanisms to request and transfer tokens.

It has been noted by Ott in [Ott97] and later been shown by Trossen in [Trossen00] that the MCS model of relying on a series of point-to-point connections creates serious scalability problems and makes MCS and thus GCC essentially unusable for large-scale conferences. The main argument is that the number of control messages and the required bandwidth required to transmit these messages grows exponentially with the number of nodes, thus introducing significant delays and packet loss. With these observations in mind, Ott has described an advancement of MCS in [Ott97] that allows for efficient multipoint communication in native multicast environments.

2.1.3 Conference Course Control

It should be noted that GCC specifies a distributed replicated database for most of the conference state, e.g., the Conference Roster and the Application Roster: Each GCC Provider maintains a copy of the state. State changes are signaled by sending a request of the *MCS Uniformly Sequenced Data Transfer* class (in order to achieve consistency) to all other nodes. This means that all messages related to state changes are via the Top MCS provider.

T.124 provides the following generic conference control services:

Conference Establishment and Termination: GCC provides a directory service that lists the

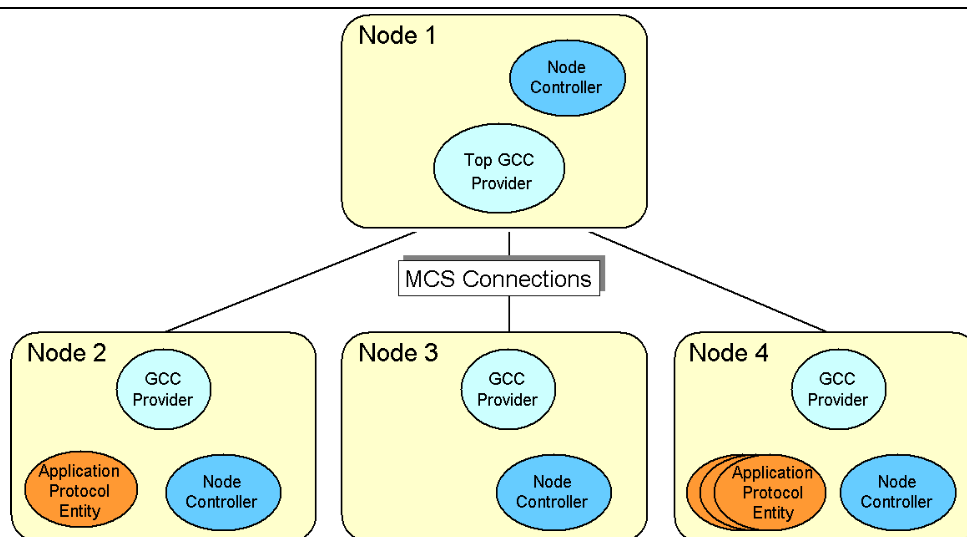

Figure 2.4: GCC components in an MCS domain

conferences that are offered at one MCU. This is not a global directory service as the SAP service in the Internet (see Section 2.2) but a local service only.

For establishing a new conference, an initiator (a conference participant or an administrator) specifies the conference parameters and policies in a conference profile and sends a `GCC-Conference-Create` request to another node (either to an endpoint or to an MCU). The node that receives the corresponding indication automatically becomes the Top GCC Provider for the conference. The conference profile contains general information on the conference such as the conference name but also defines the policies, for example whether the conference is locked for new participants, whether a password is required for joining the conference or whether the conference is started in *conducted mode* (with a designated conductor that controls the course of the conference).

New nodes may be added to an existing conference at any time. It is also possible to *transfer* participants from one conference to another, which is used to implement the merging and splitting of conferences.

Conference Roster: The *Conference Roster* is a list of all participants of the conference and is maintained by each node. The roster contains information on each node such as the name of the node and the name of the participants at the node. Each node announces its presence upon joining the conference, thus enabling existing nodes to learn the existence of the new node.

Application Roster: In addition to the Conference Roster, GCC also provides support for maintaining the *Application Roster* — a list of all application protocol entities that are available at the nodes. Upon joining a conference, a node sends its local list of application protocol entities to all other nodes. This information is integrated into the overall *Application Roster* that is replicated at each node. The roster can include application protocol entities that are based on standard application protocols as well as entities based on non-standard application protocols.

The *Application Roster* does not only list the application protocol entities but also provides mechanisms for each node to specify the *capabilities* of each application protocol entity. GCC defines rules that specify how to *collapse* the capabilities from all the application protocol entities in the conference to a common subset.

GCC provides the concept of *application-independent* capability description, i.e., an application entity's capabilities can be expressed using generic description mechanisms. Each capability is tagged with a type specifier: `logical`, `minimum`, and `maximum`. The type specifies how this capability has to be processed in a *capability collapsing* operation.

This concept allows GCC to process capability description without application knowledge to determine and to distribute a usable configuration without the need for a complete exchange of all capabilities between all nodes: instead GCC can determine the commonly supported configurations and inform application protocol entities.

Application Registry: The *Application Registry* is a central database that is hosted at the Top GCC Provider and allows applications to store and retrieve application-specific data.

Conference Conductorship: GCC provides the concept of a *conductor for a conference* that can control the course of a conference. In a conducted conference, application protocol entities operate in a special (application-specific) mode, for example certain operations may have to be approved by the conference conductor before the corresponding requests can be sent to other entities.

Conference Conductorship is supported by a dedicated token type: a node controller at a node may request the conductor token by sending a `GCC-Conductor-Assign` request.

Miscellaneous Functions: GCC supports the notion of *timed conferences*, i.e., conferences with a specified maximum duration, and so there are mechanisms to retrieve and to announce the remaining time in a conference, to request more time etc.

Summarizing, we can state that the T.120 GCC model is clearly an approach for tightly coupled conferences with a strict membership and application management that is implemented by the means of the GCC services. GCC as well as application protocols are layered on top of MCS, the multipoint communication service, which in turn layers on top of specific transport protocols.

The GCC architecture does not specify mechanisms by which a *node controller* can implement the conference control and coordinate the application entities. Any such mechanisms are a local matter of the conference applications.

2.2 Internet Multimedia Conferencing Architecture

In this section, we present the architecture for multimedia conferencing on the Internet using the set of protocols that have been standardized by the IETF (*Internet Multimedia Conferencing Architecture*, IMCA) as described by Handley et al. in [Handley00]. Although the Internet Protocol has not explicitly been designed with support for real-time data transport, there have

been many efforts since the early 1990s to develop audio and video real-time conferencing applications and to use shared applications over the Internet.[2]

One important milestone was the introduction of *IP multicast* [Deering91] — extensions to the protocol requirements for IP hosts and routers and *multicast-routing protocols* that allow for efficient and scalable group communication on the Internet.

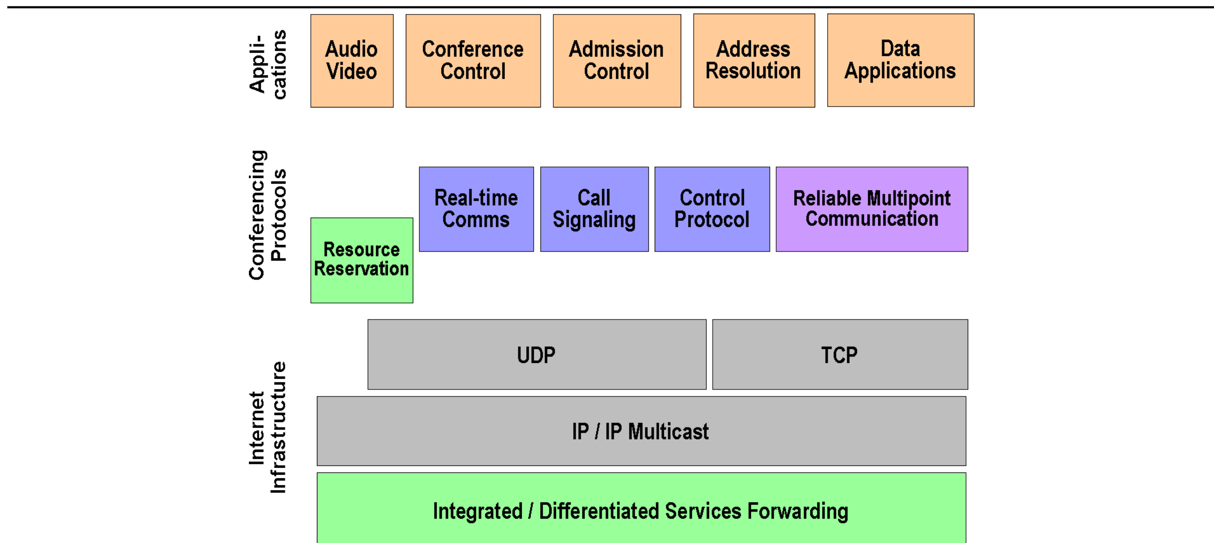

Figure 2.5: The Internet Real-time Multimedia Conferencing Architecture [Handley00]

Figure 2.5 provides an overview of the Internet Real-time Multimedia Conferencing Architecture. An important building block in this architecture is the *Real-Time Transport Protocol* (RTP, [RFC3550]), a transport protocol for media data of different types with real-time characteristics.

In addition to real-time multimedia communication, *shared applications* have been developed, including protocols for group communication that is more reliable than the best-effort IP multicast service can provide.

Regarding conference control, the Internet Multimedia Conferencing Architecture follows a significantly different model than the H.320 and H.323 recommendations — a model that is commonly known as loosely coupled conferencing. For conference establishment, the *Session Initiation Protocol* (SIP, [RFC3261]) provides services such as user location and user invitations. Details on IETF conference control protocols are provided in Section 2.2.1. Obviously security is an important requirement for any conferencing architecture. We will address this aspect directly in the relevant sections, for example SIP security in Section 2.2.1.

2.2.1 Conference Initiation

In this section, we describe the protocols of the Internet Multimedia Conferencing Architecture that are used to initiate conferences, i.e., protocols that are not used by applications such as

[2] In fact, initial work has been done as early as 1977, e.g., the Network Voice Protocol (NVP) for the Arpanet that is specified in RFC 741 [RFC741]. NVP has been implemented first in December 1973.

media engines or shared applications but that are used to control when and how conferences are conducted. We will look at conference setup first and describe the *Session Announcement Protocol* and the *Session Initiation Protocol*. Both protocols allow the use of different description languages for specifying parameters of the conference such as the configuration of the media sessions including transport parameters. Currently, the *Session Description Language* (SDP, [RFC2327]) is commonly used for both SAP and SIP.

2.2.1.1 Session Announcement

The *Session Announcement Protocol* (SAP, [RFC2974]) is used to disseminate information about multicast conferences to potential participants; it is mainly intended to be used for announcing public sessions such as broadcasts and public teleconferences that are scheduled for a given time and can thus be announced in advance. A user's SAP-client (a *session directory tool*) can be used to receive announcements from different sources and present them to the user. Announcements can either time-out (the announcer ceases the transmission) or be explicitly deleted by a corresponding SAP message.

SAP is based on IP multicast: An instance of a SAP session directory that wishes to announce a multicast conference, periodically multicasts packets containing a corresponding session description to a well-known multicast group. There is no restriction on who can announce sessions. SAP announcements are never acknowledged or in other ways answered by receivers: They are treated as *soft-state information* that is multicast periodically as long as the announcement is relevant.

SAP announcements can be sent with different multicast-scopes, and the scope of an announcement is usually aligned with the scope of the announced multicast sessions. This allows for keeping the announcement of local sessions within a local scope. (In Section 6.2.1.3, we discuss multicast scopes and corresponding protocols.)

Since announcements are sent periodically to a single multicast group (for a given scope) and since there is no feedback mechanism for indicating congestion, SAP must provide a rate control mechanism that keeps the bandwidth of all traffic in an announcement group to a well-defined limit (4 kbit/s if not specified otherwise). SAP announcers thus have to cooperate and choose appropriate repetition intervals for their announcements. Therefore, each announcer has to listen to other announcements in order to determine the total number of sessions. It uses this information and the size of its own announcement to calculate the transmission interval.

SAP is typically used by session directories such as sd (developed by LBL) and sdr (developed by UCL, the University College London). These tools are started on behalf of a user and are usually kept running permanently (especially when the user wants to announce a session). While running these tools, a receiver sees all session announcements and can thus perform another useful function: By processing the session descriptions of each announcement, a session directory tool automatically knows which multicast groups are used by currently running or scheduled conferences and can thus assist the user to allocate a multicast address that has not already been allocated by somebody else.

There are mechanisms for multicast address allocation in the Internet (the architecture for multicast address allocation is described in [RFC2908]), however these mechanisms are currently not widely deployed, which makes is necessary to rely on this SAP based approach in order to avoid address clashes. Jacobson has presented a scheme called *Informed Partitioned Random Multicast-Address Allocation Allocation* (IPRMA) in [Jacobson94] and Handley has

proposed an improved algorithm in [Handley97] that has been implemented in the session directory tool sdr. sdr allows a user to “create” new conferences and proposes multicast addresses for the media session that are not likely to be used by other conferences or applications, based on sdr’s knowledge about the current use of multicast addresses.

The *Session Description Protocol* (SDP, [RFC2327]) is normally used to describe the conferences that are announced via SAP announcements. SDP has been designed in order to allow for concise descriptions of the necessary parameters of multimedia conferences. This includes parameters for media sessions such as media type, media encoding, codec parameters and transport parameters such as multicast address, port number and used transport protocol. SDP also provides some support for describing meta-information about a conference such as information about the originator, the subject of a conference and scheduling information, i.e. at which time the conference takes place and, in case of recurring conferences, the recurring pattern.

2.2.1.2 Session Initiation

SAP is used for conferences that can be publicly announced in advance, e.g., for TV-broadcasts or public conference sessions. For the spontaneous initiation of conferences such as Internet Telephony phone calls, the *Session Initiation Protocol* (SIP, [RFC3261]) is used. SIP allows a user to *invite* other users into a conference and exchange the necessary conference parameters. In essence, SIP provides call signaling and call control services for spontaneously initiated conferences, and can thus be compared to H.225.0 and H.245, whereas the specific protocol mechanisms differ significantly. SIP provides the following functions:

- SIP uses application layer routing in order to deliver invitations to the appropriate participant. SIP provides the concept of SIP proxies — intermediary systems in the network that make routing decision and forward requests based on the specified address of the invitee (the SIP-URI), based on the result of interaction with *SIP location servers* (see below) and based on local policies.

- Users can register with *SIP registrars*, i.e., SIP servers maintaining registration databases that provide a mapping from a SIP-URI such as `sip:dku@example.com` to contact addresses that specify where the user is to be contacted. The information collected by registrars can be used by *SIP location servers* for informing clients of a user’s registered contact address and by SIP proxy servers to make call routing decisions.

- SIP provides *call signaling* services that allow a caller to establish a conference with a callee. Amongst other functions, the call signaling services provide support for transmitting call setup requests, for exchanging configurations, and for terminating calls.

- SIP provides reliable communication for the communication of *SIP user agents* and can be used with UDP, TCP and TLS (Transport Layer Security Protocol).

Figure 2.6 depicts the basic SIP call setup process. The basic call signaling in SIP can be compared to the H.225.0 procedures: a caller sends a call setup message (a SIP `INVITE` request) to a callee, and the callee sends call progress notification and a confirmation that the call has been established. Similar to the H.323 model, the main objective is to negotiate and to distribute a working conference configuration at the involved parties, which can be used to configure application entities. Unlike the H.323 model, SIP does not provide an explicit separation

into call signaling and call control. In essence, SIP focuses on call signaling, whereas the addition and removing of application sessions (that H.323 accomplishes through H.245's logical channel concept) can be achieved by updating the conference configuration. In [Schulzrinne98], Schulzrinne et al. provide a comparison of H.323 and an early version of SIP.

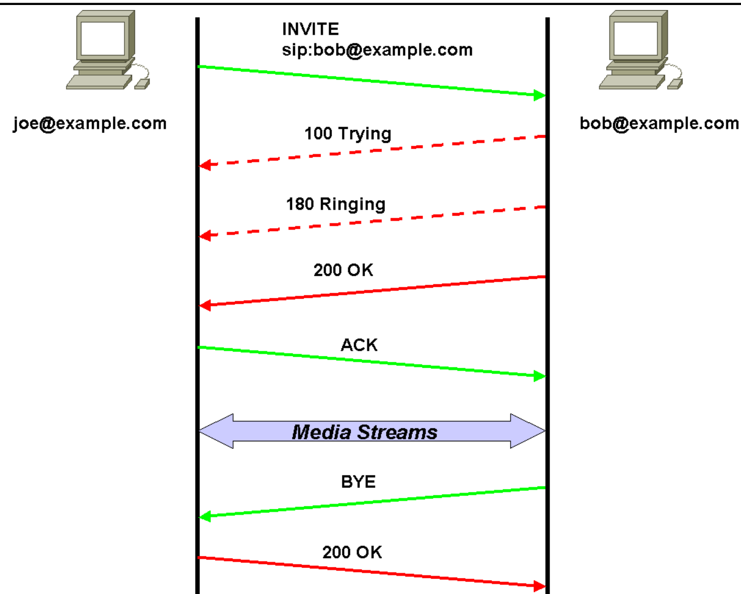

Figure 2.6: Basic SIP call setup process

For implementing this service, SIP relies on a request/response protocol, i.e., during a SIP session, a *SIP dialog*, two peers exchange requests and responses to requests. For example, SIP provide requests such as `INVITE`, `CANCEL` and `BYE` and responses such as `200 OK` (a positive acknowledgment) and `180 Trying` (a *provisional response* to an `INVITE` request).[3]

SIP messages (e.g., *INVITE requests*) can carry a payload that contains the session description for the conference that should be initiated. Today, SDP is used for that purpose. However, the requirements for publicly announcing multicast conferences differ significantly from the requirements for initiating a spontaneous conference, e.g., a phone call. For an announced multicast session, the announcer sends a fixed session description containing the set of media types, media encodings that should be used for the conference, including (typically) a single multicast address for each session. Receivers can check if they support the specified configuration and then join the conferences by starting media tools with the corresponding parameters. For spontaneously initiated sessions, it is important that both the inviting and the invited party agree on a commonly supported configuration — otherwise the conference cannot be initiated. Moreover, for two-party conferences, e.g., for Internet telephony calls, the media sessions typically do not use multicast but unicast communication. Consequently the transport address cannot be specified as a single multicast address that is used for both sending and receiving. Instead, each

[3]SIP uses the Internet Message Format [RFC2822] and mimics the HTTP request/response model by adopting some status messages and header fields. However, the requests (and obviously the semantics) are different from HTTP.

party has to know an endpoint address (an IP address and a port number) where the data, e.g., the RTP packets, should be sent to.

In order to accommodate the different requirements, the usage of SDP within SIP is explicitly specified in a companion standard [RFC3264] that defines how SDP is used in a two-stage exchange to gain a common view of two participants for a conference initiation. This *offer/answer model* specifies how SDP descriptions are exchanged over SIP and allows a minimal form of negotiation.

In summary, we can state that SIP provides similar call signaling services as H.225.0; whereas the call signaling communication semantics are largely identical, the communication mechanisms are not. In fact, we have developed mappings between SIP and H.225 for the development of gateway systems (described in Section 10.3, a detail description has been provided by Pollem in [Pollem00]) and Singh et al. have presented a similar approach in [Singh00]. Both SIP and H.225/H.245 can be routed through an application layer gateway fabric.

2.2.2 Conference Course Control

In the Internet Multimedia Conferencing Architecture, there is no equivalent of T.124 (see Section 2.1.3), and there is no agreed mechanism for providing conference course control, i.e., for providing services such as floor-control and maintaining membership-lists during the course of a conference. Instead, conferences rely on a *loosely coupled conferencing model*: Participants join an announced conference or are invited to a conference and then start the media tools for the respective media sessions of the conference. Although the transport protocols for media sessions, e.g., RTP or a multicast transport protocol, can provide some degree of membership control (participants can learn of the existence of other participants), there is no strict control. For example, in SAP announced conferences, participants can come and go at will — simply by joining and leaving the corresponding multicast group.

In the following, we describe protocols that are intended for *horizontal conference course control*, i.e., for coordinating the endpoints in a conference over wide-area communication links. Section 2.2.2.1 describes the *Agreement Protocol*, a generalized proposal for conference control that has been discussed in the IETF. Section 2.2.2.2 presents the *Conference Control Channel Protocol* that provides a scalable coordination mechanism for all user applications within a conference. The *Simple Conference Control Protocol* that we describe in Section 2.2.2.3 is a conference control protocol for *tightly coupled conferences*. Currently, proposals emerge that concentrate on providing multi-party conferencing with SIP [Rosenberg02] and there is also early work on minimal conference control functions using SIP, but we will not discuss this in detail in this thesis.

2.2.2.1 The Agreement Protocol

In [Schenker95], Schenker et al. describe protocol mechanisms for maintaining and updating ephemeral teleconferencing state for conference control. The term *ephemeral* means that the state that is managed by a corresponding protocol is only important for the conferencing session and is not persistent after the end of a session.

In this proposal, the authors describe certain dimensions of policies for the joint control of the ephemeral conference state: initiation (of state changes), voting (for anticipated state changes) and consistency. Different models of consistency are proposed. The general idea is

that every member has its own view of the distributed state and that the goal of the agreement protocol is to make these views eventually consistent.

Whereas the policies specify who can propose state changes and which degree of voting consensus must be reached to conduct state changes, the protocol framework provides different protocol mechanisms that could be used to implement these policies. The authors propose different mechanisms for different underlying communication services that can essentially be classified as reliable and unreliable communication.

- For reliable communication (that could be realized by a mesh of reliable point-to-point communication channels), the so-called *strong eventual consistency* (SEC, it can be guaranteed that after some time all members agree on a common view of the state) is achieved by deploying causal ordering (implemented by wall-clock timestamps) and by using a two-phase-commit procedure for conducting voting and applying state changes: State variables can be locked before a poll or an update message is sent by the initiator of a state change. The protocol mechanisms are designed to work with a transport service that provides reliable message transport with a bounded delay, i.e., the transport service does not try to provide absolute reliability at the cost of potentially unbounded message delay and undue complexity. Instead occasional violations of the consistency conditions are tolerable for the application of managing a shared ephemeral state that is not intended to persist beyond the session anyway. For example, in the case of network partitions or host failures, it would in general not be adequate to aim for absolute reliable transport and delay the delivery of a message indefinitely — instead the application should be informed of the problem rather soon.

- For unreliable underlying communication channels such as UDP over IP multicast, the consistency model is relaxed further: Instead of guaranteeing that the views of the state will eventually converge, the *weak eventual consistency* (WEC) model is used: the probability that the members have different views of the state decreases asymptotically but an eventual agreement cannot be guaranteed.

  In this model, the convergence is realized by a *periodically repeated transmission* of state variables. There are two variants of this basic procedure: In scenarios where every piece of state has a natural owner, this owner is responsible for retransmitting the current state. However, this model is only applicable when the overall state is separable and when the per-member state is rather small. Therefore, a second variant is proposed: The initiator of a state change is transmitting the current state of a certain variable repeatedly until he receives a message from another change initiator with a higher timestamp.

The concrete proposal of the agreement protocol has not been pursued further within the IETF's MMUSIC working group (where it has been discussed): as a framework it is too generic and abstracts too much from specific transport protocol issues. Moreover, transport technologies such as reliable multicast that could have been helpful for implementing and deploying the agreement protocol have not been available as early as it has been anticipated. In fact, reliable multicast is still (June 2003) being standardized within the IETF.

Nevertheless, the agreement protocol provides some interesting ideas: First of all, we can highlight the notion of an *ephemeral state* for the control information that is not intended to persist after the end of a session. This concept is useful as it helps to delimit the goal of managing a distributed conferencing state from other applications of distributed systems technology,

where absolute reliability and consistency are required. For maintaining a distributed conference state, we do not require all state changes to be consistent and durable; it is sufficient when there are mechanisms that will eventually provide a convergence of the distributed states. Instead of relying on an absolutely reliable transport protocol that makes very little assumptions on the characteristics of the underlying network and may thus be costly in operation, it can be more efficient to make the application aware of problems and allow it do deal with the problem appropriately.

The devised transport mechanisms to be used for unreliable underlying communication channels are also noteworthy: The agreement protocol shows that periodic retransmissions (with carefully chosen transmission rates) can be an adequate means to eventually realize consistency (although only probabilistic consistency) in a *scalable* way. The requirement of *scalability* is especially important when we think of larger groups with members that are on the wide-area Internet. Centralized solutions that can easily provide absolute consistency and global ordering are likely to fail due to their scalability issues. Instead, it can be more useful to rely on a solution that does not guarantee that all members will always have a consistent state, when it is actually sufficient that the state will eventually converge. The idea of periodic *soft state* transmission has become an important element in many protocols for distributed applications such as SRM [Floyd97] [Mccanne98].

Finally, the agreement protocol proposal provides a separation between the *mechanisms* and the *policies* that can be realized with the protocol. Thus the agreement protocol is not tied to specific assumptions on policies such as voting rules, access control and membership policy but merely provides the mechanisms that allow to implement different policy rules — a concept that we will also see in other proposals such as CCCP.

2.2.2.2 CCCP

The *Conference Control Channel Protocol* (CCCP, [Handley95]) is a conference course control protocol that provides mechanisms for application control (e.g., synchronization of applications in a conference), membership control (e.g., membership lists, access control) and floor management (e.g., managing floor control for specific applications). It addresses earlier noted requirements such as scalability (with respect to the number of members of a conferences and the geographic distribution of the members) and flexibility (the protocol should allow for different policies and applications). A conference is modeled as being composed of a number of geographically distributed people, using a variety of applications within the conference.

Figure 2.7 provides an overview of the CCCP control communication within a multimedia conference. The different applications exchange media data over their application specific media transport protocol (e.g., RTP). Each application and additional control entities for each conference participant attach to the conference control channel to exchange control messages. CCCP provides a messaging channel between all the applications of all users of a conference. Each application may be addressed independently. A CCCP address consists of three components:

Instance identifier: A unique identifier of an application on a host.

Type: The type of the application (or application component). Naming applications in CCCP is based on hierarchical typing of applications. For example, the major type of an application could be the media type that it is used for, such as `audio` or `video`. The secondary

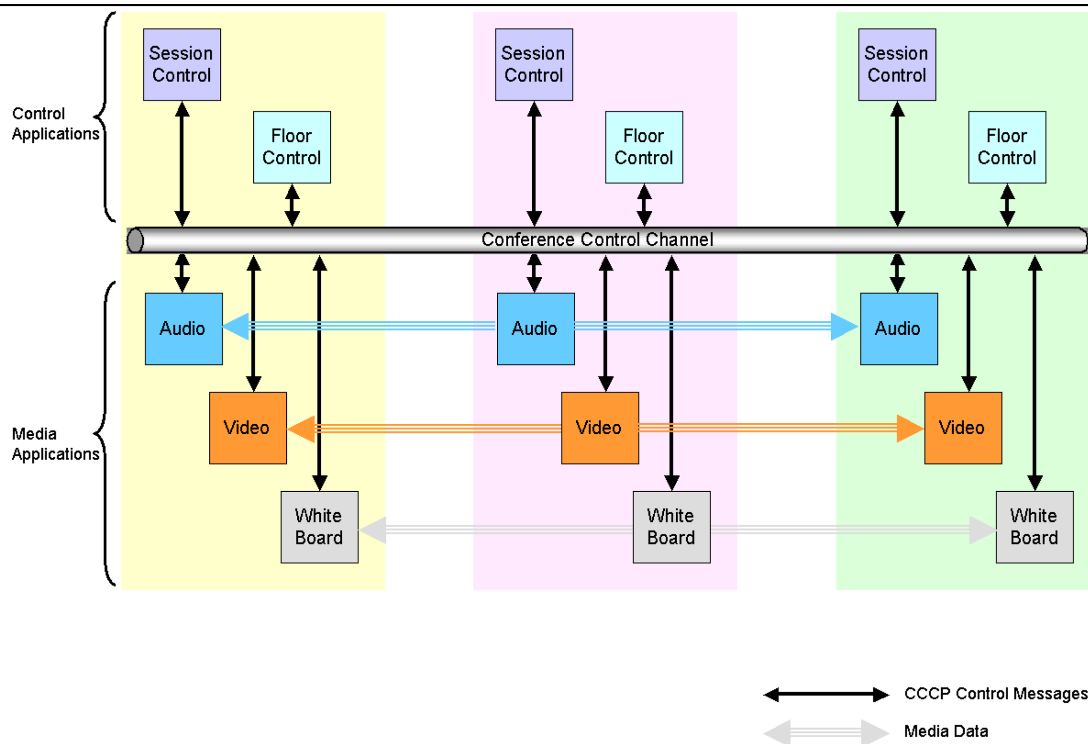

Figure 2.7: Conceptual overview of CCCP

type could signal whether we want to address a receiving (`recv`) or a sending (`send`) application. An example for a complete type would thus be `audio.send`. Due to the hierarchy concept, a message that is destined to an address with a type field of `audio` would also be delivered to an application with address type field `audio.send`.

User-name/host-name: The host-name that the application is running on (as a DNS name) or the user-name on whose behalf the application is running (as user@host-name).

Again, it is possible to use partly specified address elements: An address such as `*@host-name` will be delivered to all applications on the specified host.

Conference identifier: A conference identifier is intended to address applications that are participating in multiple conferences at a time.

An example for a CCCP address would be `(42, audio, dku@example.com, 1)`. CCCP supports the concept of *wildcarding* address elements in destination addresses: a message with a destination address of `(*,audio,*,*)` would be delivered to all applications with a major type field `audio` in their address.

CCCP provides different reliability and message ordering schemes that can be selected on a per-message basis. For example, it can be requested that a message that is sent to a group of receivers (using the wildcarding mechanism) has to be acknowledged by none, at least one, one out of n, or all potential receivers. Of course, the latter makes only sense if the sender knows the set of potential receivers. Since CCCP does not provide mechanisms to determine this set reliably, it allows this mode for messages that are destined to a fully qualified address only.

According to [Handley95], CCCP has been implemented as relying on IP multicast without using reliable multicast transport mechanisms.

For message ordering, CCCP provides the following classes: *no constraint* (deliver each received message immediately), *late discard* (discard any packets that are older than the latest seen), *adaptive playout* (place received messages in an adaptive playout buffer, similar to the way RTP implementations behave), *fixed playout* (similar to adaptive playout but with a fixed maximum delay), *adapative playout with ceiling* (a combination of the latter two) and *ARQ* (explicitly acknowledge every packet to synchronize sender and receiver).

CCCP messages have a plain text payload consisting of a function name followed by a space separated parameter list. The syntax and the semantics of parameters and function names are application specific.

CCCP provides confidentiality by supporting DES-encryption based on a shared key that has to be distributed out-of-band. Additional mechanisms for providing Message integrity and message authentication are not provided.

CCCP is an interesting proposal because of its application-oriented addressing scheme. The possibility of using wildcard addresses to send messages to a group of applications without having to consider transport addresses and details of group communication simplifies the integration of corresponding control interfaces to applications significantly.

Furthermore, the concept of choosing reliability and message ordering classes on a per-message basis is an appropriate way of dealing with diverse requirements in terms of reliability, consistency and timeliness of message delivery. By allowing the application to select the required service, CCCP does not impose a single paradigm and can thus support different classes of interactions.

One of the problems that have to be mentioned is the fundamental concept of providing a common communication channel for all applications and application components in a conference. In fact, CCCP can be viewed as combination of a horizontal and vertical control service. This limits the flexible re-use of these applications as components as it requires every application to implement CCCP and to understand the concept of conference course control. Moreover, it may lead to a significant form of overhead, if every application has to analyze and filter-out messages that are sent to a global IP multicast group (in [Handley95], Handley et al. mention the idea to automatically map application addresses to IP multicast addresses but it seems that this has not been pursued).

2.2.2.3 SCCP

In [Bormann01], we have described the *Simple Conference Control Protocol* (SCCP) that provides control services for *tightly coupled conferences* and has been inspired by the conference control services that are provided by H.323 (Section 2.1). The SCCP work can be viewed as an attempt to define a framework for tightly coupled conference control that considers the *Internet Multimedia Conferencing Architecture* and the related, existing control protocols, such as SIP [RFC3261] and SAP [RFC2974].

SCCP provides the notion of a *distributed conference state* that is established by the use of conference control protocols such as SIP and SAP and is later managed by SCCP. The SCCP services for managing the conference state can be grouped into the following functional areas:

- conference management;

- application session management; and
- floor control.

The SCCP services for *conference management* include the management of the conference membership (inviting and excluding members) and the overall conference course control. Merging and splitting of conferences is not included. In SCCP, there is the notion of applications as functional components within a conference (e.g., interactive voice communication), and the communication for operating these applications takes place in application sessions (e.g., an RTP session for voice communication). The SCCP *application management* functions provide mechanisms for coordinating application sessions, e.g., creating application sessions, terminating application sessions and membership management for application sessions within a conference.

Finally, SCCP provides a *floor control* concept for supporting application state synchronization, which includes operations such as requesting the floor and releasing the floor. SCCP assumes a reliable group communication transport that provides reliable and consistent delivery of messages and that provides a total ordering of messages.

The work on SCCP is not yet completed and the current specification does not provide a mapping to a specific transport protocol (although MTP or one of its successors have been anticipated). From the current specification, we can conclude that SCCP is strictly a *horizontal* conference control protocol that is intended to fit into the Internet Multimedia Conferencing Architecture and that does not address the local coordination of applications, i.e., components of conferencing endpoints.

2.2.3 Summary

In the previous sections, we have presented the Internet Multimedia Conferencing Architecture with a special focus on control services and on group communication. One important characteristic is that many if not all protocols are designed to work well in both unicast and multicast scenarios. In fact, the point-to-point usage of protocols such as RTP and SRM is just a specialization of the general multicast case, which they have been designed for.

We can state that the very properties of IP multicast have had an influence of the design of the transport protocols and applications: The *Light-Weight Session model* and the *Application Layer Framing paradigm* can be considered as the opposite of the strictly layered H.323 model of the ITU-T. The service model, including the unreliable best-effort service, of IP multicast has promoted a soft-state driven approach emphasizing scalability, robustness and anticipation of heterogeneity.

The loosely coupled conferencing model has its origins in public IP multicast sessions on the Mbone where participants are free to join and leave at will, where conference course control is rarely used and no regulations on the usage of specific tools are made because receivers can select application sessions they want to join, considering a user's preferences and the capabilities of the conferencing endpoint. As a result, there is no accepted conference-course control protocol (except for the T.120 family of protocols). However, the increasing interest in and work on solutions for Internet telephony style applications (based on SIP) has recently re-fueled the interest in conference course control, and solutions that work well with SIP or are based on SIP might emerge in the future.

Currently, conference control services are mainly provided for conference initiation, i.e., call signaling and call control through SIP, which has emerged from a comparatively simple approach for the spontaneous initiation of Internet multimedia conferences to a call signaling and call control protocol that provides similar services as H.225.0 and H.245 of the H.323 domain.

2.3 Lessons Learned

In the previous sections, we have presented technologies for multimedia conferencing with a special focus on conference control. Table 2.1 compares the two architectures with respect to services and their realization.

Table 2.1: Services and protocol mechanisms for H.323 and IMCA

Service	H.323 Protocol	IMCA Protocol
Real-time Media Transport	RTP	RTP
Call Signaling/Call Control	H.225.0 and H.245	SIP and SDP
Shared Applications	T.120	SRM, NTE, MTP (application specific)
Conference Control	GCC (T.124)	no acknowledged protocol, SIP-based approaches emerging

For multimedia conferencing on the Internet, both architectures rely on RTP/UDP for the real-time transport of media data. H.323 and SIP-based conferencing systems both provide a concept of call signaling and call control services for the initiation and basic control of conferences. When comparing the fundamental model and the service set of H.225.0/H.245 and SIP, we can state that both provide the following main services:

- call signaling for establishing a call control context;
- negotiating and distributing appropriate conference configurations;
- enabling basic control functions such as updating configurations and terminating conferences; and
- user registration, user location and call routing via intermediate systems.

An analysis of the call signaling protocols has shown that both protocols provide similar procedures and message types, although the communication is implemented by different mechanisms. In Section 2.2.3, we have noted that existing gateway solutions demonstrate that a mapping between individual protocol messages is possible.

We have analyzed the conference control architecture for H.323-based multimedia conferencing, i.e., the T.120 Generic Conference Control (GCC) approach. The T.120 model relies on a *tightly coupled conference model*, i.e., a strict management of conference membership that is

implemented by a set of corresponding conference control functions. GCC as well as conference applications are layered on top of the *Multipoint Communications Service* (MCS), which is a transport-protocol independent group communication mechanism and is implemented by a set of point-to-point connections between the MCS providers. We have also noted the scalability concerns with this approach.

The Internet Multimedia Conferencing Architecture has evolved from a set of building blocks for real-time media transport, conference initiation and conference announcement. Group communication by the use of IP multicast has been taken into account for the design of the different protocols right from the beginning. Rhe service model and the properties of IP and IP multicast have led to the formulation of an alternative design for real-time and group communication protocols: the concepts of *light-weight sessions* (LWS, [Mccanne98]) and *application layer framing* (ALF, [Clark90]) deliberately deviate from a strict layer approach, such as the OSI model. Unlike the MCS-based protocols of the H.323 family, ALF-based protocols do not rely on the assumption that the underlying transport protocol provides a reliable, totally-ordered service. Instead, the heterogeneity (of sub-network-links and end-systems) and the loosely-coupled nature of IP multicast sessions is taken into account.

This principle influences the design of transport protocols such as RTP and SRM [Floyd97], [Mccanne98] and the design of applications such as NTE [Handley97]: the transport protocol is essentially a rather slim layer above the underlying multicast/unicast datagram service layer. The NTE transport protocol delivers *application data units* to the application in order to let the application decide how to process them, e.g., how to deal with messages that are delivered out-of-order and how to deal with message loss.

One notable mechanism for achieving reliability in group communication sessions without the need for explicit acknowledgments or retransmission requests is the *soft state approach* that relies on periodic multicasts of information, e.g., ADUs in a shared application protocol session. Soft state communication can be an efficient way to finally reach consistency in a group communication session, where an eventual state convergence can be tolerated. SRM is a typical example of an application-independent transport protocol of this type.

For those applications that have stronger requirements in terms of reliability guarantees and message ordering, *reliable multicast protocols* have been designed that try to combine the efficiency and scalability of native multicast with the service of "traditional" transport protocols. We have presented MTP and its successors as representatives of such protocols.

We have noted that there is no acknowledged horizontal conference control for the Internet Multimedia Conferencing Architecture, although some approaches (with different scope and ambitions) have been proposed. The *Agreement Protocol* that we have presented in Section 2.2.2.1 was a first attempt to define a conference control protocol that fits the service model of IP multicast and considers scalability and efficiency requirements. The key concept of the Agreement Protocol is the notion of treating conference control state as *ephemeral* information and by considering different consistency models, *strong eventual consistency* and *weak eventual consistency*, depending on requirements of applications.

SCCP, which we have described in Section 2.2.2.3, is an example for another type of conference control protocol that provides GCC-like conference control services and is designed to be used with a multicast transport protocol such as MTP/SO. Both the Agreement Protocol and SCCP are strictly targeted at *horizontal conference control*, i.e., they do not address the coordination of entities within a single conferencing endpoint. The *Conference Control Channel*

Protocol that we have described in Section 2.2.2.2 is a more general approach that addresses both horizontal and vertical conference control. The interesting aspects of CCCP are the concept of providing different reliability and message ordering properties on a per message basis and the flexible addressing concept that allows for addressing entities by using *wildcard addresses*. The control messages can thus be destined to specific applications components, depending on their application type or depending on their conference or user affiliation. We have noted that the coupling of vertical and horizontal control can impede the deployment of the protocol, as it requires every application component to implement CCCP (and its horizontal control features).

Both H.323 and SIP-based conferencing are not tied to specific applications. H.245 and SDP can both be used to express audio, video and shared application configurations, and, for T.120 conferences, GCC provides the concept of application independent capability description.

With these observations in mind, we can state that (especially for telephony-like applications) both H.323 and SIP-based conferencing provide similar functions and a similar overall architecture: a clear separation of application sessions and control functions such as conference initiation. For both approaches, there are application entities that provide the capturing, sharing and rendering of information and there are control entities that provide conference initiation, call control and (if applicable) conference course control.

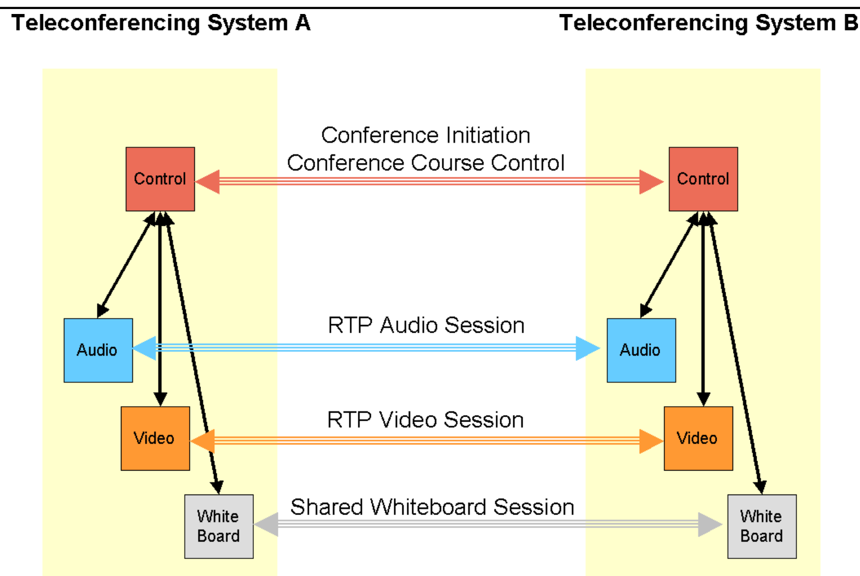

Figure 2.8: Generalized local endpoint architecture

Generalizing the architectures of H.323 and SIP-based conferencing, we can identify similar protocol services and similar structures of endpoints that implement the corresponding protocols. Figure 2.8 depicts a generalized endpoint architecture, where a endpoint controller coordinates a set of application protocol entities. For both H.323 and SIP-based conferences, the endpoint controller would perform the call initiation, i.e., either through H.225.0/H.245 or through SIP, and negotiate an appropriate conference configuration based on the capabilities of its own and the remote application entities. After the conference has been established, the configuration is distributed to the local application protocol entities and the application sessions can start. During a conference, the controller can implement conference course control

(if applicable) and implement additional call control services such as eventually terminating the conference.

Neither H.323 nor SIP-based conferences specify a local coordination service. Instead, there are different, endpoint application specific solutions. For example, in some applications such as Microsoft Netmeeting, a conferencing endpoint is an integrated application, without an exposed modular structure and thus without coordination. For some SIP-based applications, the coordination is limited to the *starting* and parameterizing of application entities (after a conference has been established).

In Chapter 3, *Use Cases and Requirements*, we will investigate a more detailed scenario for local coordination within conferencing endpoints and in Chapter 4, *Foundations and Related Work*, we will also describe existing approaches that provide some form of local coordination service.

Chapter 3

Use Cases and Requirements

In Chapter 2, *Conferencing Architectures*, we have discussed conferencing architectures with a focus on control services. We have introduced a generalized local endpoint architecture and have noted the lack of standardized local coordination mechanisms for conferencing systems. In this chapter, we will describe two use cases for local coordination: the design of decomposed multimedia conferencing systems, and, as an example for local coordination in a completely different application domain, the local coordination of devices and application components in in-vehicle networks. The discussion of these use cases yields a list of requirements that we present in Section 3.2. These requirements are later matched against existing solutions and related work in Chapter 4, *Foundations and Related Work* and are considered for the design of our coordination architecture that is described in Chapter 5, *Architecture*.

3.1 Use Cases

In Section 3.1.1, we discuss a possible architecture and resulting requirements for the design and implementation of *decomposed multimedia conferencing systems*. In Section 3.1.2 we analyze the state-of-the art and anticipated future developments for *in-vehicle networking* and sketch a corresponding coordination infrastructure. Subsequently we discuss the requirements for both use cases in detail in Section 3.2, with a focus on message-oriented coordination. Each use case is discussed focusing on the *system architecture*, because the objective is to derive requirements for network protocols and possible network architectures.

3.1.1 A Decomposed Multimedia Conferencing System

In Section 1.3.1, we have presented a rough sketch of a decomposed multimedia conferencing system that consists of different modules that are linked using a common communication channel. In this section we will look closer at the requirements for such a decomposition model, focusing on the communication aspects.[1]

[1] It should be noted that we are describing a sample architecture, i.e., other architectures are possible. However, we have been able to validate this model by investigating real-world telephony and conferencing systems in different research projects that we describe in Chapter 10, *Mbus in Projects*.

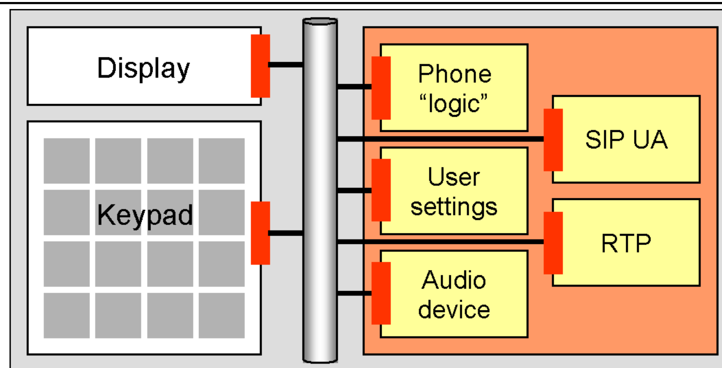

Figure 3.1: Modular conferencing endpoint

Figure 3.1 depicts an architecture that is quite typical for modern IP telephony systems. Different modules (possibly from different vendors) are integrated into an endpoint system relying on a specific component framework or inter-process communication mechanism. We can identify the following components:

Audio Device

Obviously a telephone must provide some kind of audio hardware that is utilized by the system using a software layer for abstraction and hardware control (a *device driver*). This module requires the following interactions:

- The device driver may be queried for its capabilities with respect to encodings, sampling rate etc.
- The device driver needs to be configured and initialized.
- The device driver can be dynamically configured, e.g., setting the volume.
- Audio data can be read from and written to the device.
- The device driver can report events, e.g., when the audio data is available or when the hardware is ready for the output of additional audio info.

RTP Module

An RTP module provides an RTP protocol stack and typically functions such as packetization, de-jittering and the required RTP functions for sending RTP and RTCP packets. This module requires the following interactions:

- It needs to be configured with respect to transport parameters (IP addresses and ports to send to and to receive from), packetization format, payload type and RTCP parameters (source description information).
- It can generate events such the appearance of new members in a session, and a controlling modules needs to be notified of these events.

- It can be queried for protocol stack statistics, e.g., for receiver statistics based on the received RTCP reports; and it can deliver and send media data.
- In many cases, an RTP module will be logically and physically co-located with an audio module for efficiency reasons, i.e., in order to avoid unnecessary copying of media data.
- In a multimedia conferencing system that provides video as an additional media type (and possible other forms of cooperation), there will be multiple *media engines*, where each media engine provides the media transport (using RTP) and the media-specific capture and render functionality. Concerning the control facilities for RTP, these media engines will be quite similar.
- For some applications, it can be required to allow for coordination *between* multiple media engines. For example, in [McCanne95], McCanne has described the coordination of audio and video engines for achieving *lip synchronization*, i.e., the simultaneous rendering of audio and video streams that pertain to a single source, e.g., the current speaker in a conference.

Signaling Protocol Engine (SIP UA)

A signaling protocol engine in an IP telephone, a SIP user agent in this example, operates an instance of a call signaling/call control protocol stack, e.g., an implementation of SIP (as depicted in Figure 3.1) or H.323. This module requires the following interactions:

- the call signaling stack has to be configured with respect to the IP configuration, user information and call signaling specific parameters such as proxy and registrar addresses (in the case of SIP);
- in order to set up calls, control and terminate them, a controlling module communicates with the signaling stack relying on a procedure-call scheme, e.g., by sending requests such `set-up-call` or `terminate-call`; and
- the call-signaling engine generate events such as `incoming-call` or `call-terminated` that have to be reported to a controlling module.

User Interface

A phone's user interface is not radically different from other program user interfaces: it provides a display unit (and possibly audio output for feedback) and an input device that can consist of multiple buttons, but can also be implemented as a touch-screen. Some phones also provide scroll wheels and voice control. In general, a user interface has the following communication requirements:

The display is controlled, possibly relying on a procedural scheme, i.e., a controller relies on a simple API providing procedures for changing the display state; and the input device can generate events, e.g., key presses. It should be noted that the level of abstraction on that a user interface operates may differ, e.g., input events do not necessarily have to be key-presses but can also represent higher level semantics, e.g., `the user has invoked function xy`.

Phone Controller/Coordinator

In this modular system we will also find a coordinating entity that orchestrates the interaction of the multiple modules, i.e., a controller that maintains the overall state of the phone, interprets user input events, initiates new calls using the call signaling module etc.

The coordinating entity can be viewed as a *client* of the other modules. For example, a manufacturer of a phone could deploy third-party products for call signaling, audio transport and rendering and GUI modules from different vendors and add the coordinating entity as the "glue" that links the third-party modules together.

Summarizing we can state that a component-based approach for designing conferencing systems essentially relies on the following assumptions and requirements for the component and communication technology that is to be employed:

- It is required to be able to issue commands with procedure call semantics, i.e., calling a procedure and retrieving a result.
- For these interactions, the communication between modules must be reliable.
- In addition, for a request/response interaction, we require at least a FIFO ordering of requests that are sent from a caller to another entity.
- It is required to be able to disseminate event notifications, possible after an explicit subscription by the client that wishes to receive them. For example, a controller may register for RTP event notifications of a certain type.

When analyzing existing multi-media conferencing systems, we can often observe a conceptual and physical separation of modules. The specific component technology and communication protocol that is used to implement the composition can differ significantly and is of course dependent on the requirements of the corresponding application. In essence, we have come across the following approaches:

The monolithic approach: It is possible to have a conceptual separation of functionality but to integrate all the required functions into a single system, e.g., into a single program. It is obvious that in this case we cannot speak of a component-based approach, since there are no modules of independent deployment and no third-party composition.

The software library approach: A system that is composed of software libraries provides a certain degree of independence between modules, e.g., they can be deployed and tested separately and be re-used in other contexts. However this approach requires all the modules to be linked into a common program or run in a common environment, which can be restrictive and limits the possibility of third-party composition and re-use of modules.

Nevertheless the approach may be completely viable for implementing systems where re-use and dynamic adaptation is not a key requirement. Many specialized solutions, e.g., very simple and light-weight systems, are implemented using software libraries as a sole method of re-use and separation. The interfacing between library code is the same as used for the monolithic approach: function and procedure calls.

The distributed approach: Instead of integrating modules into a common program space, some systems go a step further in decoupling modules and rely on separated programs that are coordinated using some kind of inter-process communication mechanism. For example, UNIX pipes can be used to connect two programs, message queues are a message-oriented coordination mechanism that allows more than two entities to communicate on a local system. Truly distributed solutions where programs can run on different hosts are less common.

Compared to the monolithic approach and to the software library approach, the decoupling of modules as separate programs increases the flexibility for the client who wants to integrate different subsystems from different vendors into a combined system. The individual programs can be implemented in different programming languages, and there is no need to align calling conventions for function calls. Of course, there must be an agreement of communication mechanisms to use, and the semantics of messages (for a message-oriented solution) must be well-defined.

Usually this approach is also associated with a higher degree of parallelism than, e.g., the library approach, because the separation of functionality and the decoupling of modules is an important prerequisite for multi-threading or multi-tasking implementations. In order to take advantage of the potential parallelism, appropriate communication mechanism, i.e., asynchronous communication, must be employed.

Note that we have not compared the approaches with respect to whether they are component-based or not. In fact, a software library approach can be as component-based as a distributed approach because components can be implemented as code libraries. It is conceivable that a component exists within a library, is an independent unit of deployment and provides typical component features such as adaptability. In Section 4.4, we will analyze the nature of component-based systems in more detail.

Nevertheless, it is worth noting that pure component-based approaches are rather rare for implementing multimedia conferencing systems. Certainly one reason is that many of these systems are usually not implemented on mainstream platforms, where newer techniques are more common. There are some notable exceptions, e.g., there is an RTP COM component for Microsoft Windows platforms that can be used as a component by applications.[2]

When we revisit the scenario of integrating a conferencing system into a user's computing environment that we have mentioned in Section 1.3, we can conceive different ways a conferencing system such as the IP telephone depicted in Figure 3.2 may interact with its environment:

- the conferencing system may be *extended* by external application entities such as the video entity or the shared application entity;
- additional control components outside the phone can take over the control of the phone, e.g., by remote-controlling it or by performing additional control functions during a conference, e.g., managing floor control; and
- events that are generated by the phone and by the interaction of the user with the phone can be distributed to a user's set of personal devices, e.g., to personal presence management applications.

[2]The Vovida WinRTP component (`http://www.vovida.org/`) is an example for a COM component.

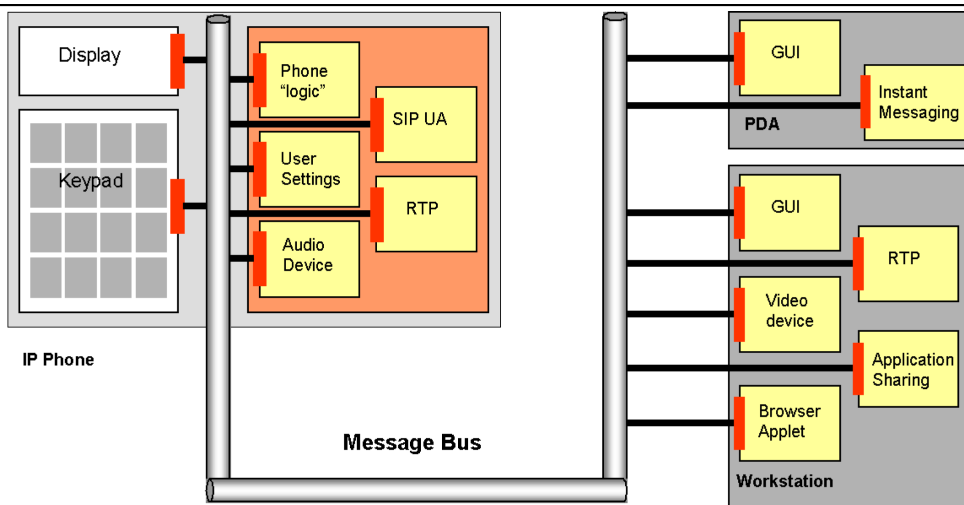

Figure 3.2: Inter-system communication

Fundamentally, the coordination of external application entities involves the same functions as the coordination of internal application entities: the application entity must be queried for its capabilities, which must be considered for the call setup, i.e., in the SIP offer/answer process; after a call has been established, the external device must be provided with an appropriate configuration, and during a call, further control messages may be sent to the application — just as for coordinating internal application entities.

Coordination between application entities should also be possible, just as for the local coordination case. In fact, for all coordination tasks, the communication with external entities should not constitute an exceptional case. Instead, the same message-oriented coordination service should be employed (from an application program's perspective).

Distributed coordination of application entities has to be *secure*, e.g., only authorized entities should be able to take over the control of a conferencing system, and crucial control information, e.g., user passwords that are conveyed in a certain control messages, must not be disclosed. In Section 3.2.10, we discuss security issues for coordination in more detail.

For *inter-system* coordination, we can name the following additional requirements:

- the components must be linked by a communication mechanism that can be used for inter- and intra-host communication;
- this communication mechanism must provide the same service with respect to reliability and consistency as we have stated above for the local decomposition;
- additionally, the communication mechanism must provide security against corruption by third-party attackers and (possibly) eavesdropping;
- the communication mechanisms must support an easy composition of modules, i.e., without manually configuring peer addresses and complicated procedures to set up a communication session;

- the components of a running system, e.g., the IP telephone, should be able to detect by themselves when new components such as a video engine become available, and consequently it should also be possible to detect when components are no longer available;
- the same types of interaction schemes (procedural and event-notification-based) must be provided; and
- since the dissemination of events and soft-state updates are potentially interesting for more than one component, the communication protocol should provide a group communication mechanism that allows to address messages to a group of components.

In Section 3.2, we aggregate the requirements for this use case and for the in-vehicle networking scenario and discuss the common important requirements in more detail.

3.1.2 In-Vehicle Networks

One of the recent major advances in automotive technologies is the introduction of computer networks in vehicles. In [Leen02], Leen and Heffernan report on estimates that more than 80 percent of all automotive innovation now stems from electronics. The introduction of computer networks into cars has originally been motivated by the need to reduce the amount of wiring required to implement electric control functions such as electric windows controls, central locking and electronic motor management. This was not only motivated by the need to reduce the overall vehicle-weight but also by the desire to reduce the complexity of the wiring harness in order to be able to develop more manageable and more *expandable* systems. Local area networks were hence introduced in order to operate control communication for multiple devices over one single network, thus removing the need for directly connecting every two devices that have a control relationship. This initial motivation is reflected by the term *multiplexing* that is often used to refer the deployment of networking technology in vehicles: a single medium is multiplexed, i.e., used with multiple devices.

After this technology has been pioneered in the mid-1990s in premium class cars such as the Citroën XM and the Mercedes Benz S-class series, we can now (2003) identify an adoption of computer network technology for mainstream and even small cars, such as the BMW Mini and the Citroën C3. In these cars, many switches are no longer power-switches (or connected to relays that perform the power-switching) but have become nodes on the car's network that generate a signal when being operated.

3.1.2.1 From Cable Replacement to New Applications

Besides reducing the complexity of the wiring, another motivation has emerged: introducing new features that become possible due to the existence of a ubiquitous intra-car communication network. For example, a rain sensor in the windshield can generate information that is not only used to control the wipers but is at the same times used to trigger the automatic closing of windows and the sunroof. Features like this have become commonplace for cars that provide "multiplexing" technology.

A closer look at specific installations reveals that typical cars do not employ a single but multiple networks that may be interconnected by gateways. The motivation for this is that there are different functional component groups with different requirements with respect to the

characteristics of the network technology. In addition, some essential and sensitive functions such as motor management and ABS control are kept separate from the rest of the system for safety reasons.

Figure 3.3 depicts an in-vehicle network in a contemporary mid-range car. The car provides the following different network segments:

- The so-called *Mechanical CAN* is the network segment for the crucial components that provide the motor management, the computer-controlled suspension, the automatic gearbox, and the ABS and ESP control. The acronym CAN refers to the term *Controller Area Network*, a serial bus system for real-time communication in device control applications. CAN is currently used by the majority of car manufacturers for real-time device control.

- Two *Body VAN* segments are used to connect controls and functional components that are not directly related to the crucial core functions. The acronym VAN refers to the term *Vehicle Area Network*, a low-cost variant of CAN that provides lower bit-rates and is intended for functions with reduced real-time communication requirements.

- The *Comfort VAN* connects controls and devices for comfort functions such as the car radio, the navigation unit, the instrument panel, the central car display and the air conditioning.

These network segments are connected through a central gateway called the *Built-in Systems Interface* (BSI). The BSI can forward messages from one segment to another (where relevant). For example, the value for the current average fuel consumption could be generated by the motor management and forwarded through the BSI to the Comfort VAN, where it is displayed on the instrument panel.

CAN and VAN are broadcast buses that deploy *Carrier Sense Multiple Access/Collision Resolution* (CSMA/CR) for media access. Nodes do not provide a unique network address; instead CAN/VAN relies on unique message numbers for every type of message. The message number is used for implementing a priority scheme that is used for media arbitration: A message with a higher priority "wins" the media arbitration competition in case two nodes send a message at the same time. The message number is also used to implement *receiver-based filtering*: The message is broadcast to all nodes, and the nodes decide by the message number whether to process the message or not. The maximum payload length of a CAN message is 8 bytes.

3.1.2.2 Real-time In-Vehicle Communication

CAN has been explicitly developed by Bosch in the 1980s as a low-cost communication mechanism for electronic vehicle components and is still being used for motor-management and similar functions. However is does not fulfill the requirements of extreme safety-sensitive applications that are currently introduced in production cars: So-called *X-by-wire* technologies aim to replace the mechanical (or hydraulic) coupling of crucial control instruments from the corresponding components by computers and computer network links. For example, the steering wheel will be decoupled from the wheels, and a computer that uses the steering wheel as an input device will control the power steering. The corresponding network technology must fulfill real-time requirements that CAN and other CSMA/CR based technologies cannot meet.

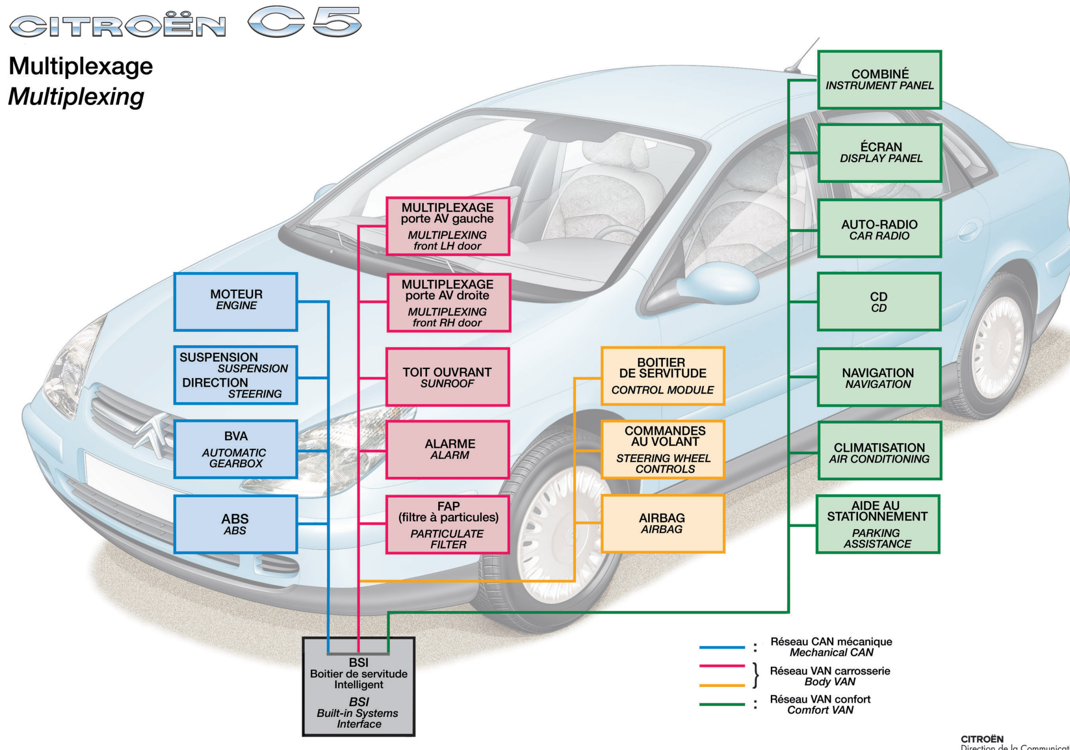

Figure 3.3: A sample in-vehicle network

CAN communication is *event-triggered*, i.e., any node can try to send a message at its own discretion. In case of collisions, the collision resolution algorithm will eventually lead to a transmission of both messages (this property is called *non-destructive resolution*), however without guaranteeing real-time properties. For controlling the brake or the car's wheels this is unacceptable, especially from a legal point of view: if the real-time characteristics of a message transmission cannot be guaranteed, i.e., formally verified, the possibility that a car crash has been caused by a malfunction, respectively by a delayed message due to congestion, can never be excluded, which in turn means that a driver could never be held liable.

Therefore, other protocols with guaranteed real-time properties are being developed and standardized for the use in *X-by-wire* applications. These protocol are called *time-triggered protocols*, and the two main approaches are called *Time Triggered Protocol* (TTP[3]) and *FlexRay*.[4]

The transmission is not event-triggered (i.e., the message is not sent upon the detection of a new value at the sensor) but *time-triggered*, because it is sent periodically, based on a static schedule, regardless whether the value has changed with respect to the last message.

[3] http://www.ttech.com/

[4] http://www.flexray.com/

3.1.2.3 Multimedia In-Vehicle Communication

In addition to controlling driving and comfort functions, network technology is also being employed for the transmission of multi-media streams. For example, the *Media-oriented systems transport* (MOST, [MOST02]) technology that has been developed by a consortium of car and multimedia equipment manufacturers[5] is intended to provide a framework for connecting entertainment equipment, personal devices of a user and microphones and speakers with another. MOST is a multimedia fiber-optic network that provides synchronous transmission for multimedia data and asynchronous transmission for "packet-data", e.g., encapsulated IP packets. The main idea of MOST is to provide a complete framework that allows for the cost-effective connection of simple digital devices such as speakers and microphone through the synchronous transmission mode *and* the connection of more capable systems that can implement control functions and connect to IP-enabled hosts through the use of gateways.

While the idea of building yet another vertically integrated protocol framework that largely duplicates functions of existing protocols is questionable, MOST provides some noteworthy ideas on the application layer. On the application level, each MOST device contains multiple components called *function blocks*. Examples for function blocks are amplifiers, MP3 players and tuners. Each function block again provides a well-defined set of functions. For example, the MP3 player function block could provide the functions `play`, `stop` and `skip`. Each function has a well-defined interface, which has to be known by calling entities. MOST distinguishes three types of function blocks: *slaves* are function blocks that always require a dedicated controller, *Human Machine Interface* blocks (HMIs) are function blocks that have an interface to the user, and *controllers* are functions blocks that use functions in other functions blocks.

MOST deploys a location-independent addressing scheme. A function is addressed as a component of a function block and each function block in a network provides an identifier. This works by creating a static registry of function block identifiers and by assigning each identifier a specific set of semantics. For example, the function block identifier `0x22` designates an audio amplifier. In case, there are several function blocks of the same type in one network, they can be uniquely addressed by the use of *instance identifiers* that may be explicitly specified in destination addresses. MOST provides a classification of functions into different categories:

Methods: Methods can be viewed as RPCs (see Section 4.1.1) for MOST. They are used to invoke remote procedures, they can be used with parameters and they can return results.

Properties: Properties can be inspected and changed through standardized interfaces. They can be compared to properties in a component framework such as JavaBeans (see Section 4.4.2).

Events: Events are used for *publish/subscribe* communication. Controllers can register for certain events at a function block and will subsequently be notified about changes of the corresponding value.

The list of supported functions, their names and types are specified in a *function interface*, an interface definition that has to be known to the developer of a component that intends to use

[5]The MOST Cooperation (`http://www.mostnet.de/`) has been founded by BMW, DaimlerChrysler, Harmann/Becker and OASIS SiliconSystems.

other functions. However, MOST also provides support for *dynamic interfaces* whose characteristics can be obtained through MOST communication mechanisms at run-time.

MOST is a typical example of a technology that has been designed *before* Internet protocols and Internet technology based appliances have become a ubiquitous commodity. Other protocols that provide a similar duplication of functionality are HAVi (*Home Audio Video Interoperability*[6]) and Bluetooth[7] that both also provide a vertically re-engineered stack of network protocols with similar functions as available Internet technology. While the definition of a proprietary stack of network protocols is not necessarily a problem for closed systems, it is clearly an obstacle for achieving interoperability of systems from different domains (which we discuss in Section 3.1.2.4), e.g., when we think of extending the car network with portable commodity systems such as laptop computers, PDAs and MP3 players.

In addition, the application layer semantics of MOST are fixed to car audio/video applications (and related applications), whereas other approaches that we will discuss in Chapter 4, *Foundations and Related Work* provide much more general frameworks. However, it is interesting to see how the car industry addresses the needs of component and service interoperability and the requirement for integrating distributed components into applications by the use of networking technology.

3.1.2.4 In-Vehicle Communication in the Future

The current state of in-vehicle networking can be summarized as follows: For fault-sensitive functions CSMA/CR-based network technologies such as CAN are used and for networking comfort and multimedia components, there is a proliferation of proprietary or yet-to-be-standardized technologies. In [EASTEEA02], the EAST-EEA project described an expected scenario for the development of in-vehicle electronics. One of the anticipated main trends in this scenario description is the

> *Improvement of the seamless connectivity between on-vehicle and off-vehicle devices.*
>
> —[EASTEEA02]

This refers to the integration of the in-vehicle network with user-devices, e.g., a portable MP3-player or a PDA. To allow users to integrate their personal devices into an existing in-vehicle network one major requirement obviously is a commonly accepted network architecture and the existence of standardized protocols and application semantics.

The *Automotive Multimedia Interface Collaboration* (AMI-C) is a consortium of car manufacturers that is addressing this issue by standardizing interfaces for "mobile information and entertainment systems". The AMI-C approach is similar to the MOST concept but tries to abstract from specific network transport mechanisms. Essentially, the in-vehicle network is viewed as a set of components that advertise their services on the network. For example a network component could be a digital television receiver that implements multiple individual

[6] `http://www.havi.org/`

[7] `http://www/bluetooth.org/`

functions including tuner, decoder and audio amplifier. Some of these functions can advertise their existence as *functional modules* and be used by other modules.

In principle, the usage models and requirements for these expandable in-vehicle networks provide some similarities to the desk area computer environment of office users. In the following, we will first present the specific architecture and selected protocols (both are still under development), before we summarize the main requirements and propose an alternative, Internet-technology based design in Section 3.1.2.6.

In the AMI-C architecture, link layer technology and network/transport protocol independence is realized by connecting different networks (that are anticipated to run different service discovery and session protocols) by application layer gateways. For this purpose, the AMI-C specification [AMIC03] employs the *Open Services Gateway Initiative* (OSGi) specification [OSGI03].

OSGi is an architecture for the deployment of network services that can be provided by different network and service discovery architectures. OSGi has originally been developed for home networking, an application where (similar to the vehicle networking area as we have seen) multiple, partly overlapping service architectures can be used in parallel. For example, architectures such as Bluetooth, HAVi and home networking technologies such as LonWorks each provide their own service discovery and deployment architecture. In order to achieve interoperability *between* services and clients from *different* technology domains, OSGi defines so called *service gateways* that provide standardized interfaces for the different services.

OSGi has specified these common interfaces as APIs in the Java programming language. An *OSGi Service Platform* in an instantiation of a Java virtual machine, an OSGi framework implementation and a set of Java-defined services, called *bundles*. A bundle is essentially a standard format for packaging resources that are required to run the service. For example, it contains a Java archive (a jar file) with byte-code that implements a specific service. The OSGi specification provides procedures for mapping services from other frameworks, e.g, Jini (which we discuss in Section 4.3.2.2) to the OSGi model. For example, it is specified how a Jini device that advertises a specific service through the Jini lookup service can be adopted as an OSGi service that can be discovered and utilized by OSGi means. Based on the OSGi framework, the AMI-C architecture defines standardized APIs and mappings to selected service architectures, including Bluetooth and MOST.

AMI-C defines network technology specific implementations for an abstract AMI-C transport service called the *Network Application Layer*. This transport layer is essentially an inter-networking convention for the different support *Network Transport Layers* such as Bluetooth L2CAP and UDP/IP (e.g., on an Ethernet link). Figure 3.4 depicts this architecture schematically.

Just like other inter-networking technologies, AMI-C defines network-independent addressing schemes and message formats. The addressing scheme provides three types of addresses: unicast, broadcast and multicast addresses. A unicast address designates a single device (a *functional module*) and provides two components: a device type part (*F-Type*) and an instance identifier (*I-Num*). Both the *F-Type* and the *I-Num* are 8 bits wide. Similar to other inter-networking technologies, AMI-C defines mapping and address-resolution mechanisms for associating network-specific addresses to AMI-C addresses. In addition, initialization procedures are defined that provide initial service discovery and address resolution. For example, if several nodes of the same *F-Type* exist, *I-Nums* can be assigned dynamically, relying on an address

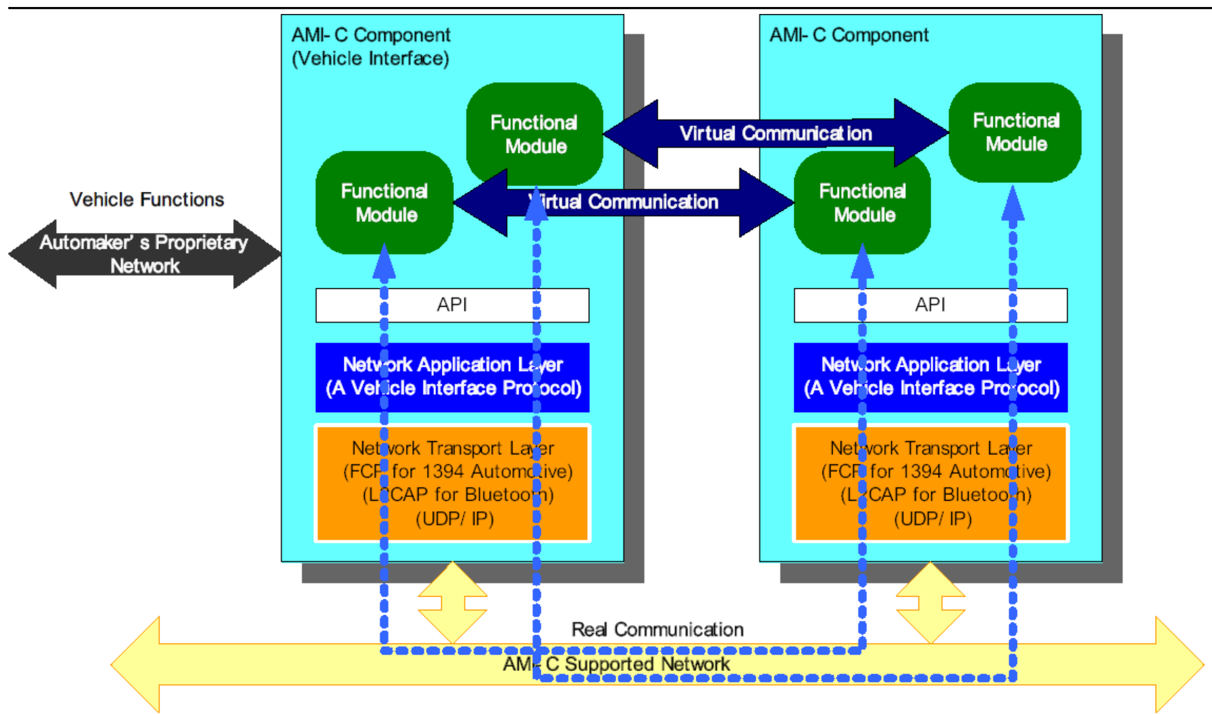

Figure 3.4: AMI-C network transport layer [AMIC03]

allocation and defense protocol.

A unicast address consists of a unique *F-Type/I-Num* tuple and is used to send a message to a specific node. A broadcast address can be used to send a message to all nodes or to all nodes of a specific *F-Type*. Multicast addresses are used for publish/subscribe communication: a component can request another component to send a specific information periodically, and the sender can allocate a multicast address that is used for disseminating this information. Multicast addresses have a fixed *F-Type* and are distinguished by their *I-Num* field. The multicast address allocation uses an *I-Num allocation request* that is also used to allocate unique *I-Nums* for unicast addresses.

AMI-C provides a limited set of message types that are used to realize different standardized interaction schemes such as RPC communication, property inspection and modification and publish/subscribe communication. For example, the message type `COMMAND` is used to request either the start of a program execution, the actuation of a device, access to a resource or the start or end of a property subscription.

3.1.2.5 Observations

The specific deployed and proposed solutions for in-vehicle networks may look awkward to some extent because they partly ignore the pervasiveness of Internet technologies based devices and architectures and hence duplicate a lot of functionality that is already provided by Internet technologies. Nevertheless we can learn a lot of the requirements for networking solutions in this application domain. In the following, we will briefly enumerate the main insights and then sketch an alternative design in Section 3.1.2.6 that takes the existence of Internet-based

technologies into account.

Separation of different network types: The different existing designs and the proposed future architectures have exhibited a clear separation of different network segments for different functions: for safety reasons the crucial control components, i.e., the motor management and the *X-by-wire* components will reside in a separate network that will be designed to fulfill real-time and fault-tolerance requirements.

There will be a dedicated network for "comfort" and multimedia functions, incorporating components including but not limited to radios, MP3 players, and navigation systems. In some designs, there may be a separation between networks for comfort and multimedia functions and networks for inbuilt vehicle components such as electric mirrors and automatically triggered wipers. The latter type of network and the safety-critical *X-by-wire* network may employ proprietary network technologies.

These different networks may be interconnected by gateways that operate on the "application layer" and can provide filtering and translation of messages. For example, some information such as the current vehicle speed from the network for "mechanical" components may be forwarded to components in other networks.

Multimedia and control communication: The discussion of MOST has shown that multimedia communication will be a major application of in-vehicle networking technology. MOST provides the concept of plugging different multimedia (and other) devices together in order to construct a coherent distributed system.

The idea is to connect service providing entities to a network that advertise their service and can be discovered and associated with other entities. Interface descriptions allow for dynamic associations and for the construction of integrated user interfaces.

Integration of user devices: The multimedia and control communication will not only be limited to components that are installed in a vehicle but will also encompass the dynamic integration of user devices such as PDAs, mobile phones or laptop computers that can be connected to the vehicle network and join the group of multimedia and control components.

Standardized interaction schemes: Both MOST and the more general AMI-C architecture have exhibited standardized interaction schemes such as service discovery, RPC communication, property inspection and modification and publish/subscribe communication through event notification mechanisms.

These are largely the same concepts as pursued by some of the coordination and component-framework technologies that we discuss in Section 4.3 and Section 4.4. However, the in-vehicle architectures do not provide these services within an Internet-technologies-based framework but have defined their own internetworking, service discovery and component architecture.

Group Communication: There are many applications that exhibit group communication requirements, which is also reflected by the protocols we have discussed such as AMI-C. For example, the efficient distribution of sensor information that needs to be received by

multiple components can be implemented by multicast messages. There is some information such as the current vehicle speed that is of interest for a lot of components. The current speed is displayed on the instrument panel, it can be received by the amplifier to adapt the volume to the speed and it is taken as an input parameter for the intelligent suspension to adapt the ground clearance.

3.1.2.6 Redesign

With the observations and requirements presented in Section 3.1.2.5 in mind, we will now sketch an architecture for in-vehicle networks that is based on Internet technologies and does allow for the direct integration of IP-based components and user devices without the need for application layer gateways.[8]

The main idea is to use IP, the Internet Protocol, for integrating different network technologies and not to rely on application layer gateways that have to know service discovery architectures and higher layer communication mechanisms for different network technologies. The interconnection of networks of different types can be realized by the use of bridges (e.g., WLAN IEEE 802.11 access points) and routers. For example, a vehicle could provide an on-board Ethernet or IEEE-1394 network link and a WLAN IEEE 802.11 network link that are connected by a WLAN access point that operates as a bridge. In addition, the vehicle could provide for other wireless access technologies such as Bluetooth by employing a Bluetooth access point.

These network elements do not know anything about transport protocols, applications and service discovery architectures. As bridges they forward Ethernet frames and as routers they forward IP packets. Their sole service is to provide an internet that may be comprised of different links.

Devices that attach to the network are Internet hosts, i.e., they provide an IP stack and use IP to communicate with other hosts. For address configuration, auto/zero-configuration mechanisms such as IPv6 stateless auto-configuration [RFC2462] and IPv4 auto-configuration [Cheshire02] can be used, depending on the IP protocol version used. We assume that IP multicast connectivity across all network links can be provided.

Based on this IP infrastructure, we design a service discovery architecture and a coordination protocol that allows to coordinate different components in the vehicle network, which may be part of the built-in infrastructure or which may be dynamically added by a user.

The general idea is to consider applications that are built on this infrastructure as component based systems. This means that applications can make use of third-party components that are discovered, associated and integrated into applications. However in addition to the traditional component based systems approach, we also have to allow for dynamically extending a running application, e.g., when a user's PDA joins the vehicle network, it must be possible for the PDA to take over the control of the vehicle multimedia system or to operate as a secondary controller in addition to the built-in user interface.

It should be noted that in addition to coordination messages, other communication will be required, such as real-time multimedia communication. We assume that RTP be used for this purpose. For example, an MP3 player would send RTP audio streams to the speakers or to an amplifier. However, in order to establish the corresponding sessions, the entities have to

[8]It should be noted that this architecture is only addressing the *multimedia and comfort* networks and *not* the safety critical vehicle-control networks that are needed for *X-by-wire* applications.

discover each other, learn about each others capabilities and negotiate transport parameters. In addition, they require coordination during a session. Thus, we conceive the use of different protocols in this scenario but focus on the requirements of the coordination protocol in the following. The discovery and coordination protocols must provide the following features:

- The service discovery process must include the exchange of interface descriptions in order to allow for dynamically extending the network with services.
- Different standardized interaction schemes must be supported that can be specified in the interface description. In order to be able to issue control commands, an RPC mechanism must be provided but other abstractions such as property inspection/modification must be supported as well.
- The communication mechanism must support group communication that should be mapped to native link layer multicast where applicable.
- The group communication service requires a flexible addressing scheme that allows for grouping entities into receiver groups, for example as depicted in Figure 3.5.

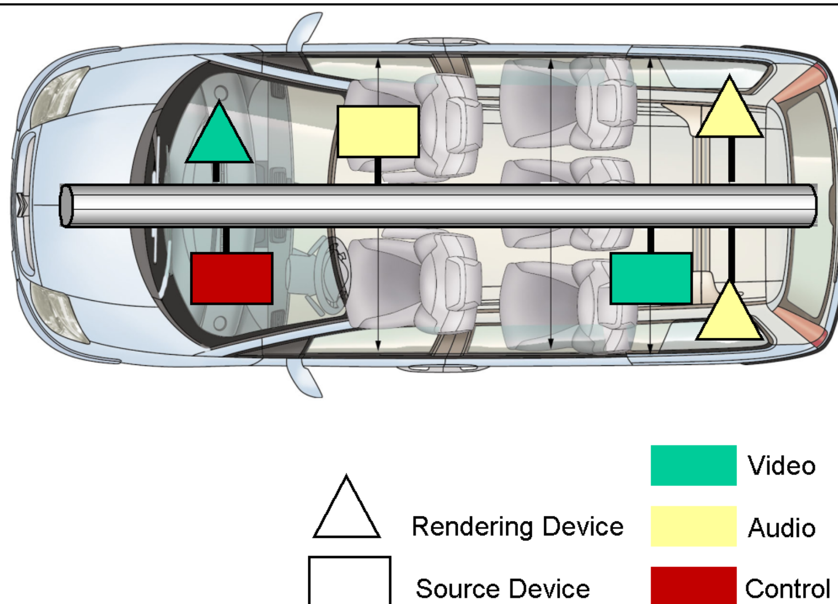

Figure 3.5: Addressing components in an in-vehicle network

Figure 3.5 depicts a schematic overview of an in-vehicle coordination infrastructure, focusing on the addressing aspect. In this network, entities are grouped by type in two dimensions: by the media type they are offering a service for and by their "directionality" class, i.e., whether they can receive or send media streams. For example, in a group communication scenario, a controller would be able to send a generic message to all receiving audio engines by using a corresponding destination address that represents the proper group of devices.

- The set of messages should not be pre-defined. Instead, new applications and new components should be allowed to create new message sets depending on the application's requirements.

There are other requirements such as security that we did not mention here explicitly (architectures such as AMI-C also provide security mechanisms). Obviously the protocols must provide security mechanisms that exclude the possibility that a user in a by-driving car gains control over the vehicle's multimedia infrastructure, be it accidentally or maliciously.

3.2 Requirements

The use cases that we have described in Section 3.1 lead us to the following list of requirements that we intend to fulfill by the design of the Mbus protocol.

3.2.1 Requirement *Ad-hoc communication*

The term *ad-hoc communication* refers to the creation of spontaneous communication sessions between peers that can dynamically join and leave sessions and require no manual configuration to communicate. Both the extensible conferencing endpoint scenario and the in-vehicle network scenario have revealed a requirement for associating devices in an ad-hoc fashion, i.e., integrating devices into coordination sessions without prior manual configuration. The term *ad-hoc communication* is used in different ways:

- Sometimes, the term *ad-hoc communication* is used to refer to *mobile ad-hoc networking* — a technology for creating ad-hoc networks of mobile hosts, which includes the deployment of dedicated *mobile ad-hoc routing protocols*, such as AODV (*Ad-hoc On-Demand Distance Vector Routing Protocol*, [Perkins02]).

- The term *ad-hoc communication* or *ad-hoc computing* has recently also been applied to future office or home networks, in which new devices can be quickly added using different underlying network technologies, such as *IEEE-802.11 wireless LAN* (WLAN, [IEEE99]) and *Bluetooth* [BluetoothSIG01].

  These link-layer technologies themselves provide a so-called *ad-hoc mode*, which describes a way of setting up a network without the need to install and operate network infrastructure components such as base stations. Instead, the stations are able to locate each other and establish a link-layer communication session (if they have been configured appropriately in advance).

It should be noted that link-layer ad-hoc networking might often entail mobile ad-hoc networking for establishing larger networks in an ad-hoc fashion. In this discussion, we are referring to yet another meaning, namely *ad-hoc networking between application entities*: A group of entities establishes a communication session with dynamic membership and without the need of external infrastructure components. However, for the cooperation of application components (that is the subject of the work described here), we state a requirement for *ad-hoc communication* between application components, not for stations that communicate on the link-layer.

However, link-layer and network-layer ad-hoc communication has some implications for application layer protocols that have to be taken into account. For example, ad-hoc networking may result in intermittent connectivity, non-constant routing topology and changing IP addresses. An application layer protocol that is intended to be used in these environments will therefore have to address these issues, e.g., by accommodating more loosely-coupled federations of session members. The requirement for *ad-hoc communication* can be stated more precisely by considering some related functional requirements:

Dynamic membership: It must be possible for entities to join and leave communication sessions dynamically.

Membership tracking: Entities must be able to determine the current set of entities in a communication session. In order to initiate a communication interaction, entities have to know if the required communication peer is available. In addition, entities have to be able to determine when a new entity joins or leaves the session.

For scenarios where an entity is dependent on the presence of other entities, the membership tracking must be rather accurate, e.g., a controller has to be able to verify that the controlled modules are available. For other scenarios, a more loose membership tracking can be acceptable, e.g., when entity coordination is an optional enhancement and the exact name or address of entities is not relevant.

Aliveness tracking: During a session, entities must be able to determine the aliveness of other entities, i.e., they must be able to ensure that an entity has not yet left the communication channel and is still reachable. For example, when an entity has not left a session but is nevertheless disconnected from the rest of the session members, communication peers must be able to detect this situation.

3.2.2 Requirement *Scalability*

Scalability (of communication systems) is a property of communication protocols and systems and means that the service provided by a protocol should scale with increasing utilization, e.g., larger number of communicating entities or larger number of messages per time.

For a local coordination protocol, this means that the protocol should work well with a few (1 to 10) application components but should also work well with many (10 to 100, or even 100 to 1000) components. When we think of inter-hosts coordination, where multiple processes can run on each hosts, entity numbers of several hundreds of entities cannot be excluded.

In addition, scalability (with respect to the number of entities in a network) does not only mean that a system functions well in a situation with increased utilization, but refers to taking advantage of a larger number of entities. For example, a larger number of entities represents a larger number of processors that can work in parallel. Taking advantage of this situation would mean that the coordination protocol enables the parallel operation, e.g., by asynchronous communication mechanisms.

3.2.3 Requirement *Group communication*

In [Kaashoek92], Kaashoek defines group communication as the abstraction of a group of processes communicating by sending messages from 1 to n destinations. The ability to send a

single message to multiple receivers is an important contribution to achieving *scalability*. The specific implications of this will be discussed in Section 6.1.

However, group communication is not only a technique to achieve scalability and efficiency — it also provides new service models because group communication differs significantly from traditional communication paradigms such as remote procedure calls (RPCs) that inherently provide a one-to-one communication only. Many interaction schemes that are useful for the coordination of application components suggest group communication as a natural realization. For example, from an application's point of view, the dissemination of event notifications that are to be received by multiple components should be possible by sending one message to a group address without having to know each addressee.

3.2.4 Requirement *Intra- and Inter-host communication*

Clearly, the coordination of application components must be possible in a distributed fashion, i.e., with components residing on different hosts. The evolution of information technology has produced a variety of personal computation and communication devices that have increasingly been put to everyday use over the last couple of years: from powerful laptop computers to wearable computers, to personal digital assistants (PDAs), to cellular phones. In office (and home) environments, such personal devices mix with stationary ones: in particular PCs and telephone sets. At a well-equipped office workplace, for example, one may easily find a mixture of four, five of even more such components.

Both the desk area computing environment and the in-vehicle networking scenario have relied on the concept of different devices in a network that are being coordinated in order to provide a certain service. Many of these devices are actually Internet hosts, i.e., they are equipped with a network interface and support one of the different link-layer protocols, such as 802.3 Ethernet, 802.11 WLAN [IEEE99], Bluetooth [BluetoothSIG01] or even infrared IrDA [Irda] communication and they provide an IP stack.

The requirement that must be stated here is that a coordination mechanism for these heterogeneous systems must not only provide inter- *and* intra-host communication but also provide *transparent* communication mechanisms, regardless of which devices the individual components reside. I.e., for a communicating entity, the communication procedures for local (intra-host) coordination should not be different from the communication procedures for inter-host coordination. The major requirement *transparent inter-host communication* implicates a set of secondary requirements such as platform-neutral representation of messages etc. These are discussed in detail in Section 6.1.

3.2.5 Requirement *Efficiency*

A set of requirements that affect the operation and the typical usage of a control protocol is summarized under the title *efficiency*:

Timely delivery of messages: For interactive applications, e.g., an application that is decomposed into a graphical user interface component and a component that provides the application logic, it is required that control messages are not delayed by the protocol implementation.

Note that this requirement may conflict with the requirements *reliability* (Section 3.2.7) and *message ordering* (Section 3.2.8).

Low Overhead and efficient representation of messages: The protocol itself must be efficient in terms of message representation and overhead for management information ("protocol headers"). Although no specific underlying transport protocol is assumed at this stage, we have stated in Section 3.2.4 that different link-layer technologies are likely to be used, which will result in different, possibly smaller MTU (*maximum transport unit)* sizes for datagrams than the commonly used Ethernet-MTU size of 1500 bytes.

Efficient usage of network resources: In addition to the mentioned requirement for efficient message representation, there is also a requirement to use scarce network resources as efficiently as possible, again especially considering low-bandwidth network links: For example, when used with a link-layer technology that operates on a broadcast medium, the protocol itself should not inhibit the use of multicast or broadcast transmission of messages that are destined to a group of receivers.

3.2.6 Requirement *Small Footprint*

In Section 3.2.4, we have pointed out that the coordination of application components will have to deal with a variety of networked computing devices, including "smaller" devices, such as PDAs and other embedded computers. In order to be able to use these devices as hosts for application components, the coordination protocol itself must allow for implementations with small memory and computing requirements.

The computational overhead for both protocol implementations and applications should be low. Application programmers should be offered simple APIs, and they should not be required to keep track of management data structures, e.g., for group communication features. Protocol implementations should not require many computation and memory resources. Note that this requirement again may conflict with the requirements *reliability* (Section 3.2.7) and *message ordering* (Section 3.2.8).

3.2.7 Requirement *Reliable Communication*

A coordination protocol for distributed applications must provide mechanisms for reliable message transport between entities. For example, a component that sends messages with RPC semantics to another component must rely on the transport services of the coordination protocol. Moreover, since network outages and system failures cannot be excluded, the reliable message transport cannot be guaranteed in all cases. But applications must be supported by providing them with indications whether a message has been received by the addressee or not in order to take appropriate actions in case of transport problems. However, it should be noted that (for acknowledgment-based reliability mechanisms) a sender can typically not distinguish between lost original messages and lost acknowledgment messages, because both messages could be lost or disordered.

Other requirements such as ACID properties (*atomicity*, *consistency*, *isolation* and *durability*) that are usually attributed to transactions are not stated here directly. However, a coordination protocol should provide either these properties itself or enable higher layers to implement corresponding services.

3.2.8 Requirement *Message Ordering*

Message ordering mechanisms are intended to provide *consistency* in distributed systems by guaranteeing that the messages in a message exchange are received in the same order at all participants. There are different message ordering classes, which we discuss in more detail in Section 4.1.

No general requirement for *total ordering* for all messages that are exchanged in a group of application components is imposed, because this can be costly to implement and would thus contradict the efficiency requirement stated in Section 3.2.5. However, for RPC-like point-to-point communication to work, it is required that the protocol offers the service to impose an ordering on all messages that are sent from one entity to another entity in point-to-point mode. Messages that are sent with this transport service must be delivered in exactly the same order as they have been sent. This property is usually called *FIFO ordering* [Babaoglu93]. In addition, other types of message ordering (such as causal ordering) *may* be implemented without loss of efficiency on top of the basic message passing service.

3.2.9 Requirement *Standardized Interaction Schemes*

We have noted that for the independent deployment of third-party components, interface description techniques are often employed. For example, the AMI-C architecture that we have described in Section 3.1.2.4 relies on dynamic discovery of components and the capabilities. We have seen that the basis for dynamic discovery and subsequent deployment of components is the existence of a well-defined set of interaction schemes. The message passing mechanisms should therefore allow for the implementation of corresponding interaction schemes, in particular:

- RPC communication;
- property inspection and modification; and
- event notification.

3.2.10 Requirement *Security*

The local coordination of application components with Internet protocols is sensitive to a few security threats:

- Messages between application components may be subject to eavesdropping. Depending on the application, this would allow an attacker to gain significant knowledge about the actions a user is currently performing (or about the actions that application components are performing on behalf of a user). Moreover, coordination messages may contain security relevant information, such as personal credentials and keying material that can enable an attacker to conduct other attacks in the future. As a result, there must be mechanisms to guarantee the *confidentiality* of messages that are exchanged between application components.

- In a distributed application, it is extremely important to exclude the possibility that external entities inject malicious control messages into a communication session. Therefore, messages must be authenticated in order to allow only authorized entities to control the application components of the user who “owns” the session.

- In addition to authentication, the *integrity* of control messages must be guaranteed. An attacker must not be able to intercept and change messages that are sent from a legitimate session member to another.

- Another security threat is the injection of messages captured earlier by an attacker, so called *replay attacks*. Syverson provides a taxonomy of replay attacks in [Syverson94]. Essentially, a replay attack is characterized by an attacker who records messages from an on-going communication session and replays them at a later time during the session, without having to know shared secrets such as encryption keys. Receivers that cannot identify the attack would deliver the message multiple times, which could (depending on the specific application) lead to unwanted results.

In summary, the three security requirements for control communication between application components are *confidentiality*, *authentication of senders* and *message integrity* (including the prevention of replay attacks).

3.3 Summary

By analyzing two use cases from quite different application domains, we have obtained a list of requirements for a local coordination service for application components, where the requirements are largely independent of the specific application type. In the following Chapter 4, *Foundations and Related Work*, we will have a look at related work in the area of distributed computing, coordination and local conference control, in order to match our requirements against existing solutions.

Chapter 4

Foundations and Related Work

The design of our coordination framework employs concepts and building blocks from fundamental existing work, such as principles and mechanisms from the distributed computing domain. We will discuss the most important foundations in this chapter and will investigate the appropriateness of different mechanisms with respect to the requirements listed in Section 3.2.

Another objective is to classify our message-oriented coordination approach with respect to existing solutions, e.g., from the component-based systems domain. In addition, we discuss related work, i.e., existing local coordination mechanisms for conferencing systems. This chapter is structured into five sections:

- In Section 4.1, we present *fundamental work* in the area of distributed computing, including a critical view on the RPC paradigm. We describe early approaches for the development of group communication systems, such as the ISIS toolkit and its successors and point at possible problems and open issues.

- In Section 4.2, we analyze some existing local coordination protocols for conferencing systems, i.e., *related work*.

- In Section 4.3, we investigate existing protocols that are explicitly targeted at local coordination in potentially dynamic computing environments such as *Universal Plug and Play*. These technologies, and the corresponding concepts can partly be considered as *building blocks* for our design of a coordination service, but at the same time they represent *related work*, i.e., alternative approaches, that we will evaluate with respect to our requirements.

- In Section 4.4, we discuss component technologies that are frequently employed to implement component-based systems. We define the term *component-based system* and classify the message-oriented coordination approach based upon acknowledged criteria. The analysis of technologies for the development of component-based systems is intended a) to investigate and explain the *fundamental concepts* of that technology and b) to identify *building blocks* for our own design.

- We summarize our observations and conclusions in Section 4.5.

4.1 Distributed Systems

The protocols and applications for conferencing and coordination that are presented in Chapter 2, *Conferencing Architectures* and Section 4.3 are based on research on distributed computing and group communication that we present in this section. In [Tanenbaum2002], Tanenbaum gives the following popular rough characterization of distributed systems:

> "*A distributed system is a collection of independent computers that appears to its users as a single coherent system.*"

The notion "independent computers" describes the entities in a distributed system as autonomous entities that are not tightly coupled. In fact, we are used to think of separate processes that might (but do not necessarily have to) communicate over some kind of computer network. The characterization "appears to its users as a single coherent system" refers to the *transparency* of the distribution. In a distributed system, protocols are employed that provide certain services that are required for achieving this transparency, i.e., the characteristics and difficulties in the communication in a distributed system are made invisible to the user by guaranteeing certain properties of the system.

A particularly important property of interactions in a distributed system is *consistency*, which refers to interactions that manipulate a state, e.g., a bank account in a distributed system for tele-banking. The peers in a distributed system should maintain a *consistent* view of this state, and interactions in a distributed systems should not compromise this consistency. *Transactions* [Weihl93] are a wide deployed interaction type that provides consistency as one element of the so-called ACID properties:

Atomicity: Transactions are atomic with respect to their execution: They are not divisible into sub-actions. This is especially important regarding crashes, lost messages in a communication session etc: The transaction is either executed completely or not at all.

Consistency: Each transaction is based on a well-defined and consistent state of a system and the execution of the transaction leaves the system in a consistent state. A stronger from of consistency can also require that transactions never violate certain invariants that are imposed on a state, i.e., it is guaranteed that the state is always consistent with respect to these invariants.

Isolation: The isolation property refers to the execution order of a group of transactions: If a group of transactions is executed concurrently, the effect is the same as if they were executed sequentially, i.e., the transactions can be serialized without changing the final result.

Durability: A transaction that has been completed cannot be reversed by subsequent failures such as machine crashes or network outages.

These requirements describe the behavior of transactions in distributed systems without prescribing implementation details. Naturally, there are different mechanisms by which some of these properties can be realized and there are different design options for the implementation of distributed systems. For example, when we think of the coupling of entities in a distributed system, we can distinguish *synchronous communication* and *asynchronous communication*. The

first variant means that, e.g., in a request-response communication, the initiator of the request sends the request to the receiver and is then blocked until it receives the response that indicates that the requested operation has been performed. With asynchronous communication and corresponding programming models, a sender process would continue (in parallel to the processing of the request at the receiver) and then eventually receive a response. Obviously, the choice of a communication scheme influences how consistency can be achieved and how much parallelism is possible in a distributed system.

In the following sections, we will analyze different protocols and mechanisms for implementing distributed systems and discuss their relative merits and shortcomings. Section 4.1.1 describes the acknowledged *remote procedure call paradigm*. In Section 4.1.2, we will describe the challenges for group communication, i.e., communication in a distributed system that consists of more than two entities.

It should be noted that the selected technologies are each just representatives of a whole class of protocols and toolkits. However, the RPC paradigm and the ISIS-based toolkits that we discuss later can be viewed as principal representatives that have influenced the design of many other protocols.

4.1.1 Remote Procedure Call Paradigm

Probably the most important aspect of the remote procedure call (RPC) paradigm is *transparency*. The main idea of RPCs is to provide the programmer with an abstraction for performing actions on a remote computer that he is used to: The invocation of a remote procedure call should feel like a local function call as if the requested action would be performed locally — hiding the communication between different processes completely. In order to achieve this transparency, a corresponding protocol implementation must provide a few services:

- The procedural semantics impose strictly synchronous behavior: The caller is blocked until the procedure has been called and the control is returned to the caller. In general, this implies that RPCs provide *synchronous communication*, i.e., there is no parallelism.

- When issuing a procedure call, we take it for granted that the *request* to execute the procedure cannot be lost. In order for RPCs to provide the same behavior, the RPC communication must be *reliable*, i.e., there must be mechanisms to detect message loss and to counteract such problems.

  In order to maintain the network transparency, transient communication failures, retransmissions etc. cannot be signaled to the application. The application just executes a function call and blocks until that call is completed.

- The procedural paradigm implies that a caller calls a named procedure by (optionally) passing parameters that are required to perform the desired operation, and that results from the operation can be passed back to the caller. An RPC protocol therefore has to do the same: It has to provide the possibility to call named remote procedures and it has to provide mechanisms to transfer parameters to the remote system and to transfer results back to the caller. The parameters must be represented in a way such that their semantics are maintained even if caller and callee reside on heterogeneous computer systems with different representation of values. The platform-neutral representation and encapsulation for sending parameters is referred to as *marshaling*.

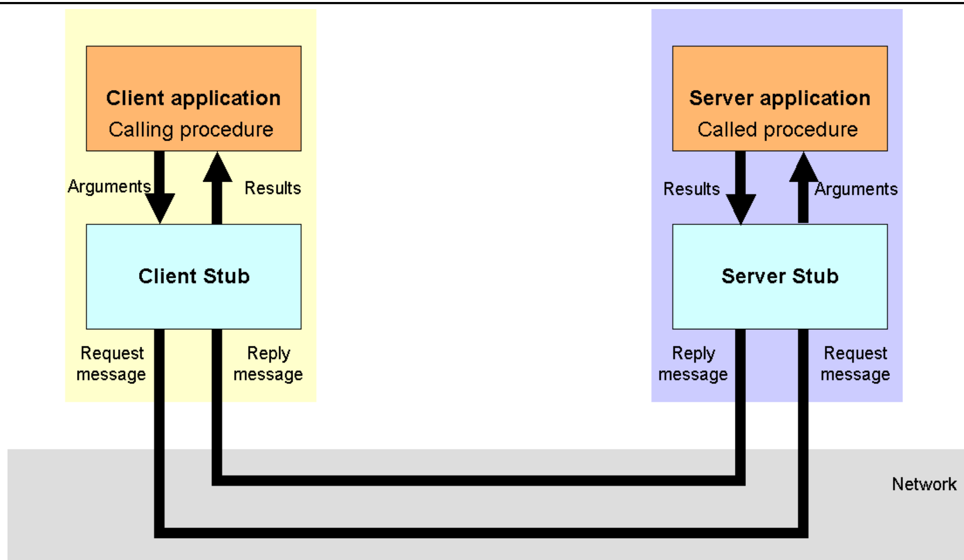

Figure 4.1: Interaction for a Remote Procedure Call

The abstract behavior has been implemented in different systems in different ways. One popular notion is to conceptually split an RPC client system into the user's application code and the *user stub*. This schema is depicted in Figure 4.1. The application code is the user's program that performs local function calls, some of which may invoke procedure calls in the user stub. The user stub generates messages for the invocation of remote procedures by specifying in the message which procedure should be called and by marshaling the parameters that the user has passed. The message is then sent to the server, where it is processed by the *server stub*: the arguments are unpacked and the server stub then calls the server application's function. Upon completion of this function call, the server stub marshals the results from the function call and sends a reply message back to the client. The client stub receives the reply, unpacks the results and passes them to the calling user function, thus returning control to the user's application program, which has been suspended during the remote procedure execution.

Preserving the procedural semantics for remote procedure calls requires a tight coupling between client and server: It must be ascertained that the client calls the right server function on the right host, and that the interface of the local and the remote function actually match. Client stubs provide an interface that is used by application programs, i.e., the application program *imports the interfaces*. A corresponding server application that complies to an interface is said to *export an interface*.

Establishing the coupling between a client stub and a server stub is known as *binding*. Before any remote procedure can be called the client stub must first be bound to an appropriate server stub. The first prerequisite of binding is a *naming concept*: Servers and exported procedures must be named uniquely to allow clients to identify and bind to the correct procedures. For example, in Open Network Computing RPC (ONC RPC, [RFC1831], [RFC1832], [RFC1833]), a widely deployed RPC protocol, a client program needs to know the *RPC program number* and the transport address in order to use a service. The program number maps to a *interface definition* of exported procedures that is known by both server and client. The interface definition contains the signatures of the exported functions, i.e., it contains the names and the argument

type specification of each function.

Service location for RPC is the concept of locating appropriate servers that provide the desired service. In their description of an RPC implementation in [Birell84] Birell et al. describe the use of a distributed database for this purpose. For ONC RPC, servers register with the so-called *Portmapper* on the same host. The Portmapper is a service roster for one host and can be reached using a well-known port number. Clients do thus not have to know the port number for the desired RPC server process. However, they still have to know the host's address in order to query a Portmapper about a certain service. This is usually adequate for the applications that ONC RPC is used for: For example, when a client wants to mount a specific NFS file system it will know which server exports the corresponding file system.

In order to facilitate the creation of distributed RPC-based applications, the process of generating client and server stubs can be automated: The programmer can specify the interface of the exported procedures using an *interface definition language*. A stub generator can use a corresponding interface definition as input for generating client and server stubs, thus ensuring that both client and server rely on consistent interfaces.

Problems with RPC Communication

The RPC mechanism has been designed to provide a clean and efficient model for building distributed systems with request/response semantics, and is has been successfully applied to distributed file systems (NFS) and databases (NIS). However, the RPC paradigm can of course not be applied to every distributed application.

Birman et al. have argued in [Birmann94a] that the RPC mechanism, while representing an efficient solution to simple point-to-point communication scenarios with request/response behavior, was essentially inadequate for the development of complex distributed systems. Their main argument is that not all distributed systems fell into the category of client-server style systems. Instead the class of distributed systems representing *process groups* that perform group communication was gaining more importance for two reasons:

1. Splitting up computations into multiple processes can significantly increase the performance of a distributed system, if computations can be done in parallel.

2. The distribution of a single state among several processes, i.e. the replication of state at multiple processes, can increase the robustness of a distributed system. Crashes can be tolerated more easily if a distributed system with state replication is aware of crashes and network outages and is enabled to react accordingly.

Birman et al. claim that RPC communication is neither able to provide efficient parallel computing nor to provide the robustness through state replication as described before.

The main reason why RPC communication does not support high-performance parallel computation lies in the strict procedural semantics with blocking remote procedure calls. A client that issues a remote procedure call at another process cannot continue in parallel until the call has completed and the control is returned to the calling function. This *synchronous* style of communication does not allow a client to perform any parallel operation while waiting for the completion of a remote procedure call, especially it is not possible to call other remote procedure calls in the meantime.

Introducing parallelism to the synchronous RPC mechanism — e.g., by using multi-threading at a client or by allowing for derived forms of RPC where a client can simply send

a message without waiting for a reply — are no acceptable solution to the problem since this would either add new complexity at other places (e.g., dealing with synchronization issues in a multi-threaded client process) or would deviate completely from the RPC semantics of unifying call behavior for local and remote procedure calls.

Robustness in an RPC-based distributed system is difficult to achieve because of the connectionless nature of (some) RPC implementations. Although the RPC protocol itself deals with retransmissions for compensating packet loss and crashes transparently, there is still room for different types of failures: For example, even if servers maintain some state information about recent RPCs, they may crash after performing an operation but before the reply is sent, reboot before the retransmission and finally execute the operation twice.

Birman et al. identify the lack of a group communication facility in RPC based communication as a serious problem, as in modern distributed systems, groups are used heavily, e.g., processes providing a replicated service, parallel processes performing a computation or processes monitoring the same source of events. Mechanisms for group communication that are required but not provided by RPCs are multi-point-communication, management of membership information and synchronization.

In [Saif2002], Saif et al. have assessed the RPC paradigm with respect to its application to communication within "ubiquitous systems", i.e., networks with ad-hoc communication characteristics, device mobility and demand for group communication, e.g., personal computing environments as we have described in Section 1.1. In addition to the limited support for parallel operation and for group communication that have already been mentioned, the following main problems have been identified:

Not suited for mobility: In a ubiquitous system, there are two kinds of mobility: *logical mobility* (mobile agents and mobile processes that "move" from one device to another) and *device mobility*. Logical mobility is useful in order to achieve load balancing, to provide fault tolerance through replication and migration of services and it can be used to dynamically extend the capabilities of the participating resources in the ubiquitous system. Device mobility refers to devices changing their geographic position and/or their network reachability, e.g., by roaming between networks.

The tight coupling of RPCs and the strict binding of client stubs to server stubs is not supporting these mobility functions. A server process that moves from one device to another cannot be used for remote procedure calls anymore and a client that calls a remote procedure at a process on a device that moved could simply block forever.

Coupling too tight: The static binding of procedures in client and server stubs and the strongly typed interfaces do not meet the requirements for ubiquitous environments. In these loosely coupled environments, different components are designed and implemented separately at different points of time, and not every interface is available at compile time.

Services in ubiquitous systems are not constantly available. Instead components can appear and disappear spontaneously. This requires flexible service location and interface discovery facilities, i.e., services must be able to discover and bind to new interfaces as they become available.

Saif et al. argue that static typing of interfaces is not desirable for ubiquitous systems because for coordination purposes in loosely coupled systems some degree of polymor-

phism and flexibility is required: For example, in a house automation scenario, the command `make the ambient light very dim` could be carried out as `switch off the light` in situations where light dimming is not available.

Summarizing, we can state that the RPC paradigm provides a useful abstraction for a certain class of applications that rely on request/response semantics. The procedural semantics and the transparent calling of remote procedures allows for simplifying the creation of distributed applications because programmers do not have to deal with procedure binding and network transport issues themselves. Obviously, the RPC mechanism has not been designed for group communication, so it is not surprising that RPCs alone do not provide the required communication mechanisms for, e.g., coordinating a group of entities. Birman's argument that the strict synchronous communication style impedes parallelism and can thus lead to poor utilization of computing resources and poor performance is certainly valid.

We have seen that there are certain applications that need other communications mechanisms in addition to RPCs: In ubiquitous systems, the tight coupling of RPCs is inappropriate, and a more flexible approach is needed.

We conclude that the RPC paradigm can only be one element amongst others in a coordination framework. There are interactions that need exactly the simple and clear request/response behavior, e.g., a device that is tightly coupled to a specific controller can be controlled by the use of RPC communication if no parallelism is required. Other interactions, e.g., distribution of events, can certainly be implemented more efficiently using message-oriented communication. For group communication there are many other requirements, such as consistency of distributed states and message atomicity that are not met by the RPC mechanism — because it has not been designed for group communication. We will discuss these requirements and corresponding solutions in the next sections.

4.1.2 Group Communication and the ISIS Toolkit

When we view distributed computing with an emphasis on group communication, we have to consider new requirements and have to face different problems than those of point-to-point communication scenarios. One example of an aspect that is intrinsic to group communication is *message ordering*, because it is generally required to achieve *consistency*. In a point-to-point scenario with synchronous communication, message ordering is of no concern, it is provided automatically by the semantics of the synchronous communication: The messages of a sender arrive at the receiver in exactly the same order as they have been sent. This property cannot be taken for granted in group communication sessions: Senders can send messages "in parallel" and it depends on many factors (network paths to the receivers, load on senders and receivers) how these messages are ordered at each receiver. Inconsistencies can occur when the messages relate to each other (e.g., all message manipulate a distributed state) and are not *commutative*: Different receivers will generate different states depending on the order of the received messages.

Protocols for group communication can provide mechanisms that allow maintaining consistency (amongst other functions). The ISIS toolkit [Birmann87b] is a distributed computing toolkit that provides communication primitives for sending messages to a group of entities guaranteeing consistency and other desirable properties. The ISIS toolkit can be used for implementing distributed systems such as networked file systems, content distribution and parallel

computing applications. The ISIS toolkit is targeted at rather tightly coupled process groups, with stable group membership and a common understanding of messages and their semantics.

In the following, we will present some more details on group communication, presenting the most common problems and typical protocol mechanism to address these problems. Subsequently we will discuss the ISIS toolkit and its communication primitives in more detail.

Group communication and reliable message transport: The prerequisite for group communication is obviously a mechanism that allows sending messages to a group of communication peers. There are different alternatives how this can be achieved: For example, IP multicast is a group communication mechanism that is provided by the Internet's network layer (often with support from the link layer). In networks where such a service is not available, group communication can also be accomplished by establishing a mesh of point-to-point relations between all communication partners. A sender that wishes to send a message to all group members would simply duplicates the message and sends a copy to each participant. Of course a multicast service that is provided by the network is in general more efficient and scalable with respect to the group size.

If we consider a multicast mechanism that is based on a mesh of unicast communication channels, fault tolerance will typically be achieved by employing a reliable unicast transport protocol — however this cannot scale to larger group sizes. An interesting problem is the provision of reliable transport mechanisms for group communication in the presence of unreliable transport mechanisms such as IP multicast. The established solutions for unicast communication, e.g., the acknowledgment-based approach with transmission windows for enabling pipelining, cannot be applied directly to large groups. ACK-based (acknowledgment-based) approaches impose problems with respect to scalability; therefore NACK-based (negative acknowledgment-based) approaches tend to be more usable. Some NACK-oriented reliable multicast protocols are: MTP [RFC1301], MTP-2 [Bormann94a] and MTP/SO [Bormann99], PGM [RFC3208] [Gemmell03], SRM [Floyd97] [Mccanne98], and the NTE protocol [Handley97].

Consistency and message ordering: For enabling processes in a distributed system to act reliably, it is useful to specify the message ordering that can be achieved by the deployed communication mechanisms. For example, if a process responds to a message that has been sent by another process earlier, it might not make sense for a protocol implementation at a third process to deliver the response before the original message has been received and delivered. This is especially true for situations where messages change a distributed state and are not commutative, i.e., the order of applying operations that might be denoted by the messages is relevant for the final state of the system.

Different types of message ordering can be distinguished:

FIFO ordering: FIFO (*first-in-first-out*) ordering is provided when all messages from a process `A` are delivered to process `B` in the same order as `A` has sent them, i.e., this ordering class only considers messages that are sent from one process to another and does not assure any form of ordering between messages sent by different processes.

A typical mechanism for achieving FIFO ordering is the use of sequence numbers or timestamps that increase strict monotonically.

Total ordering: When all broadcast messages are delivered at each process in exactly the same order, *total ordering* is preserved.

Total ordering can be implemented by either relying on a central sequencer, i.e., a process over that all messages are relayed, or by relying on message timestamps. Relying on message timestamps only works if all processes have synchronized local clocks, i.e., this approach requires additional mechanisms for clock synchronization. This make total ordering comparatively expensive: a central sequencer is a potential bottleneck and impedes scalability, and requiring clock synchronization adds unwanted complexity.

Causal ordering: Lamport has described the concept of *happening before* in [Lamport78] that defines an invariant partial ordering of the events in a distributed system. The main idea is that in order to avoid the costs that are associated with total ordering based on timestamps, it can in many cases be sufficient to order only those messages that are *causally related* by relying on *logical timestamps*. The ordering is partial only, because (unless other mechanisms are employed) some messages can appear to have the same logical timestamps, i.e., they cannot be ordered totally.

In Lamport's model the "clock" is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. An event is an abstract action that is performed by a process. The sending or the delivery of a message could also be viewed as an event. The logical time at each process is incremented by every event.

Atomicity: *Atomic multicasts* are messages that are either received by every process in a distributed system or not received by any process. That means, protocol implementations may only deliver a message to the application if they know that all other protocol implementations have also received the message.

One way of implementing this behavior is the use of a central coordinator that relays messages to each process and performs a two-phase-commit-procedure: only after each process has acknowledged the reception of the message, the coordinator indicates that the message may be delivered. Again, the use of a central coordinator implies limited scalability.

Membership control and security: In order to accomplish reliable communication and consistency, a *membership service* is required, i.e., the possibility to find out which processes are currently in the group. For example, for causally ordered messages, it would not make sense to send messages to a new participant if it has not received the logically earlier messages before. Therefore, mechanisms such as *latecomer updates* can be employed or the new member can be excluded from the communication until a new thread of causally related messages is started.

Group security can be divided into two aspects: authentication of messages and encryption. Authentication enables receivers to verify that the sender is a legitimate group member and is often implemented by employing hashed messages authentication codes. Encryption can be used for implementing privacy and access control. Only legitimate members are enabled to receive messages. In order to be effective, group security requires additional management functions such as key management that we shall not cover

here in detail. Baugher et al. provide a description of the Internet group key management architecture in [Baugher03], and Harney et al. have defined GSAKMP, the *Group Secure Association Key Management Protocol*, in [Harney03].

Admission control, latecomer updates etc. are often implemented by relying on a central entity. For example, a new member would request admission at the central entity and the entity would in turn poll the group members. If approved, the new member would be provided with the security parameters and the current state.

The ISIS toolkit provides protocol mechanisms for most of these services (amongst other functions such as creation of process groups, synchronization through token passing and data replication). The service model of ISIS is described as *virtual synchrony* [Birmann87b] — any two processes that receive the same multicast messages or observe the same group membership changes see the corresponding local events in the same relative order (*causal order principle*), and a multicast message sent to a process group is delivered to its full membership. For applications, the send and delivery events appear to happen as single, instantaneous event. Relying on the ISIS virtual synchrony service model, application programmers do not have to solve the consistency and message ordering issues for these functions themselves, instead, the ISIS toolkit provides different protocol primitives that provide this service.

A detailed description of the ISIS protocol mechanisms (and of other distributed computing toolkits that are based on ISIS) is provided in [Birman94b]. Essentially, ISIS provides different protocols for group communication, each of which provide different services with respect to message ordering and atomicity.

The ABCAST (*atomic broadcast*) primitive is used to ensure that the order in which a message is received at a destination is the same order at all destinations of the group. The CBCAST (*causal broadcast*) primitive provides the *causal ordering* of messages. CBCAST guarantees that all messages are delivered in the same order at all sites and additionally allows to enforce that if a message `B` causally precedes a message `B'`, then `B` will be delivered first. The GBCAST (*group broadcast*) primitive is used for group management purposes within an ISIS session: GBCAST can be used to inform group members when another member fails, recovers, joins or leaves the group. For example, when a process `f` fails, it is made sure that all other messages that the process has sent so far are delivered before the failure indication.

The ISIS primitives provide properties such as causal ordering and message atomicity and are the basis for higher-level algorithms that are implemented in the ISIS toolkit such as data replication and state transfer. The ISIS primitives imply a certain cost: they introduce complexity in terms of memory and processing requirements for implementations and they introduce latency for message transport because of message retention and protocol communication. For the implementation of CBCAST, messages may be sent multiple times in order to achieve convergence and both ABCAST and GBCAST involve the communication with a central coordinator using multiple message exchanges.

The ISIS protocols assume an underlying reliable group communication mechanism. It has been noted by Birmann et al. in [Birmann87a] that the protocols are sensitive to network partition failures and have a tendency to block in cases when a group is divided into two subgroups, within which communication remains possible. Due to their complexity, the site-view management protocols need some time to terminate and will therefore perform suboptimally in the presence of frequent failures and recoveries.

The concept of *virtual synchrony* and its implementation in the ISIS protocols, relying on central coordinating instances, lead to serious scalability limitations. [Birmann00] has described this *close-grained cooperation model* to be useful for applications that need to scale to tens of members but not to hundreds or thousands.

Due to the way that protocols such as CBCAST work, the ISIS stack (i.e., the application program if the toolkit is linked as a library to a user application) has to maintain data structures that buffer a number of messages in order to allow the required forwarding of messages. These data structures have to be pruned from time to time, e.g., by applying a garbage collection mechanism. This is typically not a problem for a workstation application but this complexity and the corresponding memory requirements might not be appropriate for smaller devices, e.g., embedded systems. It has to be stated that ISIS was not targeted at these kinds of devices.

In [Renesse94], van Renesse has noted that the ISIS toolkit turned out to be too complex for many applications: because it tried to provide many features and different forms of message ordering and fault-tolerance it became too difficult to use and too difficult to maintain. In addition, performance problems have been noted, especially for applications that do not actually need most of the features but suffer from the overhead on behalf of mechanisms that are intended for much more demanding types of applications. Due to its rather monolithic, inflexible structure it was not easily possible to customize the toolkit and to leave out unneeded protocols [Birmann00]. These experiences have lead to the development of successor systems such as the Transis system [Dolev95] and the Horus system [Renesse94].

Despite these deficiencies, ISIS has been deployed in many applications such as trading stocks and managing telecommunications networks. The concept of virtual synchrony for fault-tolerant group communication has influenced the design of many successor approaches and the design of reliable multicast protocols.

4.1.3 Lessons Learned

In this section, we have analyzed protocols and frameworks for distributed computing with a focus on group communication. The remote procedure paradigm that we have discussed in Section 4.1.1 is an important abstraction that is applicable to many distributed computing scenarios. Making remote procedure invocations appear as local procedure calls is the main concept of the RPC paradigm that has also led to its wide deployment. While this paradigm is suitable for classical client-server interaction, it is less useful for group communication and applications that rely on asynchronous, parallel operation. In addition, the tight coupling of communication peers does not allow for the flexibility that is required for ubiquitous computing scenarios where entities cannot be statically bound to each other using strict interfaces. We believe that while the fundamental paradigm of RPC communication will be useful in many applications, RPC communication should not be used *as the sole basis* of a (group) communication system, because this would incur too much inefficiency and inflexibility.

The ISIS toolkit (Section 4.1.2) is an example of a general-purpose group-communication framework that addresses the most important problems of message ordering (amongst other problems) and reliability. The main technical merit is the definition of the *virtual synchrony* concept and its implementation using the ISIS primitives. Virtual synchrony provides applications with a well-defined service definition for group communication and still allows some degree of parallel operation, e.g., for message exchanges that are not causally related. We have seen that implementing the ISIS protocols can be associated with a significant cost and that us-

ing the protocols can result in performance and scalability problems. We have also seen that the intertwined ISIS primitives have led to a monolithic framework that does not allow for a high degree of customization and incurs some significant complexity. Summarizing we can state that ISIS and similar systems are targeted at tightly coupled process groups and are intended for applications that require absolute reliability and can tolerate the performance and complexity cost. It is certainly not suited for group communication in ad-hoc scenarios and also not designed to scale to very high numbers of processes.

The successor systems Horus and Ensemble, while providing the same virtual synchrony service as ISIS does, are interesting because of their building block approach and the idea to compose layer modules into customized protocol stacks. While this idea and its implementation in Horus increase the flexibility of the framework, the building block approach does not necessarily decrease the complexity. We have seen that the approach can incur performance problems if layer modules are composed without further optimization. The Ensemble system and its approach of having the protocol stack analyzed and optimized by automatic tools using a programming environment with strong semantics is interesting. Not only does it provide the possibility for automated optimizations but is also the basis for formally verifying the correctness of a composition and the layer implementations and for providing dynamic adaptability to observed conditions. While some of these ideas have later been re-used in more widely deployed protocols (e.g., the building block approach for the standardization of reliable multicast), the concrete systems have mainly been deployed in the academic community. Similar as ISIS itself, these frameworks are not intended and not suitable for ad-hoc communication scenarios.

We believe that, for these scenarios, a less ambitious approach would be more suitable, i.e., an approach that does not provide strong guarantees for message ordering, reliability and atomicity but bears a lesser degree of complexity and provides a reasonable quality of service for typical usage scenarios.

4.2 Local Coordination (Vertical Control)

Whereas the discussion of conference control protocols in Chapter 2, *Conferencing Architectures* provided an analysis of *horizontal conference control protocols* that do not address local coordination (with the exception of CCCP), we will turn to *vertical conference control protocols* in the following subsections and present two representative basic approaches for local coordination. In our discussion of endpoint architectures, we have already touched upon the relation of inter-system control communication (such as call signaling and conference course control) and local coordination. In fact, local coordination has been characterized as *vertical control* [Ott97] as opposed to horizontal control that is implemented by inter-endpoint control protocols. In this architecture, vertical control can be viewed as an implementation tool for horizontal control, i.e., a conference controller that communicates with controllers of other endpoints could employ the local, vertical control mechanism to implement control commands.

In Section 4.2.1, we describe the *LBL Conference Bus*, one of the first approaches for the coordination of entities in a local conferencing endpoint, and in Section 4.2.2 we present the *Pattern Matching Multicast*, that can be viewed as a second-generation protocol.

4.2.1 LBL Conference Bus

The LBL Conference Bus [McCanne95] is a local coordination mechanism that has been designed to allow for the coordination of individual media tools that have been developed at LBL (Lawrence Berkeley Laboratory) such as the visual audio tool *vat*, the video tool *vic* and the shared whiteboard *wb*. These media tools have become popular in the research community under the name *Mbone tools*, as they have been developed for multimedia conferencing over the IP multicast backbone (*Mbone*) of the Internet. The LBL Conference Bus is the first conferencing specific coordination protocol.

The main design goal for the LBL Conference Bus was to provide a mechanism that enables the composition of stand-alone tools into integrated applications without the need to change or extend existing programs. The Mbone tools have been developed using the *Tcl/Tk* scripting and GUI-toolkit environment (see Section 4.4.3). In general, the time-critical functions have been implemented in C or C++ and have been "glued together" in Tcl/Tk which was also used to implement the graphical user interface. Tcl/Tk provides a simple interprocess communication mechanism called the *Tk send command*. This command could be used in a Tcl/Tk application to send a Tcl command to another Tcl/Tk application (that is addressed by the window name). This simple mechanism allowed the execution of arbitrary commands in remote applications (running on the same host).

It was obvious that this mechanism was too simple and too generic to be useful for coordinating multiple conferencing tools. This perception has led to the development of the LBL Conference Bus. It was used for the following purposes:

Voice-switched windows: When used together with the audio tool vat, the video tool vic could receive indications about the current speaker over the conference bus and switch the viewing window to the corresponding person automatically. These so-called `focus` messages are broadcast on the conference bus and could be processed by other applications as well.

Floor control: Floor control could be implemented by explicitly sending `mute` or `unmute` messages from a local coordinator over the conference bus to all media applications, i.e., all participants except for the appointed speaker are muted.

Synchronization: Because of the independence of the different media streams and the corresponding applications in an Internet multimedia conference, the streams are initially not synchronized to each other. Each application has its own jitter buffer and calculates the dynamic play-out delay with respect to the experienced delays for one particular stream.

In order to achieve lip-synchronization (i.e., the synchronization of an audio stream with the corresponding video stream) the media tools broadcast their current play-out delay on the conference bus.

Device arbitration: The conference bus was also used to coordinate device access by different applications of one or multiple conferences.

Messages have a type field, which does not appear to be used, and a process-ID as the destination address. A single, free form, text command is sent in each message (and interpreted directly as a Tcl command string by the recipient). Messages on the LBL conference bus are sent using IP multicast, with zero time-to-live (restricting the multicast to a single host). Each

process listens on two multicast groups, one for messages affecting all sessions, and one specific to the individual session. There is no reliability and no security included in the LBL conference bus protocol. We note that the key features of the LBL conference bus are the use of IP multicast for local communication and the use of multiple communication channels (different multicast groups).

4.2.2 Pattern Matching Multicast

Pattern Matching Multicast (PMM, [Schulzrinne95]) is another local coordination mechanism for teleconferencing applications that relies on the concept of message exchange in a host-local IP multicast group. It has been developed for the NeVoT (Network Voice Terminal), an early RTP-based audio conferencing system.

PMM supports three different transport methods: a centralized architecture with a message replicator that distributes messages to registered entities via either UDP/IP or TCP/IP and a de-centralized model using host-local IP multicast without a central message replicator. The replicator process can also act as a message filter by sending certain messages only to entities that are interested.

Messages are represented in ASCII and consist of an address specification and a textual command. The address specification is a hierarchically constructed identifier with three components: A conference identifier, the media type and a media instance. E.g., the address `Conference1/audio/3` designates the third audio session in the conference named `Conference1`. The textual command consists of a procedure name and optional space-separated arguments and is intended to be directly interpretable by a Tcl interpreter.

```
Conference1/audio/3 cname joe@example.com
```

Figure 4.2: Sample PMM message

Figure 4.2 is an example of a PMM message. The semantics of this message are that the third audio tool in the conference `Conference1` should use `joe@example.com` for the RTCP-CNAME field.

In order to enable message filtering, PMM entities can register with the central replicator for messages that they are interested in. However, this does not seem to be applicable for the IP multicast based transport that is commonly used. Registration and de-registrations are messages that are prefixed with a `+` or `-`, respectively. In such a message, for both the address and the command wildcards can be used. For example, the registration message

```
+ Conference1/audio/* *active
```

could been sent in order to receive all messages in `Conference1` that are related to audio and provide a command that ends with `active`.

PMM has some open issues: There are no mechanisms for reliable transport (for the UDP based unicast and multicast transport variants), and it provides no security. The latter is especially problematic with respect to the concept of interpreting command strings directly by a Tcl

interpreter. Although the concept of registering for specific messages seems to be attractive it seems to imply the use of a central message replicator and is thus not very scalable.

4.2.3 Summary

The local coordination protocols that we have described in the previous sections are quite limited in scope and consequently notably simple, as they have actually not been designed with a broader conference control architecture in mind. For example, the LBL Conference Bus has initially been created and used for lip synchronization between audio and video engines within a conferencing endpoint. The LBL Conference Bus used host-local multicast communication in order to directly multicast control commands to entities.

PMM augments this idea of local coordination through host-local multicast communication by an addressing concept that (similar to CCCP) introduces a wildcarding mechanism. The protocol provides a "publish/subscribe"-like mechanism that relies on a central "replicator" entity.

Table 4.1: Comparison of the LBL Conference Bus and PMM

	LBL Conference Bus	**PMM**
Ad-hoc communication	no[a]	no
Scalability	yes	limited
Group communication	yes	yes
Intra- and inter-host communication	no[b]	no
Efficiency	yes	yes
Small Footprint	yes	yes
Reliable Communication	no	no
Message Ordering	no	no
Standardized Interaction Schemes	no	no
Security	no	no

[a] Both the LBL Bus and PMM do not provide membership aliveness tracking.

[b] Intended for host-local coordination only.

Table 4.1 shows a comparison of the LBL Conference Bus and PMM with respect to our requirements described in Section 3.2. We conclude that both the LBL Conference Bus and PMM represent first attempts to achieve coordination between application entities in conferencing endpoints and provide some interesting features such as the use of local multicast communication or PMM's filtering concept. However, both approaches are limited because they only provide a very basic message exchange, i.e., they do not provide different interaction types, and they also do not address security, which we have described as a fundamental requirement in Section 3.2.10.

4.3 Coordination and Ad-Hoc Communication

Obviously, the conferencing domain is not the only application area for coordination protocols. In the following subsections, we will analyze the characteristics of general-purpose protocols and frameworks for device coordination and ad-hoc communication and highlight the typical services and implementation techniques of some selected approaches.

The growing pervasiveness of networked devices, the application of Internet technologies and distributed computing approaches to new application areas have fueled the interest of researchers and developers in distributed computing systems that accommodate the ubiquity of Internet technology based systems, where loosely coupled, heterogeneous systems join distributed computing sessions in order to provide a common service.

The requirement and the underlying assumptions for the development of distributed applications for ubiquitous networked computing devices can be quite different from those that we have identified for *synchronous RPC communication* (Section 4.1.1) and for *virtual synchrony* based distributed applications. For example, usage scenarios can provide a much more dynamic group membership behavior of systems. In addition, it is not always possible to rely on static interfaces that are well defined and well-known in advance. Instead, a loose coupling between potential session members can be observed.

In [Waldo94], Waldo et al. argue that for applications, where entities cannot be fully described by the specification of the set of interfaces they support and their corresponding semantics and where entities themselves may be mobile and not constantly available, the traditional approach of distributed system development is not applicable. The authors' criticism refers to systems for object-oriented computing such as CORBA (Section 4.4.4) that are essentially based on the RPC paradigm.[1]

Waldo et al. argue that this approach ignores the differences between local and distributed computing such as the difference in latency, memory access, partial failure and concurrency and propose an approach where these irreconcilable differences are taken into account for the design and the implementation of distributed applications. In their model (which is intended for object-oriented distributed applications), implementers will have to know whether they are sending messages to local or remote objects. Programming distributed applications will thus require the use of different techniques than those used for non-distributed applications. The goal of this approach is to help application developers avoid making mistakes by ignoring the difference of local and remote computing, enabling more robust and efficient applications.

These considerations lead to distributed systems where the distribution of communicating entities is not hidden but is the underlying assumption. In such systems, the focus is on *coordinating* theses entities; therefore these systems are also called *distributed coordination-based system*. In addition to the characteristics that Waldo et al. have described, these systems provide a set of interesting features, which we will generalize in the following.

In a classification of distributed systems in [Tanenbaum2002], Tanenbaum and van Steen describe *distributed coordination-based systems* as a class of distributed systems that assumes an inherent distribution of systems into components, where the main difficulty for application

[1] In such systems, an object, regardless of the location (remote or local) is defined in terms of a set of interfaces declared in an interface definition language, and the implementation is independent of this interface. The main idea of such systems is to make the distribution of objects and the procedure of method invocation *transparent* to the application, just as the RPC mechanism does for procedure calls.

development is to *coordinate* the various components of such a system. An important characteristic of coordination-based systems is the *separation between computation and coordination*: a distributed system is viewed as a collection of (possibly multi-threaded) processes, each of which performing a specific computational activity, which in principle is carried out independently of the activities of other processes.

The *coordination* of this process group "forms the glue that binds the activities performed by processes into a whole", and distributed coordination-based systems provide different mechanisms that implement this coordination. Tanenbaum and van Steen distinguish these systems in two dimensions: with respect to their *temporal coupling* (i.e., whether all components run and communicate at the same time or not) and with respect to their *referential coupling*. The latter refers to the question whether communications have to explicitly reference each other in a session, e.g., by the use of addressing mechanisms. For example, IP multicast sessions do generally not require referential coupling as messages are sent to an anonymous group and receivers do not have to know the address of senders to receive messages from them. However, such sessions typically require the use of a rendezvous point that enables entities to discover and locate other entities (for IP multicast this would be the multicast group); for this reason Tanenbaum and van Steen characterize this model as *meeting oriented*.

In the rest of this section, we will discuss protocols and frameworks for the development of *distributed coordination-based systems* that have influenced the design of the Mbus protocol. In Section 4.3.1, we describe the *TIBCO Rendezvous* messaging system, and in Section 4.3.2, we analyze solutions for *service discovery* (that can be required for establishing a distributed coordination-based system with temporal coupling) and for a coordination model that is sometimes referred to as *Plug and Play*. In this section, we describe the *Service Location Protocol* (Section 4.3.2.1), the Jini framework Section 4.3.2.2 and the *Universal Plug and Play* approach (Section 4.3.2.3).

4.3.1 TIBCO Rendezvous

TIBCO Rendezvous is a messaging system for local and wide area communication that relies on a coordination model where processes are in general referentially uncoupled, but temporally coupled. TIBCO Rendezvous is based on the concept of a (comparatively) simple *information bus* [Oki93] that provides group and point-to-point communication services for large scale distributed systems and can be considered as an alternative approach compared to the ISIS virtual synchrony service.

The concepts of TIBCO Rendezvous are described in [TIBCO02]. The developers of TIBCO Rendezvous have coined the term *subject-based addressing* to describe the way that processes communicate: Messages are not sent to a specific destination address but are rather tagged with a *subject* and are then sent to a group of potentially interested receivers.

Receivers do not specify the source address of senders they want to receive from but *subscribe* to a set of subjects they are interested in. The concept of *subject-based-addressing* is often also referred to as *publish-subscribe communication*.

4.3.1.1 Ordering and Synchronization

Compared to systems such as ISIS (Section 4.1.2) and its successors, TIBCO Rendezvous does not provide message ordering semantics such as *causal ordering* and *total ordering*. In a pa-

per entitled "Understanding the Limitations of Causally and Totally Ordered Communication" [Cheriton93] Cheriton and Skeen have denied the applicability of *causally and totally ordered communication support* (CATOCS) to several classes of distributed applications as the mechanism on the group communication layer were not able to deal with application semantic ordering constraints. Cheriton and Skeen also argued that relying on CATOCS can cause efficiency problems with respect to communication and computational overheads. [2]

Cheriton and Skeen suggested that message ordering should therefore be dealt with on the application layer if required at all. The TIBCO Rendezvous system (that is built on work done by Cheriton and Skeen) is based on the mentioned *information bus* concept and only guarantees FIFO-ordered message delivery per source.

TIBCO Rendezvous provides three fundamental message-sending operations:

`send`: The `send` operation is a non-blocking primitive that is used to send a message under a certain subject asynchronously to a group of receivers.

`sendreply`: The non-blocking `sendreply` operation is used to reply to messages that have been received on a certain subject. The reply is then sent with a corresponding `reply-subject` and delivered to the original sender if it has subscribed to its reply-subject.

`sendrequest`: In addition to the non-blocking `send` operation there is also a blocking `sendrequest` operation that can be used to send a message synchronously by blocking the calling application until a corresponding reply has been received. The reply is sent directly to the address of the original sender, using point-to-point communication.

In addition, TIBCO Rendezvous provides a service called *transactional messaging* that allows sending messages to peers as part of a transaction. This mechanism is layered on top of the basic messaging service and employs central entities, so-called *transaction managers* to temporarily store transaction messages until they have been delivered to all subscribers.

4.3.1.2 Architecture and Communication

TIBCO Rendezvous relies on a multicast network for implementing the group communication services. Every message that is published by an entity is multicast to all other hosts on a network. Receivers that have not subscribed to the message's subject, discard the message. The multicast transport is using native multicast mechanisms such as IP multicast where applicable. Messages that are sent to a dedicated addressee, e.g., replies to `sendrequest` operations, are sent via unicast transport. Each process can create a unique subject name, called an *inbox name* that can be used by other processes to directly send messages to that process.

Messages in TIBCO Rendezvous are *self-describing*, i.e., they can be received and processed without having to know the exact message structure in advance. This is implemented by constructing messages as a set of fields, each of which providing attributes such as `field name` and `data type`.

[2] In a response to this paper [Birman93], Birman has defended ISIS's CATOCS services and in turn denied the applicability of Cheriton and Skeen's criticism by emphasizing the need for a group transport layer that addresses message ordering in order to allow for the design of robust and efficient applications.

The subject of a message is a sequence of character strings that are separated by dots, syntactically similar to names in the Domain Name System (DNS). When subscribing to messages of a certain subject, a subscriber can specify a *partly qualified* subject name through the use of *wildcard characters*. For example, the subject specification `media.*.received` could be used to subscribe to messages on the subjects `media.audio.received` and `media.video.received` (a subject fragment may be replaced by `*`). The subject specification `media.audio.>` can be used to subscribe to messages on all subjects that start with `media.audio.` (all subjects in the `media.audio.` hierarchy).

It should be noted that, in general, TIBCO Rendezvous implements group communication, i.e., the distribution of a single message to multiple receivers, by relying on the *receiver-initiated* selection and filtering of messages. Typically, senders multicast messages with fully-qualified subject, i.e., without wildcard elements, and only receivers specify wildcards in their message filtering specifications. Although in theory, it would be possible to send messages with partly qualified subjects, the TIBCO Rendezvous documentation [TIBCO02] deprecates this.

While the basic TIBCO Rendezvous transport service provides group communication within a local (multicast-enabled) network, the TIBCO Rendezvous specification additionally allows for coupling multiple local TIBCO Rendezvous networks by establishing a point-to-point overlay network, in which pairs of TIBCO Rendezvous routers are connected through TCP connections. Each of these routers knows about the topology of the overlay network and forwards messages based on a computed multicast distribution tree. TIBCO Rendezvous routers can be configured to only forward messages from a local network to remote routers that match a certain subject specification.

4.3.1.3 Reliability

TIBCO Rendezvous addresses reliable message transport in the presence of transmission failures with negative acknowledgments: Each message is assigned a sequence number that increases linearly by each message. Because messages are always sent to all entities, receivers can detect missed messages when receiving a subsequent message from that entity. The receiver can then request a retransmission for the missing messages. Senders have to keep transmitted messages for 60 seconds in order to be able to answer retransmission requests.

In addition, TIBCO Rendezvous provides another reliable transport mechanism called *certified message delivery*, that guarantees stronger reliability that the retransmission based scheme. The basic idea of this mechanism is to acknowledge each message explicitly. *Certified Message Delivery* is layered on top of the regular TIBCO Rendezvous message transport and is implemented by a message storage service called a *ledger*. A ledger is responsible for keeping track of the sent and received certified message and can be implemented as a *process-based ledger*, providing certification for the lifetime of a process or as a *file-based ledger*, providing a persistent certification service.

Instead of using UDP over IP multicast, recent TIBCO Rendezvous versions have been augmented with support for a reliable multicast protocol: applications can alternatively choose to use PGM, the *Pragmatic General Multicast Protocol* [RFC3208], [Gemmell03]. PGM is a reliable multicast transport protocol for applications that require ordered or unordered, duplicate-free, multicast data delivery for many-to-many communication. It meets the requirements for a TIBCO Rendezvous transport protocol that can be used instead of plain IP multicast and provides additional reliability, especially for wide-area coordination sessions.

4.3.1.4 Programming Model

The message-oriented coordination model that is provided by TIBCO Rendezvous leads to an asynchronous, *event-notification-based interface* from an application point of view. In TIBCO Rendezvous implementations, an application registers a callback function when subscribing to a subject. The local TIBCO Rendezvous implementation on a host provides a so-called *TIBCO Rendezvous daemon* that provides the message sending and local delivery functions for all entities on a host.

When the TIBCO Rendezvous daemon has received a message for a subject that matches a local subscription, it delivers the message by adding the message to a so-called *event queue*. When more than one entity has registered for a certain subject, the message will be put onto multiple queues. Each queue is processed by a so-called *event dispatcher* that takes a message from the head of the queue and calls the callback function that has been registered for the given subject of the message.

In order to not restrict applications to process messages in the order they have been received, it is possible to employ multiple event queues, i.e., assign an event queue to certain subjects. In a multi-threaded environment, each queue can be processed by an independent event dispatcher. Furthermore, it is possible to group event queues into so-called *queue groups*, each of which is handled by an independent event dispatcher. The queues within such an event group can be assigned different priorities, so that the dispatcher will dispatch all messages from the top-priority queue first before processing the next queue.

4.3.1.5 Summary

Although TIBCO Rendezvous is (at least partly) targeted at the same application domain as ISIS (large-scale coordination of process groups), the *subject-based addressing* messaging that is provided by TIBCO Rendezvous is a different class of group communication than the *virtual synchrony* based approaches that we have described in Section 4.1.2 and can be considered as a simplification with a focus on *coordination* instead of tight coupling of entities in a distributed system. TIBCO Rendezvous is therefore targeted at more loosely coupled process groups, where processes do not need to know each other's addresses to be able to communicate, and where processes do not need to know the format of messages in order to receive and process them. Essentially, TIBCO Rendezvous provides a message passing mechanism for process groups with *receiver-based filtering* and *self-describing messages*. For message transport, TIBCO Rendezvous assumes a generally reliable local multicast/broadcast network but can also be used with PGM as a reliable multicast transport protocol. The extensions for wide-area communication do not rely on native multicast facilities of the underlying network but are achieved by application layer routing on the basis of a reliable transport protocol.

Whereas in ISIS (and its successors) fault tolerance is achieved by providing a transport layer for group communication that employs protocols for reliable transport and message ordering, TIBCO Rendezvous itself does not address many of these issues but leaves them to the application. The reliability mechanism that is based on negative acknowledgments and retransmission is problematic because it can lead to implosions. In case many entities did not receive a certain message, all of them would send a retransmission request. In addition, the mechanism can lead to unnecessary negative acknowledgments and retransmissions: a process can only decide how to process a certain message after it has received it and will therefore request the

retransmission of messages it has to discard afterwards because of its subject-based filtering rules.

The transactional messaging relies on central entities (the transaction managers) and leaves the implementation of the ACID properties to the application, because transactional messaging is merely a mechanism to group published messages into a transaction context that guarantees the combined delivery of the messages.

From an application point of view, TIBCO Rendezvous could be classified as a coordination mechanism that is targeted at process coordination and news distribution applications, such as messaging within a corporate network. It is less suited to some of the application domains, in which CATOCS-based systems are used, e.g., applications that are sensitive to message ordering and reliable communication and that require a tight referential coupling of components, such as coordination of flight-control workstations.

Summarizing, TIBCO Rendezvous provides an interesting implementation of the concept of a referentially uncoupled distributed coordination-based system. In particular, we note the concept of *subject-based-addressing* with receiver-defined filtering rules and wildcarding, the possibility to receive self-describing messages with the need for static interface definitions and the asynchronous, event-based programming style that is suggested by the TIBCO Rendezvous communication model.

4.3.2 Service Discovery and "Plug and Play" Solutions

Having analyzed the protocol mechanisms for the message-based TIBCO Rendezvous coordination framework, we now describe some work that can also be characterized as support for loosely coupled process groups, but focuses less on the group communication aspects. Instead, the protocols that we describe in the following address the issues of locating and associating with (potentially unknown) communication peers in order to establish a coordination session.

We start this discussion with a presentation of an acknowledged standard for *service discovery* in Section 4.3.2.1: the *Service Location Protocol* (SLP). SLP represents techniques and protocol elements that can be found in many similar protocols in this area. This is followed by a description of Sun's Jini framework in Section 4.3.2.2 that provides service discovery and service association for the Java environment. Finally, we analyze the mechanisms of Microsoft's *Universal Plug and Play* approach for service discovery and device control in Section 4.3.2.3, which is targeted at service control in ubiquitous computing environments such as home-networking scenarios.

4.3.2.1 Service Location Protocol

The *Service Location Protocol* (SLP, [RFC2608]) is a framework for service discovery and service selection in IP networks. It is intended for services such as network printers, remote file systems in an administrative domain that require the knowledge of certain parameters such as host addresses, port numbers and device capabilities. Without SLP, these parameters would have to be configured manually on the devices that want to use the services.

SLP is a very lightweight protocol: Essentially, an *SLP user agent* (a client application that is looking for a service) sends a request for a service and receives responses from *service agents* (entities that advertise services) or *directory agents* (entities that aggregate service advertisements from multiple service agents). The request can either be directed to a known

service agent or directory agent or can be sent via multicast to a standard multicast group using a well-known port. Service agents never advertise their service using multicast or broadcast announcements.

Reliability is achieved by having clients retransmit requests that have not been answered. An exponential back-off is used for retransmitted requests. In order to increase scalability for multicast requests, a user agent can add the addresses of service agents that have answered before to a *previous responders list*. A service agent that sees its address in the previous responders list of a multicast request does not answer the request.

Directory agents are intended for larger networks in order to enhance scalability. If directory agents are present, service agents and user agents can discover them, and service agents can send their service descriptions to directory agents and have them answer service requests on their behalf. In these scenarios, user agents send service requests directly to a directory agent. There are three different ways how a user agent can discover a directory agent:

1. A user agent can issue a multicast service request for the *directory agent*, e.g., on application start-up.

2. The directory agent multicasts periodic, unsolicited advertisements that user agent and service agents can listen for. The *IPv4 Local Scope* [RFC2365] is used for SLP multicast requests and directory advertisements.

3. User agents can be configured to use a specified directory agent. RFC 2610 [RFC2610] specifies DHCP options for this purpose.

Services can be grouped together using a *scope concept*: Services can be advertised for certain scopes only, and user agents can query for services in a given scope, e.g, a work group or a room. Scope identifiers are strings that can be specified in requests and are administratively configured at directory agents and service agents. A user agent that is not configured with a particular scope will discover all available scopes and allow the client application to issue requests for any available service.

User agents can specify three types of information in service requests: scope identifiers, a service type and a query predicate. The *service type* is a fragment of a *SLP service URI*, a URI that can be used to specify the type and the address of a service, e.g., `service:printer:lpr://printer.example.com`. In this example, `service:printer:lpr` is the service type (divided into the abstract service type `printer` and the concrete service type `lpr`). In a service request, a user agent can either specify `service:printer` or `service:printer:lpr` as service types. Requests for `service:printer` solicit response with service descriptions for all available printers, regardless of the concrete type. In order to locate a directory service, a user agent would set the service type to `service:directory-agent`.

A *filter predicate* can optionally be specified to query a service with respect to its *attributes*. Each service can be registered with a set of attributes that can provide one or multiple values. A filter predicate in a request can be compared to the attributes in order to check whether it matches the service specification. A predicate is an LDAPv3 search filter [RFC2254]. The following expression is an example for a predicate and could be used to query for a printer service: `(| (pagesize=A4) (pagesize=A3))`.

SLP also provides for optional authentication of service URIs and service attributes, enabling user agents and directory agents to verify the integrity and authenticity of service URIs

and attributes of SLP messages. However, all involved systems must be administratively configured to generate and to check authentication information. Confidentiality is not provided by SLP.

SLP provides mechanisms for enterprise service discovery and is intended to remove the need for static, manual configuration of clients of file services, printers and similar devices. One major requirement for these applications is scalability with respect to the number of services. Therefore, SLP in general does not rely on periodic multicasting (or broadcasting) of service announcements. Instead, user agents have to query actively for services (or for service directories). This reflects the typical usage scenario of enterprise service discovery: A client application usually wants to locate a persistent service such as a printer and configure itself to be able to use that service.

In summary, SLP is interesting because it is an application-independent mechanism for discovering services in a local network, i.e., within an administrative domain. One of SLP's merits is the scalability that is achieved by a well-designed usage of multicast communication and aggregation of service descriptions by directory agents.

SLP has been designed to provide a solution to the problem of locating services, not to the problem of associating to the services themselves. For this reason, SLP provides authentication but no encryption of service advertisements. Service association, i.e., allocating a service resource, exchanging confidential access credentials, would have to be done in a second step, with a different protocol.

4.3.2.2 Jini

Jini [Sun01] is a framework for building (potentially) *referentially uncoupled*, distributed systems and has been developed by Sun Microsystems for the Java environment. [3] The main idea is to exploit the Java code mobility (and security) functions for distributed computing applications with a focus on service location and service association.

The key concept within Jini is that of a *service*. A Jini session is thought of as a federation of services that are brought together and coordinated in order to perform a common task. We have characterized Jini as a coordination system for *referentially uncoupled* entities because services can interoperate without having to know the addresses nor the interfaces of their communication peers. Both are discovered in a so-called *lookup* step.

Service Location and Association

Jini employs a central bootstrapping mechanism called *the lookup service* that is used by entities to locate services in the network. A network may provide one or more lookup services, and services (as well as clients) locate lookup services by sending multicast requests.

Services register with the lookup service and *upload* a *service object* and *service attributes* to the lookup service. The *service object* is an implementation of the service's interface, i.e., it provides the methods that users and applications will invoke to use the service. The *service*

[3] In principle, Jini is not tied to the Java programming language but to the Java application environment, i.e., it could be used with any programming language with a compiler that can compile object code for the Java virtual machine platform. It is needless to say that there are not many implementations of compilers for programming languages other than Java that can generate object-code for the Java virtual machine.

attributes provide a description of the service, i.e., its capabilities that can later be used by clients to locate and select the appropriate service.

Clients locate lookup services by sending multicast requests and can then query existing lookup services for a certain type of service by passing a *service template* to the lookup service that provides a filter for acceptable services and a set of required service attributes. The lookup service answers by sending a list of matching services that have registered. The client selects the appropriate service and uses its *service object* to deploy the service.

Jini provides the concept of *leases*, i.e., grants of guaranteed access over a time period. clients and services negotiate leases during the service association step: a client requests to use the service for a certain period of time and a service grants access for a certain period. If a client requires using the service for a longer time, it can *renew the lease*. If it does not explicitly renew its lease, the association is implicitly terminated and the service resources are freed and made available to other clients. The lease concept is essentially contributing to Jini's fault tolerance, because it helps to avoid unnecessary blocking of services in situations of network or host failures. Even if a client does not explicitly terminate the association, the service and the used resources can be freed automatically as soon as the lease expires.

Communication

Once a client has selected an appropriate service and negotiated a lease time, it can start to use the service through the interface of the *service object* that has been obtained from the lookup service. The Jini architecture does not specify how the communication between client and service is realized. For example, it may well be that there is no communication at all, because all the functionality is contained in the obtained service object.

Typically however, the service object represents an interface to remote procedures of a service's process that are invoked through the *Java Remote Method Invocation* (RMI) mechanism. Essentially, RMI is a Java extension to the RPC paradigm (see Section 4.1.1). RMI allows to invoke methods on objects that are passed along with the method invocation message, while the RMI message can also contain the code that is to be invoked at a remote system.

The service object can thus be viewed as an interface module that controls a process on a remote system using an extended RPC mechanism. The communication between service object and service process does not necessarily have to rely on RMI, however this is most often the case.

RPC-like control communication is not the only type of interaction model that is employed by Jini. Jini also provides event notifications that work as follows: an entity can register for a certain class of events at another entity and will subsequently receive corresponding events. This mechanism is essentially an extension of the JavaBeans event model to distributed applications and as such is a Java-specific mechanism.

In addition, Jini does also support *transactions* for coordinating state changes using a two-phase-commit protocol: entities that participate in a transaction send a first message to a *transaction manager* in order to indicate that they have completed their task of a certain transaction. The *transaction manager* then issues a commit request to each entity in order to trigger the entities to finally commit the transaction.

Summary

Jini is Sun Microsystem's ad-hoc communication solution for the Java environment that is targeted at client-server communication. It does not provide specific mechanisms for group communication, although group communication is not explicitly excluded either. [4]

The main idea is to dynamically locate services and learn about their interface by employing a service object that provides an implementation of the interface or can act as a proxy to a remote service implementation on a service providing process. One of Jini's noteworthy contribution is the *lease concept* that adds fault tolerance by avoiding service blocking in failure situations. However, although Jini is in principle language independent (it is an extension to the Java virtual machine functionality), it is essentially a Java-only mechanism and not used for other programming languages.

4.3.2.3 Universal Plug and Play

Universal Plug and Play (UPnP) is an architecture for device control in local networks and relies on different protocols for different functions such as device discovery, device description, transmission of control commands, and event notification. UPnP has been developed by Microsoft and is intended to provide peer-to-peer communication between different types of devices without the need for manual configuration, e.g., in home networks where devices from different vendors are employed and connected to an IP network in an ad-hoc fashion. To allow these different devices to interwork, it must be possible to not only locate devices but also to learn their capabilities dynamically and to exchange information without knowing in advance which devices are present. UPnP assumes a configured IP network, i.e., it is not targeted at configuring IP nodes with respect to their IP parameters. Mechanisms such as DHCP [RFC1541], IPv4 address auto-configuration [Cheshire02] or IPv6 stateless auto-configuration [RFC2462] are to be used for this purpose.

In the following, we will briefly describe the different protocols and mechanisms that are employed by UPnP for its different functions. Note that most of these protocols have not been published as formal specifications such as RFCs. In these cases we refer to working documents that can be downloaded from Microsoft at `http://www.microsoft.com/hwdev/tech/nonpc/upnp/`.

Device discovery: For device discovery, UPnP uses the *Simple Service Discovery Protocol* (SSDP) that relies on periodic unsolicited service advertisements from devices *and* on explicit service requests from clients (that are called *control points*). Devices periodically send service advertisements to a well-known multicast group and a well-known port. These advertisements contain information about the device type, an identifier, a URI for more information about the device and a duration for which the advertisement is valid. There is no mechanism in SSDP to calculate a suitable transmission interval for periodic advertisement in order to allow the protocol to scale to large numbers of devices. In addition to announcing the availability of a device, it can also be explicitly announced that a device is no longer available.

[4]There are additional mechanisms that are layered on Jini such as JavaSpaces, which provide distributed persistence and object exchange mechanisms that are targeted at group communication.

SSDP also allows a control point to search for devices of interest by multicasting a *search message* containing a service type specification. Devices that match the service type respond with a regular service advertisement message that is in this case *unicast* to the control point.

SSDP is based on a protocol that is called HTTPU/HTTPMU (*HTTP Unicast/Multicast over UDP*). Whereas the UPnP Device Architecture refers to HTTPU and HTTPMU as separate protocols, they are essentially one protocol that is used with unicast and multicast, with different rules for answering multicast requests. In the following, we will therefore refer to the protocol as *HTTPU* only. HTTPU allows for encapsulating HTTP messages in multicast and unicast UDP packets to be sent within a single administrative scope. Because the semantics of HTTPU significantly differ from HTTP, HTTPU merely represents a message transport protocol that uses the HTTP message syntax. Several integral functions of HTTP such as caching and proxying cannot be applied to HTTPU directly. In order to avoid response implosions for multicast requests, responding entities have to wait a random amount of time before answering the request (the maximum wait-time is specified in the request). However, there is no mechanism to avoid that all devices in the network eventually respond to a search message.

Moreover, SSDP itself does not use any HTTP method at all. Instead it relies on the *General Event Notification Architecture Base* (GENA) — a specification that defines a `SUBSCRIBE` and a `NOTIFY` method that are intended for subscribe/notify communication between peers. The specification claims to "provide for the ability to send and receive notifications using HTTP", whereas in fact it is a completely new protocol that re-uses the HTTP message syntax and the semantics of some HTTP status codes.

Device description: UPnP provides a description framework that allows control points to obtain detailed information about devices, e.g., general information such as the vendor and device name and their services, i.e., information about the control facilities that the device is offering for control over UPnP. In order to interact with a device, a control point retrieves the corresponding service description from the device that provides information about the supported control methods. The service description is retrieved in a second step, after the control point has discovered the device. In fact, the control point learns from the service advertisement where to obtain the service description.

A UPnP service description includes a list of actions that the service can perform, including parameters and possible results. In addition, variables of the service can be described that can be inspected and modified in a control session.

The service description is an XML document that conforms to a *UPnP Service Template* — a schema that is specified by the UPnP Forum for specific device types (allowing for potential vendor-specific extensions).

Description documents are obtained by regular HTTP interactions, i.e., a control point sends an HTTP-GET-request to a device and receives the description in the response message.

Device control: For controlling devices, UPnP provides two services: remote procedure calls and remote variable inspection and modification (that can be viewed as a specialization of remote procedure calls).

UPnP relies on the *Simple Object Access Protocol* (SOAP, [Box00]) for realizing these services. SOAP is a specification of how to use XML to represent remote procedure calls, including their parameters and responses, in XML documents that are transmitted using HTTP. (In fact, SOAP can in theory be used with other transport protocols, but HTTP is most commonly used.) SOAP provides the concept of a message envelope — a message structure consisting of a header and a body. The header can be used to specify attributes such as who should process the message and how to process it. The body contains the actual elements that represent the actions that are to be executed by the receiver. SOAP also specifies the representation of data types such as integers, strings and compound data types.

In SOAP, the actions that are described in the body of a SOAP message are application specific, i.e., in the case of UPnP there is a UPnP-specific XML-schema that defines element types such as `actionName` and `QueryStateVariable`. UPnP control points can thus send SOAP messages to devices requesting the execution of remote procedures that they have learned of before from the service description they have obtained from a device. The SOAP-binding for HTTP (the only one that has found significant deployment) specifies the use of the HTTP-POST method for sending SOAP requests to a service and the use of regular HTTP-response-messages for SOAP-responses. In both cases, the content-type is `text/xml`.

SOAP-1.1 itself does not define any security mechanisms that could be used to authenticate control points and to provide integrity and confidentiality of the communication, although HTTP security mechanisms could in theory be used. UPnP does not specify any security mechanisms either.

Event notification: UPnP allows control points to subscribe to certain state variables of a service that have been described in a service description before. For this purpose, UPnP employs the same *General Event Notification Architecture Base* (GENA) that is employed for service discovery: GENA specifies the methods `SUBSCRIBE`, `UNSUBSCRIBE` and `NOTIFY` that are used for realizing the subscribe/notify communication.

Presentation: In addition to control a service, a control point can also present the status of a service to a user. UPnP provides the possibility that services include a service URI in their service description that are later traversed by control points using regular HTTP to retrieve corresponding presentation resources, e.g., an HTML document.

UPnP is problematic with respect to scalability and security. Compared to SLP that carefully addresses the issue of scalability by minimizing the use of multicast messages and by other mechanisms such as the previous responders list in multicast requests, UPnP has obvious deficits: The service advertisement rules do not allow UPnP to scale to larger environments with many services. In addition, the multicast service requests can lead to response implosions.

The control features of UPnP are completely insecure. There is no authentication of control points, no message integrity and no confidentiality. This makes UPnP unusable except for controlled environments. For control communication, UPnP relies on two mechanisms only: RPC-like remote actions and event notification. There is no support for group communication, e.g., dissemination of information in a group, group coordination, and agreement procedures.

However, group communication appears to be a natural communication paradigm for the applications that UPnP is intended for such as coordination of devices in a home network.

The use of HTTP for service discovery, event notification and remote method invocation is also problematic: SSDP and GENA have nothing in common with HTTP except for the general message format. The usage of HTTP for SOAP is questionable, since many of HTTP's features such as caching, proxying, and authentication are not applicable to SOAP anyway. The only reason for using HTTP seems to be to facilitate firewall traversal for SOAP messages, which is a wrong approach in itself as it tries to secretly undermine security policies instead of addressing them.

In addition, the specification of UPnP cannot be called complete. Some components such as SOAP have been standardized by standard bodies. Other specifications such as SSDP, GENA and HTTPU are only provided as expired Internet-Drafts and are in rather premature state, which aggravates the realization of interoperable implementations.

4.3.3 Lessons Learned

We have analyzed a variety protocols for coordination-based communication that differ in their application scope and provided functionality:

- TIBCO Rendezvous is representing a class of distributed communication frameworks that focus on coordination by flexible message distribution for a group of referentially uncoupled communication peers.

  In Section 4.3.1, we have seen how mechanisms such as *subject-based addressing* allow implementing group communication between referentially uncoupled groups. The loose coupling between communication peers is also reflected by the concept of *self-describing messages*, which allow for interoperation without requiring static bindings of interfaces as it would be required for RPC communication (Section 4.1.1).

  Consequently, distributed message-oriented, coordination-based systems fall into another category than CATOCS-based approaches such as ISIS and its successors. They are less suited for applications that have strong requirements on reliability and message ordering constraints for group communication but are rather targeted at *information dissemination* scenarios, e.g., applications such as stock tickers and corporate news distribution systems.

- Distributed systems for *service* coordination represent another class of distributed coordination-based systems. For service coordination, we have identified the need to *locate* services in a network and to *associate* with services before the actual coordination session commences.

  The Service Location Protocol that we have discussed in Section 4.3.2.1 provides an acknowledged architecture for the discovery of services within an enterprise network. One of its interesting features is the bandwidth-efficiency and scalability. SLP abstains from periodic service announcements and tries to minimize the use of multicast communication as far as possible. In addition, it provides authentication of services and a flexible mechanism to query for services based on service filter predicates. SLP does not address the association and coordination services.

- Java's Jini extension that we have described in Section 4.3.2.2 goes a step further and provides mechanisms for service location and service association. The key idea is to provide clients with a *Jini service object* that encapsulates the functionality of the service and possibly the communication mechanisms for using the service. This association is Java-RMI-based, i.e., relies on a Jave-centric extension of the RPC paradigm. One interesting feature of Jini is the lease concept that allows for robust service association and release procedure even in the presence of host or network failures. The actual communication between clients and services within a Jini federation is not specified by the architecture specification, however, in most cases RMI can be expected to be deployed.

- UPnP is a service coordination architecture for controlling service providing entities in a local network, e.g., a home network. It provides individual protocols for service discovery, service association and coordination and for event notification. The service discovery relies on periodic service announcements, however without address scalability concerns. UPnP addresses the issue of bringing entities together that are not known to each other by providing the concept of service descriptions that contain so-called service templates, i.e., an interface description that relies on predefined templates for specific device types.

  Device control is implemented by the use of SOAP, an HTTP-based protocol that follows the RPC-paradigm and does thus support point-to-point communication only. Event notification is realized by relying on a subscribe/notify scheme, and events can be multicast to a group of subscribers. For all UPnP protocols, the communication does not provide any security mechanisms at all, which makes UPnP difficult to use except for controlled environments.

For our design of the Mbus framework, we have adopted some of the concepts of the described coordination and ad-hoc communication approaches:

- the TIBCO Rendezvous concept of subject-based addressing and receiver-based filtering is an interesting way to allow for flexible group communication in distributed systems;

- the self-describing message format provided by TIBCO Rendezvous enables interoperation without requiring static interface bindings;

- in the analysis of service location and service coordination frameworks, we have identified different phases for service coordination: service location, service association and service coordination itself;

- for service discovery, multicast communication can be employed in different ways, i.e., either for multicast service announcements (as provided by Jini and UPnP for service discovery in ad-hoc scenarios) or for multicast queries (as provided by SLP for scalable enterprise service discovery); and

- in ad-hoc communication scenarios, concepts such as *service leases* as provided by Jini can enhance robustness and simplify the service dissociation.

4.4 Component Technologies

In Section 1.3, we have presented a simplified model of a multimedia conferencing endpoint and showed how it is decomposed into different *components*. In this section, we will discuss the term *component* more deeply, describe the component-based approach for developing software and present existing component technologies.

The production and deployment of components is a characteristic of many mature engineering fields, such as mechanical and electrical engineering. It is usually a precondition to mass deployment because it enables the easy re-use of existing building blocks instead of the development of new solutions and thus helps to limit production costs significantly. Requirements for the deployment of component technologies are standardized interfaces, the establishment of recognized architectures and emerged *best current practices* to solve particular problems in a certain domain.

The goal is to be able to construct an application from readily available components from possibly different sources *without* requiring manual adaptations. In the component based systems domain, the term *commercial off the shelf* (COTS) deployment is used to refer to the usage of readily available components as elements of a larger systems, where the components do not require any form of manual adaptation or configuration in order to be used.

Although re-use, abstraction and modularization have been design goals in the software industry for quite a long time, the component-based development of software has only recently been able to obtain acceptance and to play a role for the main-stream software development. Though many technical foundations have been developed earlier, the first noteworthy deployment of software components in terms of commercial importance is often considered to be the usage of *Visual Basic custom controls* (VBX) in Microsoft's *Visual Basic* programming environment that has been released in 1992. The *Visual Basic* programming environment is based on the BASIC programming language and provides a graphical environment to compose a software system by using the desktop *drag and drop* metaphor: Users can re-use components by integrating them graphically into their own applications without the need for manual adaptation and integration work. The VBX technology has later evolved into Microsoft's *ActiveX* technology.

When we try to define the term *software component* we can identify the following main characteristics that Szyperski describes more deeply in [Szyperski99]:

A component is a unit of independent deployment: In order to be re-usable in many different contexts, a component should not depend on a particular environment and should not require other components to be useful. The component represents a unit, i.e., it is not deployed partially but as a whole.

A component is a unit of third-party composition: A component must be re-usable without the need to know construction or implementation details. That means, the component has to be sufficiently self-contained and it must be possible to deploy the component by the use of well-defined interfaces.

A component has no persistent state: A component can be used by activating it in a particular system but it is usually not copied and used in multiple instances. Instead, a component may provide abstraction such as classes that may be instantiated multiple times.

It is important to note that the concept *component* is orthogonal to the concepts *object* and *distribution*. Components may rely on object-orientation and provide a set of classes that can be

instantiated into objects. However, a component could as well be implemented in a functional programming language or simply contain a list of procedures that can be accessed via some well-defined interface. Moreover, components can interface with their environment in many different ways: They can be integrated into a process's program space, e.g., by dynamically loading a shared library, and provide a list of subroutines and variables that can be accessed from an application program. Alternatively, components may run as separate processes and may be accessed using a network protocol, e.g., a component may be used as an entity in a distributed application.

In the following section, we will present some established component technologies: Microsoft's COM/DCOM framework, Sun's JavaBeans technology for the Java programming environment, the Tcl scripting language approach, and CORBA.

4.4.1 Component Object Model (COM)

The *Component Object Model* (COM, [Microsoft95]) is Microsoft's component architecture for the Windows family of operating systems. COM is programming language and even programming paradigm independent and is merely a *binary standard* for interfaces of components in a system. This means COM specifies how a component's interface is represented in the form of a table of function pointers. Clients can learn the interface of a given component by calling the common function `QueryInterface` that has to be provided by every COM object.

COM components can reuse other COM components by *containment* (a component *owns* another component) or by *aggregation* (a component includes another component but makes it visible to the outside). In addition, COM also allows deriving COM interfaces from other COM interfaces using single inheritance. COM provides a *versioning concept*, i.e., a component can provide multiple versions of an interface, which can also be queried for by a client. Component interfaces are usually specified using a COM-specific interface definition language that can be used as a basis to generate client stubs.

Summarizing, COM is a binary interface standard for components of Windows-based systems. Distributed COM (DCOM) builds on the basic COM concepts and allows building distributed component-based systems, relying on an RPC-like mechanism that includes a transformation of values into a platform-independent representation. Similar to RPC implementations (see Section 4.1.1), DCOM relies on the concept of client-side proxy objects and server side stub objects. Before a client can use a component on a remote system, a *binding process* must be performed that couples exported COM objects on a server to client stubs on a client system.

COM and DCOM are component frameworks that rely on the concept of *binary interface specifications*. As such they are to some degree platform respectively operating system dependent. In fact COM/DCOM is essentially only used for Windows-based systems. For the distributed DCOM case, RPC-like communication is deployed to implement inter-component communication, which includes a tight (temporal and referential) coupling of entities.

4.4.2 JavaBeans

JavaBeans [Sun97] are components for the Java environment. A JavaBean is essentially a collection of classes that perform a certain functionality and can be re-used by application developers. JavaBeans provide the concept of *dual usage behavior*, i.e., the component can be used at *design time*, e.g., when they are composed using an application builder that might provide

graphical application composition, and they can be used at *run-time*, i.e., when the application is executed. JavaBeans support the notion of *interface inspection*, i.e., an application or an application builder tool can inspect the provided interface of a JavaBean in order to learn how to use it, which removed the need for formal interface descriptions.

JavaBeans are intended to support an event-based interaction model with its environment, i.e., the application that deploys the JavaBean. JavaBeans can declare that their instances would generate certain events or listen to certain events, and when the component is embedded into an application context, the Java event mechanisms is used to connect the component's event sources and listeners to those of the application. JavaBeans also provide a *property concept*: A bean can specify a number of properties of arbitrary types. Properties are simply attributes that can be accessed (obtaining or modifying the attribute value) by standardized method names. Properties are usually persistent attributes of a bean instance, such as the background color that is to be used to display a graphical bean.

Summarizing, we can state that JavaBeans provide some interesting concept such as the abstractions for *properties* and *events* that seem to be useful for deploying components in applications, especially in conjunction with the *interface inspection* facility that allows clients to learn the interface, the provided properties and the event notification behavior of a component dynamically, i.e., at design time or at run-time. These functions are in general realized by deploying standard Java mechanisms and as such obviously limited to Java environments. JavaBeans do not provide mechanisms for distributed components.

There is a component-based *distributed* computing approach for Java that is called *Enterprise Java Beans* (EJB, [Sun02]) and that is intended for the development and deployment of component-based distributed business applications, e.g., access to corporate databases. The approach has similarities to CORBA that we describe in Section 4.4.4 (and can inter-operate with CORBA systems), therefore we will not describe EJB here in detail.

4.4.3 Scripting Languages (Tcl)

A completely different approach that does per se not qualify as a component architecture is the *Tcl/Tk* approach of "gluing" components together by the use of the Tcl scripting language [Ousterhout94]. Tcl is a simple scripting language that has been explicitly designed for allowing linking a Tcl interpreter to a binary program, where a Tcl program could be used to orchestrate a set of other binary modules.

In this way, object code modules (that can be generated from source code written in different programming languages) can be combined to a single application and a Tcl script could be written that interfaces and interacts with each module. For this purpose, an interface of each module's exported functions must be created that can be used inside the Tcl interpreter. This interface generation is typically automated by the use of programming language specific interface generators.

We mention this approach here because many of the aforementioned Mbone tools such as vat, vic, and rat are implemented by integrating modules that have been written in C or C++ with a Tcl interpreter into a coherent application. Tk is a GUI toolkit that has bindings to other languages besides Tcl as well, but is mostly used with Tcl in order to provide an easy way to program graphical user interfaces in Tcl scripts. The GUIs of the mentioned Mbone tools, for example, have been programmed in Tcl/Tk.

The way that components are integrated is quite different from other approaches, as there are no possibilities for querying interfaces. To some extent, the automatic generation of Tcl interfaces from source code (e.g., in C++) by the use of interface generators can be compared to an interface description approach, however it is certainly more limited. The usage of components still requires a significant amount of manual work by a programmer.

As an integration platform of object code modules, Tcl/Tk does obviously not qualify as a distributed component-based approach. Interestingly enough, however, there are some approaches that try to leverage the possibility of remote-controlling Tcl/Tk applications by the *Tk-send-mechanism* — a mechanism that does allow an application to send arbitrary Tcl expressions to another application that are then interpreted. Some local coordination systems such as PMM (see Section 4.2.2) do even provide the concept of coordinating applications by sending Tcl-commands that could directly be passed to a corresponding interpreter.

We have to state that both approaches, the Tk-send approach and the PMM approach are not adequate for a serious coordination mechanism as they provide too little abstraction, are dependent on the existence of Tcl interpreters at all applications and can impose security problems. These problems have been one motivation for the development of more adequate solutions, such as CCCP (see Section 2.2.2.2) and the Mbus.

4.4.4 CORBA

The *Common Object Request Broker Architecture* (CORBA, [OMG03d]) has initially been designed as an architecture for *distributed* object-oriented systems, i.e., distributed systems in which entities communicate using object-oriented interfaces. The motivation of this approach was two-fold: For some programming languages such as C++, it is (still) a problem to integrate modules that are written in the same programming language and are compiled for the same platform but have been compiled by different compilers due to incompatible binary object standards. The problem gets worse when different programming languages (that largely support the same concepts) are used, because in general each programming language relies on a unique object model. Another problem is that RPC implementations (*nomen est omen*) are addressing *procedural* communication, i.e., they do not directly support features that are usually expected from object-oriented programming environments such as object state persistence, encapsulation, inheritance and polymorphism.

These problems of *binary incompatibility* and the need for a distributed computing mechanism with the support for object-oriented programming support have led to the development of an architecture that strictly relies on carefully standardized interfaces and on standardized, programming language and platform-independent communication mechanisms between entities in a distributed system. Unlike COM/DCOM (see Section 4.4.1), CORBA does not rely on binary standards for interfaces, but relies on a generalized representation of objects and their interfaces. The communication between objects in a CORBA system relies on so called *object request brokers* (ORBs) that are essentially implementations of the CORBA remote method invocation protocol. CORBA defines different so called *inter-ORB protocols*, the most common one being the *Internet inter-ORB protocol* (IIOP), which uses TCP for the transmission of requests and responses.

CORBA however not only addresses the basic interoperability issues for remote object method invocation but has evolved into a so called *Object Management Architecture* (OMA) that is targeted at the development of large-scale (“enterprise”) distributed object-oriented sys-

tems. We describe the basic remote method invocation technology in Section 4.4.4.1 and the OMA in Section 4.4.4.2.

4.4.4.1 Remote Method Invocation Service

The ORBs are the link between server objects and application code on clients. Method invocations on a client system that refer to objects hosted on a remote system are passed from the local method invocation interface to a local ORB. The local ORB locates the receiver object and the invoked methods and transports the invocation's arguments to a remote ORB. The remote ORB invokes the requested method on the receiver object.

Similar to RPC communication, the invocation of remote methods involves the *marshaling* into platform-independent representations. In addition, all programming languages that are to be used for a CORBA application must have *bindings* to the common language, i.e., a mapping from the common CORBA language to a specific programming language. Again, similar to RPC-based systems, CORBA provides the concept of *client-stubs* (client-proxies that provide the same interface as a given server object and forward method invocations via the ORB to the server object) and *server-stubs* (server-interfaces that receive requests for method invocations, unmarshal the arguments and invoke the method on the "real" server object). In CORBA, client-stubs are simply called *stubs* and server-stubs are called *skeletons*.

In order to facilitate the construction of these programming-language and ORB-implementation specific stubs and skeletons, CORBA provides (again similar to RPC implementations) the concept of a generalized *interface definition language* (IDL). IDL is the standardized common language in CORBA that is used to specify the *interfaces* of modules. An IDL specification can be compiled by an IDL compiler to platform-specific, language-specific and ORB-implementation-specific stubs and skeletons. In addition, the interface description is deposited in an ORB's interface repository, where clients can request it later.

Relying on stubs and skeletons that are generated from IDL descriptions leads to a *static binding* of interfaces, i.e., client programs must know the interface of server objects in advance in order to invoke methods on them. In order to allow for a more dynamic form of remote method invocation, CORBA also provides a *dynamic method invocation interface* (DII) and a *dynamic skeleton interface* (DSI). DII and DSI allow for the dynamic selection of methods at a server or a client.

4.4.4.2 Object Management Architecture

In addition to the basic remote method invocation service that is specified by the ORB architecture, CORBA has also been extended to so called *enterprise distributed computing* by the definition of new services and facilities.

A set of so called *CORBAServices* have been defined that are intended to support the development of CORBA-based distributed applications by providing a set of application-independent, commonly used interfaces and objects. For example, the CORBAServices include a *Object Transaction Service* (OTS) that allows application developers to include transactional behavior in their applications without having to implement the ACID properties themselves. Other examples of CORBAServices are:

- the event notification service for generalizing event notification communication;

- the concurrency service that provides mechanisms for synchronizing access to objects; and
- the naming service that allows to associate object identifiers to names in a hierarchically organized namespace.

A complete list of CORBAServices is available at [OMG03a].

CORBAFacilities are essentially services that are not as application-independent and fundamental as CORBAServices. [OMG03b] provides a list of currently defined CORBAFacilities.

The set of CORBA specifications has been augmented by a specification of a *CORBA Component Model* [OMG03c]. A CORBA component is a named set of features with described features and standardized interaction mechanisms, similar to a *JavaBean* (see Section 4.4.2). In fact, the CORBA component model has been aligned with the *Enterprise JavaBeans Specification* (that we have mentioned in Section 4.4.2). The idea is to provide a way to access remote components, relying on standard CORBA mechanisms, i.e., formal interface descriptions in IDL, using IIOP communication.

4.4.4.3 Summary

We have described how CORBA has evolved from a solution for the development of object-oriented distributed systems with a focus on platform and programming language independence into an "enterprise" distributed application framework. CORBA is an ambitious approach to provide a standardized framework for all sorts of distributed enterprise applications, ranging from remote database access to dissemination of events and other information. The basic remote method invocation protocol is essentially based on the RPC paradigm; the IIOP is based on TCP.

The extension of CORBA to an enterprise application framework has significantly contributed to its complexity, although the fundamental model is fairly simple. CORBA is thus rather targeted at the development of timely and referentially coupled "large-scale" applications and not for the coordination of a loosely coupled group of application components.

4.4.5 Lessons Learned

In the previous sections, we have discussed approaches for supporting the development of component-based systems, i.e., systems that do not provide a monolithic structure but are arranged from a set of independent, re-usable modules. The purpose of this discussion is to

- provide a definition of the terms *component* and *component-based system*; and to
- analyze typical interaction schemes that are provided by component frameworks and investigate their applicability for the Mbus coordination framework that we describe in this thesis.

The key concept that characterizes a *component* (that we have adopted from Szyperski [Szyperski99]) is that it represents a unit of independent deployment that is not tied to a specific application, but can be deployed by an application programmer in different environments and for different applications. The goal is to be able to construct an application from readily available components from possibly different sources *without* requiring manual adaptations.

We have presented different acknowledged component frameworks and investigated different techniques that are intended for fulfilling this fundamental requirement.

The COM/DCOM approach (Section 4.4.1) is a representative for component frameworks with *binary interface standards*. A COM component has a defined layout in memory that can be used by clients to learn the interface of the component. DCOM extends COM to distributed systems, whereas the communication between clients and components relies on RPC-like communication, which includes the use of client-side proxies (*stubs*). COM/DCOM is essentially only used on Windows platforms.

JavaBeans (Section 4.4.2) provide a far-reaching formalization of components through standardized Java interfaces, e.g., through the *property* abstraction, through the *event* mechanism and by allowing for the dynamic *inspection of interfaces*. The JavaBeans mechanism (hence the name) is tied to the Java environment.

While COM/DCOM and JavaBeans are component frameworks that are specific to a certain platform or computing environment, CORBA is strictly platform-independent and programming language independent, i.e., it allows for the composition of applications from components that can reside on different platforms and enables the communication between components and applications regardless of the programming language that was used to implement a specific component (provided an ORB implementation and corresponding language bindings exist for a given platform). The enabling technology for this platform-independence is a formal interface description, i.e., the CORBA IDL. Again, the interaction between distributed components relies on RPC-like communication. CORBA itself has become quite an ambitious and complex technology because it has been extended to support “large-scale” distributed applications, such as database transactions, and is thus going beyond the scope of component frameworks. One of these extensions is indeed a component model that is aligned with the Enterprise JavaBeans technology and provides common interfaces and characteristics for components.

The scripting language approach that we presented in Section 4.4.3, using Tcl/Tk as a representative technology, is not a proper component framework as the other approaches we have described. However, it can be used to “glue” different binary components together to a standalone application, in principle also relying on interface descriptions. In this case the interface is defined by the programming language specific interface, e.g., a list of C++ function signatures that is made visible in the Tcl/Tk environment, often by the use of interface generators.

We have learned from this discussion that component frameworks typically provide the following three characteristics:

1. The main characteristic is the existence of a well-defined interface. This interface can either be static, as in the Tcl/Tk case and in the simpler CORBA and COM/DCOM cases, but it can also be dynamically queried by relying on *interface inspection techniques*.

2. Another insight is that in many frameworks, applications tend to interact with components using certain standardized communication patterns, e.g., examination and modification of properties and event-notifications.

3. A third observation is that for distributed component frameworks, the communication between components is almost always implemented with RPC-like communication. All communication abstractions are layered on top of the fundamental RPC communication facility.

4.5 Summary

In this chapter, we have looked at four different areas: distributed systems (Section 4.1), local coordination for conferencing systems (Section 4.2), coordination and ad-hoc communication (Section 4.3) and component technologies (Section 4.4). The purpose of this analysis was to describe fundamental concepts such group communication in distributed systems and component-based application development, but also to document the state-of-the-art and to identify gaps with respect to our requirements described in Section 3.2. The discussion on distributed systems has led to the following insights:

- The popular and acknowledged RPC paradigm is useful for applications where synchronous communication and static bindings of communication principals fit the application semantics. It is less appropriate in situations where static bindings can either not be established or are not desirable (ad-hoc communication). It is also not appropriate for communication scenarios that require a high degree of concurrency and are thus better off with asynchronous communication.

- The ISIS-class systems are intended for applications that impose very strong requirements on reliability, consistency (in all phases of a session) and fault-tolerance. Examples of such applications are distributed stock-order systems or the coordination of flight-control workstations.

  Fault-tolerant group communication systems such as ISIS and its successors try to overcome the performance problem that is caused by the synchronous RPC communication style. They provide protocols that allow splitting up an application into a group of processes that can inhibit a much higher degree of concurrency because it does not rely on synchronous request/response communication anymore.

  In order to achieve consistency in such group communication systems, fault-tolerant transport and message-ordering mechanisms must be provided. The ISIS-class systems provide these services as part of their group communication transport layer for message passing, so that application developers do not have to provide the corresponding functionality themselves.

- It is hard to design corresponding toolkits, i.e., protocol implementations, efficiently. The early monolithic ISIS approach has proven to be problematic and has led to implementations that were difficult to maintain. The successor systems Horus and Ensemble provide a more modular approach, that caused however other problems: Multi-layer approaches tend to be inefficient at run-time due to multiple encapsulation and the corresponding processing overhead which may be a problem on less powerful machines. Manual optimizations tend to be error-prone; therefore automatic optimization approaches have been developed.

- The protocols for message-ordering and corresponding services assume an underlying reliable transport mechanism. If this cannot be provided, efficiency problems occur.

The local coordination protocols that we have described in Section 4.2 are limited in scope and consequently notably simple. Some of them have not been designed with a broader conference control architecture in mind. Although they illustrate basic concepts for the coordination

of application entities, we have stated that both the LBL Conference Bus and PMM do not fulfill our requirements. In addition, both approaches are explicitly designed for the coordination within conferencing systems and thus not application-independent.

In order to investigate fundamental concepts for the coordination of application entities in decomposed, distributed applications, we have analyzed some existing work from the coordination and ad-hoc communication domain and from the component-based systems domain.

In the discussion of dedicated protocols for coordination and ad-hoc communication in Section 4.3, we have learned some design principles of protocols that are intended to be used for applications that cannot provide or do not require a tight coupling between entities. All protocols and frameworks we have presented rely on a *rendezvous mechanism*, which is in general implemented by IP multicast. This rendezvous mechanism is required to establish some form of binding and a common communication channel between typically referentially uncoupled entities.

One interesting communication mechanism for the coordination of a group of entities is the *subject-based-addressing* scheme that is deployed by the Rendezvous system and allows for inter-operation without requiring static bindings of interfaces. It is accompanied by the concept of *self-describing* messages that can be interpreted without requiring static interface definitions.

We have seen that this communication model leads to a different programming model: There is no static binding between protocol messages and functions at a receiver and there is obviously no synchronous communication. Instead an *event-based* model has shown to be more appropriate. In this model, the receiver associates a set of event handler functions with messages that match a subject specification, and these event handlers are called asynchronously upon receiving corresponding messages.

Communication systems such as Rendezvous are often used for implementing a message-oriented communication channel in a well-defined environment, e.g., in a corporate network. For the establishment of communication sessions, service-coordination-oriented protocols and architectures are commonly used. These protocols do not rely on a pre-established configuration and are designed to accommodate scenarios where the type and number of available communication peers is not known in advance but must be determined dynamically.

Consequently, protocols such as SLP, Jini and UPnP provide the concept of *service discovery*, a process where communication peers locate each other, learn about their capabilities and offered services and associate with each other in order to establish a common coordination session. SLP is strictly limited to the service location and capability description step, whereas Jini and UPnP use a location mechanism in order to *bootstrap* a communication session. In this communication session, mechanisms such as interface descriptions are used to provide means to interact with services that are not known in advance. Different distributed communication mechanisms are used, including RPC communication and event notification (potentially based on group communication mechanisms).

Most of these protocols are not explicitly targeted at supporting the development and deployment of distributed *component based systems*. In Section 4.4, we have first defined the terms *component* and *component based system*. A component represents a unit of independent deployment that is not tied to a specific application, but can be deployed by an application programmer in different environments and for different applications. We have noted the three main characteristics of component frameworks:

1. the existence of a well-defined component interface;

2. the usage of standardized communication patterns, such as examination and modification of properties and event-notifications; and

3. the usage of RPC-like communication for distributed component frameworks.

Many component frameworks that we have analyzed are targeted at a specific platform or programming environment, such as COM/DCOM and JavaBeans. The CORBA approach (and the Enterprise JavaBeans approach) have emerged into frameworks for large-scale distributed systems, i.e., as substitutions for traditional client-server computing in enterprise environments and are less often used for the development of component-based applications that fulfill a specific purpose for user, e.g., a component based conferencing system. However, there are also exceptions such as the light-weight CORBA-2.4 implementation ORBit[5] that is used for the GNOME desktop environment.[6]

In the following Chapter 5, *Architecture*, we will describe the overall architecture of our design of a coordination framework that is intended as a basis for the development of distributed, component-based systems, where application components can be federated in an ad-hoc fashion and the coordination framework itself is not tied to a specific application.

[5] `http://orbit-resource.sourceforge.net/`

[6] `http://www.gnome.org/`

Chapter 5

Architecture

Based on our discussion of use cases and requirements in Chapter 3, *Use Cases and Requirements* and our analysis of foundations and related work in Chapter 4, *Foundations and Related Work*, we have developed an architecture for a local coordination framework that is targeted at coordinating application entities of component-based systems. Although we have applied the framework to the coordination of decomposed multimedia conferencing endpoints and gateways, which we describe in Chapter 9, *Mbus in Conferencing Systems* and in Chapter 10, *Mbus in Projects*, we are relying on a strict separation of application-independent protocol mechanisms and application-specific semantics. Hence, the coordination framework is applicable to different local coordination scenarios.

The fundamental motivation for developing such a coordination framework is to enhance component re-use in decomposed applications — moving away from monolithic systems and from proprietary, system-dependent coordination mechanisms towards a generalized approach for integrating independent modules into application contexts. We have therefore analyzed existing frameworks for component-based systems in Section 4.4 and described their main fundamental properties. In our design, application components are instantiated as independent processes that attach to an Mbus coordination session pertaining to a certain application context.

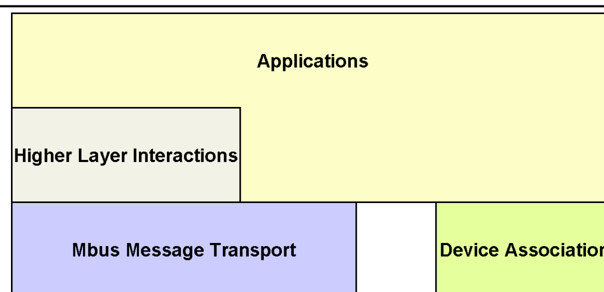

Figure 5.1: Mbus Protocol Layers

Figure 5.1 depicts a conceptual layering of the Mbus framework. The bottommost layer provides the fundamental group communication messaging service offering basic, asynchronous message exchange in a group of application entities with minimal support for reliable message transport. The message transport layer does not provide different interaction types such as RPCs

and event notifications, instead, these services are provided by a separate layer that is based on the fundamental messaging service.

We have defined an *Mbus session concept*, whereas session admission can either be accomplished by sharing static configurations, which is appropriate for locally distributed applications, or by employing the *Dynamic Device Association* service, which provides the possibility to establish Mbus sessions dynamically. On top of these services, application semantics can be defined. In Section 6.3, we describe so-called *Mbus application profile*, i.e., messages and semantics for specific applications, which are typically defined in terms of higher layer interaction abstractions.

One such application profile is the *Mbus Call Control Profile* (described in Section 9.3), which is a set of Mbus messages for the coordination of call control engines in a distributed conferencing system. The *Call Control Profile* is part of our Mbus-based conferencing endpoint architecture (described in Section 9.1), which is a model for decomposing endpoint (and gateway systems) into independent re-usable components focusing on generality and extensibility.

In the following sections, we briefly describe the main components of this architecture focusing on the most important concepts. In Chapter 6, *The Mbus Framework*, we present the Mbus framework and its specific mechanisms in more detail.

5.1 Basic Message Transport

The Mbus messaging layer provides group and point-to-point messaging between so-called *Mbus entities* — members of an Mbus session. Based on network layer multicast services, the Mbus transport layer establishes an Mbus group communication service that relies on application layer addressing of entities and groups within an Mbus session.

The addressing concept relies on a variant of the *subject-based addressing* concept as described in Section 4.3.1 and also employs a *receiver-based-filtering mechanism* for facilitating the exchange of Mbus multicast messages between referentially uncoupled entities. The Mbus transport service also employs the concept of *self-describing* messages to allow coordination in scenarios where a static binding of interfaces is not feasible. The messaging service relies on a completely *asynchronous* model, and the fundamental transport layer does not support higher layer abstractions such as RPCs nor a virtual synchrony service for group communication. The group communication service relies on a best-effort model and the point-to-point communication service provides an optional reliable transport mode that relies on a message acknowledgment mechanism and provides per sender FIFO ordering. Unlike other local coordination protocols, the Mbus messaging service provides a mandatory security architecture that provides message authentication and message encryption.

5.2 Higher Layer Interactions

Based on the fundamental messaging service, higher layer interaction protocols have been defined. In essence, we have developed protocols for those interactions that we also identified as the minimum requirement for component frameworks such as JavaBeans and for coordination frameworks such as UPnP: RPCs, event notifications and property inspection/modification. In

addition, the higher layer mechanisms provide a *control relationship concept*, which formalizes control communication within coordination sessions, e.g., by defining different types of control relationships and abstractions for registering a controlling entity with a service providing entity.

5.3 Device Association

The device association service allows for establishing Mbus sessions in "dynamic" environments, e.g., for integrating entities into an existing Mbus session. With the architectures and services of service location and device coordination frameworks in mind, our device association service provides mechanisms for *service discovery* and *service association* and fulfills the security and scalability requirements we have mentioned in Section 3.2.

5.4 Local Conference Control Architecture

Our local conference control architecture is based on the Mbus framework as a local coordination mechanism and provides *internal management services* for conferencing systems. In such an Mbus-based conferencing system, there are entities of certain types, e.g., application entities such as audio engines, call control engines such as SIP engines and controlling entities that coordinate the whole Mbus application on behalf of a user.

Using Mbus communication mechanisms, we have developed the notion of *generalized controllers* that can coordinate these entities regardless of their specific type. With standardized interfaces and Mbus addressing mechanisms, a controller can dynamically detect the set of available application entities and coordinate them for conducting multimedia conferences. The conferencing architecture is described in detailed in Chapter 9, *Mbus in Conferencing Systems*.

Chapter 6

The Mbus Framework

This chapter provides a detailed description of the *Message Bus* protocol. Section 6.1 provides an overview of the Mbus protocol and Section 6.2 contains the in-depth protocol description. A higher layer of interaction schemes that can be used with the basic Mbus transport services is presented in Section 6.3. In Section 6.4, we describe an Mbus bootstrapping procedure called *Dynamic Device Association* that allows to set up Mbus sessions securely without prior manual distribution of session parameters, and in Section 6.5, we present extensions to the Mbus and the Dynamic Device Association protocol for ad-hoc communication scenarios.

6.1 Design Overview

Based on our requirements and the overall architecture we have described in Chapter 5, *Architecture*, we have we have formulated the following design goals for the Mbus transport service:

Protocol framework for component-based systems: The Mbus protocol is designed to serve as a platform-independent and programming-language independent framework for component-based systems with a coarse-grained granularity. The framework should provide mechanisms for independent deployment of modules, and, in the light of Szyperski's characterization provided in [Szyperski99], the framework is intended for the coordination of component *instances*.

For the Mbus framework, we view a component as an entity with a well-defined message-based interface that will typically run in its own execution thread, e.g., as an independent process. In this model, components are not implemented as binary modules that are linked to a single program, instead, modules can be implemented in different programming languages and run on different platforms.

As a consequence, we focus on the *distribution* of components: The Mbus framework is designed to support distributed component-based systems that may be distributed on a local network, while still allowing for the composition of local, single-process systems.

The baseline framework does not provide some features that are typically found in component-based systems such as *inspection*, *reflection* and *naming services*; however, the Mbus framework can be used to implement these services, as we demonstrate in Section 10.4.

Group communication: Since the design of the Mbus framework explicitly emphasizes the *distribution* of components, the main element of the framework is the *Mbus protocol* — a message-oriented protocol that provides the fundamental communication services.

In a distributed system that encompasses more than two entities, RPC-based mechanisms are generally inadequate, as we have discussed in Section 4.1. In order to allow for a higher degree of concurrency and robustness, the Mbus protocol provides group communication mechanisms, i.e., one-to-many communication, as one of its main features.

We have seen in our requirements discussion in Section 3.2 that scalability and bandwidth efficiency is an issue, even for local (group) communication, e.g., when we consider larger groups and low-bandwidth wireless links. Hence, scalability with respect to the number of session members is an important feature that we address by a decentralized design and by the direct use of multi-point communication features of the underlying network layers, namely IP multicast, in order to avoid unnecessary duplication of messages where possible. In addition, IP multicast serves as a rendezvous mechanism enabling entities to locate each other without the need for exchanging transport addresses prior to session establishment. Moreover, we follow a *receiver-driven* design for message filtering and delivery, thus facilitating the use of group communication and the creation of sub-groups.

In order to avoid the complexity and performance-penalties resulting from group protocols that provide services such as the complete *virtual synchrony* as presented in Section 4.1.2, we separate the protocol into a basic multi-point communication service and higher layer services that rely on the basic mechanisms and can be used on an optional basis, e.g., by applications that require the additional functionality.

The basic communications service provides point-to-point and multi-point messaging with FIFO-ordering and secure, i.e., authenticated and encrypted, communication. The general communication paradigm is *soft-state communication*. In addition, retransmission based reliability is (optionally) provided for point-to-point communication. For multi-point communication, a simple probabilistic reliability model is employed by limiting the deployment to local network-links, by relying on soft-state updates and by providing a liveness-detection mechanism for continuously monitoring the network link and the state of all session members.

While it is possible to implement applications that rely directly on this basic communications service, many applications require additional services such as causally ordered message transport or RPC-oriented communication. We have specified corresponding protocols as a higher layer, relying on the basic Mbus transport layer.

Ad-hoc communication mechanism: In order to be usable in ad-hoc communication scenarios where a group of peers — that are not known to each other in advance — enter a communication session, the Mbus framework has been designed to support service location and secure service association. In Section 6.4, we present the association protocol that can be used to bootstrap Mbus communication sessions.

The Mbus protocol itself provides a rendezvous mechanism that allows peers to learn of the existence of other peers and enables them to communicate without having to know communication parameters such as transport addresses in advance, thus minimizing manual configuration as far as possible. The rendezvous mechanism relies on the IP multicast

service model, where hosts can receive messages by *joining* a well-known multicast group and where messages can be sent to all group members by sending to the group address.

In order to be useful for real-world ad-hoc communication applications, the Mbus rendezvous and auto-configuration mechanisms must work well together with auto-configuration mechanisms on the network layer, namely IPv6 stateless auto-configuration [RFC2462] and IPv4 auto-configuration [Cheshire02]. The Mbus protocol is designed to operate on configured (i.e., functional) IP stacks but is not dependent on static, manually configured IP addresses.

Local coordination for multimedia conferencing systems: The application area that we want to deploy the Mbus framework for first, is *local coordination for multimedia conferencing systems*, i.e., we want to use the Mbus as a *vertical conference control protocol* that is used to coordinate a group of application entities that constitute a conferencing endpoint for a single user.

We have tried to learn from the previous efforts in that area such as the LBL conferencing bus (Section 4.2.1), PMM (Section 4.2.2) and CCCP (Section 2.2.2.2) and have designed the Mbus as a general message-oriented group communication mechanism without tying application-semantics to the base protocol. As mentioned above, the general base protocol provides the required features *service location*, *group communication* and *security*, where the application-specific semantics are defined separately (Chapter 9, *Mbus in Conferencing Systems*).

By limiting the scope of the protocol to local network links, we achieve a clear separation between Mbus and the functions of a horizontal control protocol. This deliberate restriction allows to keep the protocol comparatively simple and easy to implement.

6.2 Mbus Transport

As its basic service, the Message Bus provides local (intra-system) exchange of messages between components that attach to the Mbus (*Mbus entities*). A system is typically expected to comprise exactly one host, but may also extend across a network link and include several hosts sharing the tasks; a local system does not extend beyond a single link — the Mbus is not intended or designed for use as a horizontal (wide-area) conference control protocol.

UDP is used as transport protocol; each Mbus session uses a common IP multicast group. Periodic heartbeat messages from each entity are used as a membership information mechanism and to track the aliveness of Mbus entities.

The Mbus protocol relies on *receiver-based filtering* of messages: In general, messages are destined to an Mbus-address, are sent to the configured multicast address and are thus received by all group members. Each group member applies a matching operation by comparing its own address to the destination address and decides whether to deliver the received message to the application or not.

An Mbus destination address may either represent a group of entities (for *multicasts*) or a unique destination (for *unicasts*), and in both cases, the same message transport mechanisms are applied. Unicast messages can optionally be sent reliably by relying on an acknowledgment-based retransmission mechanism.

Mbus is a text-based protocol and defines a context-free grammar for the representation of message headers and payloads. In order to be usable as a coordination mechanism for decomposed applications in arbitrary environments, Mbus must address three security issues: message authentication, message integrity and message encryption.

In the following, we will present the Mbus protocol in some detail. The complete specification is RFC 3259 [RFC3259]. In Section 6.2.1, we present the fundamental Mbus transport mechanisms, i.e., the use of multicast communication and the reliable transport mechanism. In Section 6.2.2, we describe the rendezvous and entity awareness protocol, Section 6.2.3 explains the Mbus message syntax, and Section 6.2.4 describes the security concept. Section 6.2.5 describes how Mbus entities can be configured with respect to transport and security parameters, and Section 6.2.6 provides an overview of Mbus commands for fundamental services. Finally, we present some examples for Mbus communication interactions in Section 6.2.7.

6.2.1 Basic Transport Mechanisms

In the following, we describe the basic Mbus transport mechanisms that provide the basis for the message passing mechanism.

Addressing: Section 6.2.1.1 describes the Mbus addressing concept that is the basis for the group communication on the Mbus layer.

Group Communication: Section 6.2.1.2 explains how the Mbus addressing mechanism is used to realize the group and unicast communication on the Mbus layer, relying on the *receiver-based filtering* concept.

Multicast Communication: Section 6.2.1.3 provides a detailed description of the mapping of the Mbus message passing layer to IP multicast for IPv4 and IPv6.

Unicast Communication: Section 6.2.1.4 describes the usage of unicast communication as an optional efficiency enhancement.

Reliable Communication: Section 6.2.1.5 provides a description of the reliable transport mechanisms.

6.2.1.1 Addressing

Each entity has a unique Mbus address that can be used to identify it. The address is composed of any number of named address elements. An address element is a key-value-pair, where the key specifies the type of the address element. The types and values of address elements are application-specific with the exception of the `id` element that is used to specify a unique identifier based on a host's IP address.

Address element names may be associated with semantics: by providing certain address elements, entities can signal the type of service functionality they are able to supply. The Mbus specification itself does not impose any restrictions on application specific address elements. The semantics are provided by application specific profiles. For example, the addressing scheme for conferencing applications includes address elements such as

- media type (`media`);

- module type (`module`);
- application name (`app`); and
- application identifier (`id`) as depicted in Figure 6.1.

```
(media:audio module:engine app:rat id:1035-0@192.168.1.1)
```

Figure 6.1: Sample Mbus address

The `id` address element is used to uniquely identify entities and is mandatory for entity addresses. It is composed of a *host identifier* (an IP address) and a per host identifier, e.g., a process ID. Mbus addresses are used as source and destination specifiers in Mbus messages. Section 6.2.1.2 describes the usage of fully and partly qualified Mbus addresses for Mbus group and unicast communication.

6.2.1.2 Group Communication

A message that is to be sent via the Mbus has a destination address — at the application level — that is used to determine how to deliver the respective message. This allows for subject-based addressing-like message delivery semantics and different communication models represented by the different group addressing features:

Mbus-broadcast: When a destination address list is empty, the message is broadcast to all entities on the Mbus.

Mbus-unicast: A message providing a destination address that is a unique Mbus address of another entity will be processed by this entity only. The Mbus defines mechanisms allowing such messages to be sent directly via IP unicast to the specific entity.

Mbus-multicast: A destination addresses can be used to identify groups on a per-message basis. All entities that match the given destination address will receive and process corresponding messages: In situations where a certain module requires a specific service functionality that can be provided by more than one other module, it can use a multicast address specifying the group of service providers to locate the desired entity. This can facilitate the implementation of service clients significantly: addresses of service providers do not need to be hard-coded and the communication model can accommodate many different specific scenarios regardless of the number of potential service providers.

It is not necessary to know the exact addresses of all potential receivers of a message. Instead a sufficiently unambiguous address list can be used. For example, in order to reach all audio engines in a session the address list `(media:audio module:engine)` could be appropriate.

These different transport classes are implemented by the usage of Mbus addressing and the rules for receiver-based filtering: A fully qualified Mbus destination address will be delivered by exactly one receiving entity, whereas a partly qualified address or an empty address will be delivered to multiple or all entities.

The rules for receiver-based filtering are quite simple: Each entity has a fixed sequence of address elements constituting its address and must only process messages sent to addresses that either match all elements or consist of a subset of its own address elements. The order of address elements in an address sequence is not relevant. Two address elements match if both their tags and their values are equivalent.

```
isSubsetOf(addr a1, addr a2) yields true, iff
   every address element of a1 is contained
   in a2's address element list.
```

Figure 6.2: Algorithm for an Mbus address-matching predicate

Figure 6.2 depicts the algorithm for receiver-based filtering and Figure 6.3 lists a simplified C++ code excerpt. In this example, an Mbus address is represented as a C++ STL-container, and the C++ standard-library function `includes` is applied to two addresses, i.e., two iterator ranges, each of which represents an Mbus address.

```
const MAddress me; // my Mbus address


bool MAddress::isSubsetOf(const MAddress& a) const
{
  return ((&a == this) ||
          std::includes(a.elements.begin(),a.elements.end(),
                        elements.begin(), elements.end()));
}


int MBus::receiveMsg(const MMessage& msg, const EndpointAddr& soa)
{

  /*
   * is message for us?
   */

  const MAddress& destAddr = msg.header.destination;

  if (!destAddr.isSubsetOf(me)) {
    // message is not for us
    return 0;
  } else {
    // deliver message
  }
```

Figure 6.3: Receiver-based filtering by an Mbus implementation (simplified)

Note that the presented Mbus addressing classes do not directly map to UDP/IP-layer transport mechanisms: Obviously, a message with an Mbus-multicast or Mbus-broadcast address

should be sent using IP multicast, and a message with an Mbus-unicast address should best be sent with IP-unicast — however, the receiver-filtering allow for a less strict mapping, which we will discuss in Section 6.2.1.3.

6.2.1.3 Mapping to IP Unicast and IP Multicast

All Mbus messages are transmitted as UDP messages. There are two alternative transport methods that the Mbus addressing classes can be mapped to. Note that in the following, "unicast", "multicast" and "broadcast" mean IP unicast, IP multicast and IP broadcast respectively. It is possible to send an Mbus message that is addressed to a single entity using IP multicast.

1. Local multicast/broadcast:

   This transport method is used for all messages that are not sent to a fully qualified destination address. In order to allow for simple basic implementations, it may also be used for messages that are sent to a fully qualified destination address.

   In environments where IP multicast is not available, e.g., when hosts are employed that do not provide a complete IP implementation, IP broadcast may be used instead. We have used this for Mbus implementations on experimental systems, e.g., one-chip-computers with a minimal IP stack. However, for the future discussion in this thesis, we assume that IP multicast is used.

2. Directed unicast:

   Mbus messages that are destined to a unique Mbus address can be sent directly to the corresponding Mbus entity, using IP unicast. The details for using IP unicast as a transport method are discussed in Section 6.2.1.4.

An Mbus session can be configured for either link-local or host-local communication. Technically, the scope is limited by the scope of the IP multicast group. However, due to the different implementations and addressing architectures for IPv4 and IPv6, implementing multicast scoping is not straight-forward. RFC 2365 [RFC2365] defines the administratively scoped IPv4 multicast space, and RFC 2373 [RFC2373] defines the IP Version 6 Addressing Architecture, including multicast scopes and address prefixes.

Applying Multicast Scoping to the Mbus Protocol

We have specified the usage of Mbus for both IPv4 and IPv6. Since the addressing architecture of both protocols differs, different rules for the usage of multicast have been specified. Table 6.1 provides a mapping of IPv4 multicast scopes for the corresponding IPv6 scopes as given by RFC 2365 [RFC2365]. The IPv6 addressing architecture provides a more fine-grained scope structure, e.g., for IPv6, there are dedicated link-local and site-local scopes, whereas the smallest IPv4 multicast scope is the link-local scope. Compared to the IPv6 address space, the IPv4 address space is not segmented uniformly, which is due to the historic development of the addressing architecture and the different address assignment strategies that have been defined in the past.

Table 6.1: Multicast scopes for IPv6 and IPv4 [RFC2365]

IPv6-Scope	RFC 2373 Description	IPv4 Prefix
0	reserved	
1	node-local scope	
2	link-local scope	224.0.0.0/24
3	(unassigned)	239.255.0.0/16
4	(unassigned)	
5	site-local scope	
6	(unassigned)	
7	(unassigned)	
8	organization-local scope	239.192.0.0/14
A	(unassigned)	
B	(unassigned)	
C	(unassigned)	
D	(unassigned)	
E	global scope	224.0.1.0-238.255.255.255
F	reserved	
	(unassigned)	239.0.0.0/10
	(unassigned)	239.64.0.0/10
	(unassigned)	239.128.0.0/10

For IPv4, the link-local scope is `224.0.0.0/24` — an 8-bit address-space that is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols. It is not intended to be used by applications.

For this reason, we do not use the IPv4-link-local scope, but a *Local-Scope-address* as defined by RFC 2365 (Administratively Scoped IP multicast, [RFC2365]). RFC 2365 provides a similar concept as the scope-independent group identifiers that are used for permanently-assigned IPv6-multicast addresses: The high order `/24` space in every scoped region is reserved for relative assignments. A relative assignment is an integer offset from the highest address in the scope and represents a 32-bit address. For example, in the Local Scope, `239.255.255.0/24` is reserved for relative allocations. The offset (a number from 0 to 255) can therefore be viewed as a scope independent group identifier and is used together with the low order bits for a given IPv4-multicast scope in order to construct a complete IPv4-multicast address.

The administratively scoped IP multicast architecture distinguishes different scopes: The IPv4 Local Scope (239.255.0.0/16) is the minimal enclosing scope for administratively scoped multicast and is not further divisible — its exact extent is site dependent. The offset for the scope relative address for Mbus is 8. For the IPv4 Local Scope, applying the rules of RFC 2365 and using the assigned offset of 8, the Mbus multicast address is therefore 239.255.255.247. For IPv4, the different defined Mbus scopes (host-local and link-local) are implemented as follows:

IPv4 host-local multicast: Unless configured otherwise, the assigned scope-relative Mbus address in the Local Scope (`239.255.255.247`) is used. Mbus UDP datagrams are sent

with a TTL of 0.

In essence, we rely on TTL scoping for host-local multicast as a way to effectively limit the distribution of traffic to exactly one host.

IPv4 link-local multicast: Unless configured otherwise, the assigned scope-relative Mbus address in the Local Scope (`239.255.255.247`) is used. Mbus UDP datagrams should be sent with a TTL of 1.

Again, we use a combination of TTL scoping and an administratively scoped address, in order to ensure that — e.g., in the presence of routers that do not support administratively scoped addresses — Mbus traffic does not leave a local network link. The mentioned disadvantages with respect to multicast routing protocols and the use of TTL scoping obviously do not apply for link-local multicast, because the last router and the source reside on the same network link, so pruning would not be applied anyway.

For IPv6 we simply use two different addresses for host-local and for link-local scope:

IPv6 host-local multicast: The permanent IPv6 host-local multicast address for Mbus over IPv6 is FF01:0:0:0:0:0:0:300.

IPv6 link-local multicast: The permanent IPv6 link-local multicast address for Mbus over IPv6 is FF02:0:0:0:0:0:0:300.

The usage of other multicast scopes for IPv6 (and IPv4) is not defined.

6.2.1.4 Unicast Communication

Directed unicast (via UDP) to the port of a specific application is an alternative transport class to multicast. Directed unicast is an optional optimization and may be used by Mbus implementations for delivering messages addressed to a single application entity only — the address of which the Mbus implementation has learned from other message exchanges before. Every Mbus entity should use a single unique endpoint address for sending messages to the Mbus multicast group or to individual receiving entities. A unique endpoint address is a tuple consisting of the entity's IP address and a UDP source port number, where the port number is different from the standard Mbus port number.

Messages are only sent via unicast if the Mbus destination address is unique and if the sending entity can verify that the receiving entity uses a unique endpoint address. The latter can be verified by considering the last message received from that entity. Implementations that wish to use unicast transport, maintain a table of Mbus addresses and the corresponding endpoint addresses. Note that several Mbus entities, say within the same process, may share a common endpoint address; in this case, the contents of the destination address field is used to further dispatch the message. Given the definition of "unique endpoint address" above, the use of a shared endpoint address and a dispatcher still allows other Mbus entities to send unicast messages to one of the entities that share the endpoint address. So this can be considered an implementation detail.

Messages with an empty destination address list are always sent to all Mbus entities (via multicast if available). The algorithm depicted in Figure 6.4 can be used by sending entities

to determine whether an Mbus address is unique considering the current set of Mbus entities. The algorithm checks the uniqueness of a destination address `ta` by counting the number of addresses for which `ta` is a subset. If `ta` is a subset address of only one of the known entity addresses (`ta` itself), `ta` is unique.

```
let ta=the destination address;
iterate through the set of all
currently known Mbus addresses {
   let ti=the address in each iteration;
   count the addresses for which
   the predicate isSubsetOf(ta,ti) yields true;
}
```

Figure 6.4: Algorithm for determining the uniqueness of an Mbus address

Figure 6.5 depicts a C++ implementation of the algorithm described by Figure 6.4 relying on the function `isSubsetOf` that is listed in Figure 6.3. The function `isUnique` iterates through all the Mbus addresses that are specified by the range designated by `start` and `end` and checks for each address whether it matches the tested Mbus address `m` (that is assumed to be an element of the range). The Mbus address `m` is unique if (and only if) exactly one match has been found (the equivalent of `m`).

```
// check if MAddress is unique on the Mbus
// (try to match every address in map)

template<class Input>
bool isUnique(const MAddress& m, Input start, Input end)
{
  int total=0;

  for(;(total<=1) && (start!=end);start++) {
    if(m.isSubsetOf(start->first)) {
      total++;
    }
  }
  return (total==1);
}
```

Figure 6.5: Implementation of an algorithm for determining the uniqueness of an Mbus address

If the count of matching addresses is exactly 1, the address is unique. The algorithm depicted in Figure 6.2 is used by the predicate `isSubsetOf` that checks whether the second address matches the first. (A match means that a receiving entity that uses the second Mbus address must also process received messages with the first address as a destination address.)

An address element `a1` is contained in an address element list if the list contains an element that is equal to `a1`. An address element is considered equal to another address element if it has the same values for both of the two address element fields (tag and value).

6.2.1.5 Reliable Communication

Although the Mbus is limited to host local and network link local communication, transmission failures cannot be excluded. In the following, we first analyze some failure models that are relevant for the local Mbus communication. Subsequently, we describe a reliable transport mechanism for Mbus messages that addresses the identified failure models and provides the possibility for senders to transmit messages with a higher degree of reliability and to detect transmission failures.

Failure Models for Mbus Communication

We can roughly distinguish three different causes for transmission failure for Mbus communication: network congestion, overloaded hosts and intermittent connectivity, which we will analyze in more detail in the following.

Network Congestion

In the Internet, the dominant reason for packet loss is *network congestion*. Packet loss due to transmission failures is not very common in typical wired and wireless network links. For example, switched Ethernets reduce the probability of very high frame collision rates (that could cause packet loss) and are typically highly over-dimensioned. Moreover, some link layer protocols such as IEEE 802.11 [IEEE99] (that cannot detect frame collisions reliably) provide link layer retransmissions and flow control mechanisms themselves. Gevros, Crowcroft, Kirstein and Bhatti define network congestion as the *state of sustained network overload where the demand for network resources is close to or exceeds capacity* [Gevros01]. There are two network resources that have to be considered: network bandwidth and router buffer space. The reason why network congestion can occur fundamentally lies in the *Internet Service Model* that is based on an *best effort service* in conjunction with uncoordinated resource sharing [Gevros01].

Congestion control mechanisms are (usually) not provided on the network layer, i.e., the inter-network packet forwarding service, but on the transport layer, i.e., by protocols such as TCP and SCTP. Due to the common use of mandatory congestion control mechanisms for TCP implementations, the congestion control in the Internet is largely based on the cooperative usage of TCP's congestion control mechanisms, namely its *acknowledgment window management algorithm* and the *Slow Start* and *Congestion Avoidance* algorithm [RFC2581].

For unicast communication that uses TCP or other, so-called *TCP-friendly*, protocols, the congestion control mechanisms work sufficiently well. Multicast datagram-transmission-based communication however, is in general more likely to cause congestion problems than unicast (connection-oriented) communication. This is due to the nature of the flow control and congestion-avoidance algorithms that typically work well for unicast communication, where a sender can adapt its sending behavior according to feedback it receives from its communication peer.

For multicast communication, congestion control and congestion avoidance is typically much harder to realize because feedback-based mechanisms as described above have intrinsic scalability problems, especially in multi-sender configurations. One way of implementing at least some form of congestion avoidance is to limit the sending-rate (*rate-control*). However, by setting the sending-rate to a certain limit, applications and protocol stacks cannot adapt to

varying network conditions anymore. Even though some transport protocols such as RTP provide means for participants to analyze the receiver statistics of other receivers, it is not trivial to infer an appropriate way of adapting the sending behavior (in a timely fashion).

In general, the full complexity of the congestion control issue with multicast communication applies primarily to Internet, i.e., inter-domain, multicast communication, where different heterogeneous network links with vastly different characteristics have to be traversed. For example, some group members may be connected to a high-speed, over-provisioned Ethernet, whereas others may be connected to a low-bandwidth wireless link. Of course, overloaded backbone routers and backbone links can also cause congestion.

For link-local multicast communication, congestion control is more manageable, because all group members share the same link and are (in a simplified model) able to observe the same link characteristics. When congestion is observed (e.g., by experiencing packet loss), all group members could infer to reduce their sending rates.

However, this does not mean that congestion is not an issue for link-local multicast. In bridged topologies that are prevalent in many networks, a *local network link* is not necessarily a homogeneous link but can consist of high-bandwidth segments and low-bandwidth segments that appear as one network. For example, 802.11 Wireless LAN access points are typically connected to an existing Ethernet segment as *transparent bridges*, forwarding all multicast and broadcast traffic between the attached network segments.

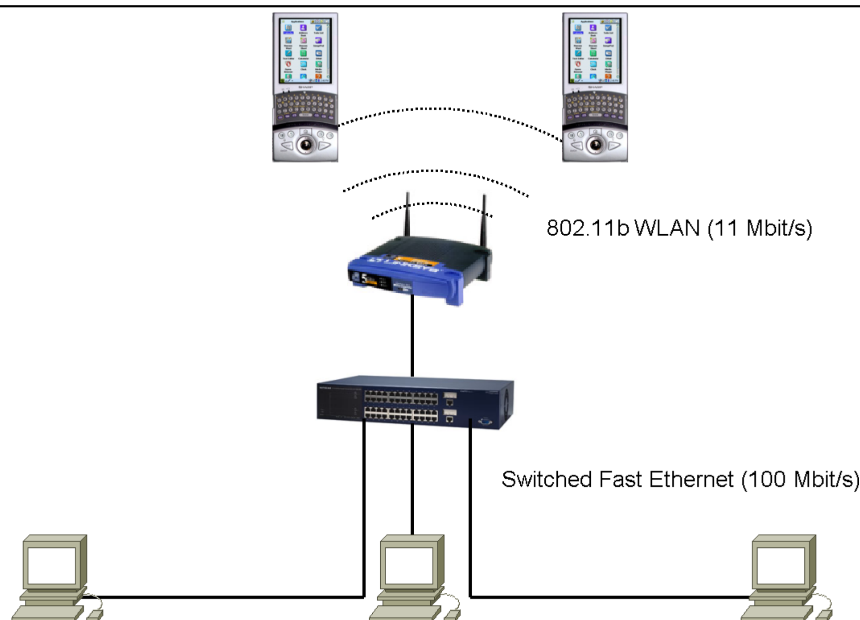

Figure 6.6: A bridged local network (WLAN and Ethernet)

Figure 6.6 depicts such a network architecture. Multicast traffic that originates in the Ethernet segment could easily flood and congest the WLAN segment which would be difficult to detect by sending hosts in the Ethernet segment. In this special configuration, the situation would be aggravated because WLAN access points do not send multicast traffic using the highest nominal sending rate but using the *lowest* standardized sending rate in order to make sure that every host is able to receive data (IEEE 802.11 provides different rate modes for different

levels of link quality).

This scenario highlights some of the problems of multicast with respect to congestion control and reliable communication. The problems that we have described mainly apply to high-bandwidth communication, e.g., multimedia real-time streaming, but the fundamental problem is independent of the specific bandwidth and type of communication. At least, protocols should be able to *detect* failure situations and be able to react properly, e.g., by notifying the application as a last resort.

Overloaded Hosts

In addition to packet loss due to network congestion, we have to take transmission delays and buffer overflows on hosts into account — another form of congestion. We distinguish two failure causes:

- Typical UDP/IP implementations provide a limited buffer for UDP messages that have been sent to a port but have not yet been read by applications. If this limit has been reached, UDP packets will be dropped (without further notice to the application). This can be generalized as a *flow control* problem, where the receiver is not able to process the received messages fast enough.
- Mbus applications and Mbus protocol implementations are likely to run in user space (of multi-user operating systems) and can thus be affected by delays caused by preemption and general system overload. In single-threaded applications there can also be delays caused by blocking input/output operations. This can either lead to the first problem (the application can not read fast enough) or to the problem that an application has read a specific message but is not able to react timely and, e.g., send a response message.

Intermittent Connectivity

In addition to failures caused by congestion, we have to take into account that in ad-hoc communication scenarios, entities are likely to suffer from *intermittent connectivity*, especially in wireless environments. Mobile entities can enter and leave the range of a wireless base station, or they can roam between different layer 2 networks and experience periods of interrupted connectivity.

In wireless scenarios without central base stations, i.e., *ad-hoc networks* in the sense of uncoordinated stations with potential asymmetric connectivity, the situation is even worse, because inconsistencies with respect to the set of available stations (and correspondingly Mbus entities) can arise. For example, the so-called *hidden-terminal-problem* can lead to situations where a station `A` can communicate with two stations `B` and `C`, but `B` and `C` do not have direct connectivity. Figure 6.7 depicts such a scenario.

In this scenario, routing protocols for *mobile ad-hoc-networking* such as AODV [Perkins02] could be applied, however without guaranteeing constant connectivity between all hosts at all times. With respect to our reliability discussion, we can state that the design of the reliability mechanism has to take short periods of intermittent connectivity into account. These phases of intermittent connectivity would not necessarily be detected by the periodic heartbeat mechanism (see Section 6.2.2) and could thus lead to unexpected message loss.

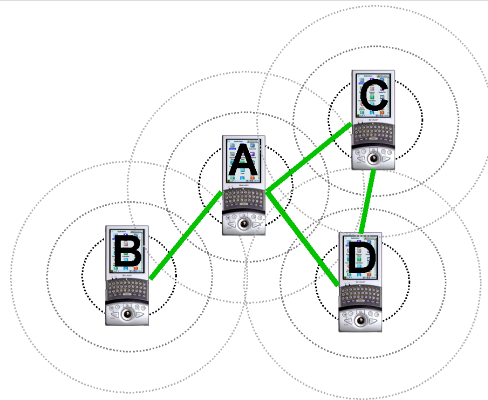

Figure 6.7: The hidden-terminal problem

Summary of Transmission Failure Causes

Summarizing these observations, we can state the following requirements for an Mbus reliability mechanism:

- It must be possible to detect message loss that may be caused by network link congestion, transmission failures, mobility or system crashes.
- Although the protocol is intended to be used in a local network, delays have to be taken into account that may be caused by system overload and blocked execution threads on hosts.
- We have to take intermittent connectivity into account, i.e., sporadic message loss that might not be correlated to network and host load.

For the Mbus protocol, we provide two mechanism that address these failure models:

- a retransmission-based reliable transport mode; and
- a soft-state concept for sending information periodically.

  Periodic *soft-state updates* can replace retransmission-based reliability mechanisms in many cases and should be used for communication that is idempotent and does not require an acknowledgment per message. In situations of prolonged loss of connectivity, periodic soft-state updates are more appropriate than retransmission-based reliability mechanisms, because the sender can simply continue sending, and the receiver will eventually, after the connectivity has been re-established, receive the information.

The Mbus Mechanism for Reliable Message Transport

With these observations and requirements in mind, we have designed a reliability mechanism for communication scenarios where soft-state updates are not sufficient. This mechanism can be activated on a per-message basis (i.e., when required) and relies on an acknowledgment

and retransmission scheme. Applications are thus enabled to detect message delivery problems, and the Mbus protocol layer can retransmit messages that have not been acknowledged.

It should be noted that a sender will typically not be able to detect the reason *why* a particular message has not been received. When relying on acknowledgment-based feedback mechanisms, the sender will usually only be able to detect that no acknowledgment has been received, which may have one of the following reasons:

- the original message has been lost on the way to the receiver;
- the original message has been received by the intended recipient, however no acknowledgment has been sent due to receiver-internal reasons, e.g, a system crash; or
- the receiver has acknowledged the message but the acknowledgment has not been received (in time) at the original sender.

In order to maintain scalability and to allow for simple implementations, reliable delivery is only defined for Mbus-unicast messages, i.e. for messages with a single recipient only. As a consequence, the reliability mechanism can only be used for messages with a fully qualified Mbus destination address. An entity can thus only send reliable messages to known addresses, i.e., only to entities that have announced their existence on the Mbus (e.g., by the Mbus rendezvous mechanism as defined in Section 6.2.2). A sending entity does not send a message reliably if the destination address is not unique. A receiving entity will only process and acknowledge a reliable message if the destination address exactly matches its own source address (the destination address must not be a subset of the receiver's address).

Although sending Mbus-multicast messages reliably is not supported by the base protocol, it is of course possible: The necessary coordination can be performed at the application layer by sending individual reliable messages to each fully qualified destination address, if the membership information for the Mbus session is available — taking the mentioned scalability problems into account.

Each message is tagged with a message sequence number. If the message type is set to `reliable`, the sender expects an acknowledgment from the recipient within a short period of time. If the acknowledgment is not received within this interval, the sender retransmits the message (with the same message sequence number), increases the timeout, and restarts the timer. Messages are to be retransmitted a small number of times before the transmission or the recipient are considered to have failed. If the message is not delivered successfully, the sending application is notified. In this case, it is up to the application to determine the specific actions (if any) to be taken.

The timer for acknowledgments increases *linearly* with a small maximum number of retransmissions (3) , i.e., the retransmission uses a *linear back-off* scheme. Other protocols such as Ethernet's CSMA/CD use exponential back-offs, however these are targeted at reacting to sustained network link congestion.

Reliable messages are acknowledged by adding their sequence number to a list of acknowledgments in a header field of a message sent to the originator of the reliable message. This message is always sent to a fully qualified Mbus destination address. Multiple acknowledgments can be sent in a single message. Implementations can either piggyback the acknowledgment list onto another message sent to the same destination, or can send a dedicated acknowledgment message, without any payload.

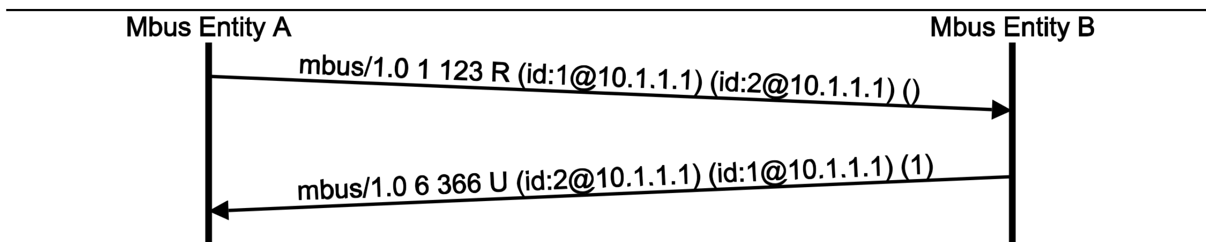

Figure 6.8: Acknowledging a reliable message

Figure 6.8 depicts a message exchange where entity B acknowledges the message with the sequence number `1` it has received from entity A before. In Figure 6.9 it is shown how entity B acknowledges multiple reliable messages (sequence numbers `1`, `2`, and `3`) in a single acknowledgment. In both figures, the message bodies are not shown.

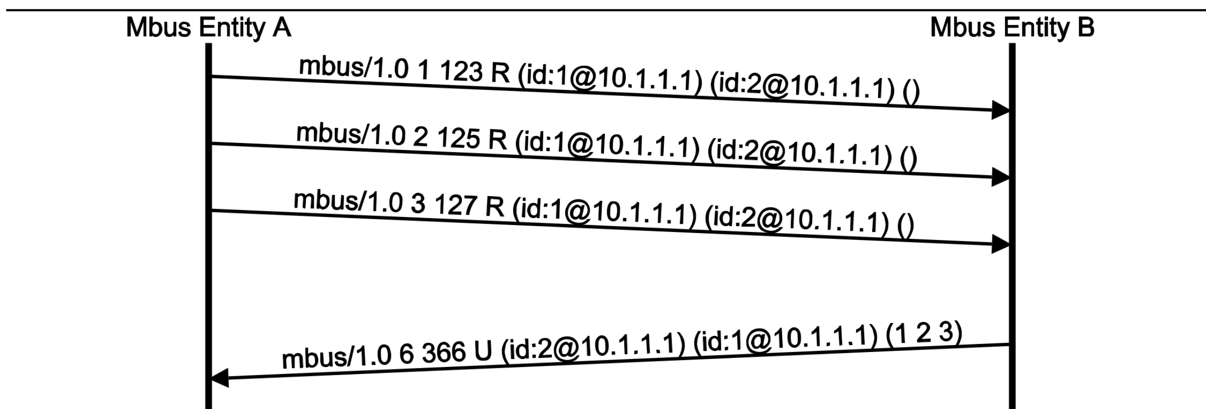

Figure 6.9: Acknowledging multiple reliable messages

In the Mbus retransmission protocol, a message is retransmitted three times (with linearly increasing back-off durations), before an error condition is indicated to the application. The precise procedures are as follows:

Sender: A sender `A` of a reliable message `M` to receiver `B` transmits the message either via IP multicast or via IP unicast, keeps a copy of `M`, initializes a retransmission counter `N` to `1`, and starts a retransmission timer `T` (initialized to `T_r`). If an acknowledgment is received from `B`, timer `T` is canceled and the copy of `M` is discarded. If `T` expires, the message `M` must be retransmitted, the counter `N` is incremented by one, and the timer is restarted (set to `N*T_r`). If `N` exceeds the retransmission threshold `N_r`, the transmission is assumed to have failed, further retransmission attempts will not be undertaken, the copy of `M` is discarded, and the sending application will be notified.

Receiver: A receiver `B` of a reliable message from a sender `A` acknowledges the reception of the message within a time period `T_c` $<$ `T_r` (transmission delay is not taken into account in order to simplify the protocol). This can be done by means of a dedicated

acknowledgment message or by piggybacking the acknowledgment on another message addressed only to A.

Receiver optimization: In a simple implementation, B may choose to immediately send a dedicated acknowledgment message. However, for efficiency, it could add the sequence number of the received message to a sender-specific list of acknowledgments; if the added sequence number is the first acknowledgment in the list, B starts an acknowledgment timer TA (initialized to T_c). When the timer expires, B creates a dedicated acknowledgment message and sends it to A. If B is to transmit another Mbus message addressed only to A, it should piggyback the acknowledgments onto this message and cancel TA.

In either case, B should store a copy of the acknowledgment list as a single entry in the per-sender copy list, keep this entry for a period T_k, and empty the acknowledgment list. In case any of the messages kept in an entry of the copy list is received again from A, the entire acknowledgment list stored in this entry is scheduled for (re-) transmission following the above rules.

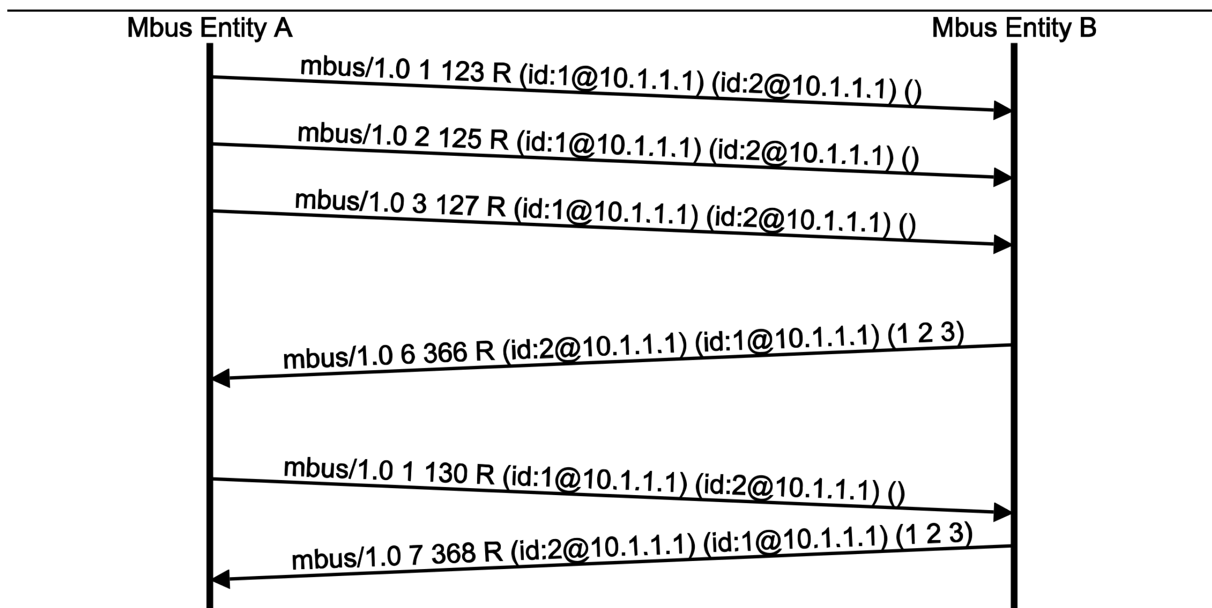

Figure 6.10: Retransmitting a complete list of acknowledgments

The retransmitting of complete acknowledgment lists is intended as an optimization in cases where acknowledgment messages have been lost. In order to avoid original senders to re-send every un-acknowledged message that has in fact already been received, an receiver can send the complete list of recently sent acknowledgments again, thus reducing the total number of messages that are required to converge the views about sent and received messages.

Figure 6.10 depicts a corresponding message exchange, where entity A sends a series of reliable messages to entity B. Entity B acknowledges the reception of all of these messages in a single message to A. For some reason, A does not receive this acknowledgment

(or does not receive it in time) and starts to retransmit its first message to B. Upon receiving the retransmitted message (with sequence number 1), entity B sends out a new message that again acknowledges the reception of all recently received messages from A. Of course, due to concurrency, it is not deterministically predictable that entity A receives the second acknowledgment message early enough to cancel the scheduled retransmissions of message 2 and message 3. However, this does not incur consistency problems, as B would only receive a retransmission of one or two messages and re-send its acknowledgment lists.

Constants: In RFC 3259, we have specified the following constants for timers:

$$\begin{aligned} T_r &= 100ms \\ N_r &= 3 \\ T_c &= 70ms \\ T_k &= N_r * \frac{N_r + 1}{2} * T_r \end{aligned}$$

The receiver optimization of delaying the sending of acknowledgments is intended for increasing the likelihood that the acknowledgment can be piggybacked onto another message that is sent at a later time from the original receiver to the original sender of the reliable message. For many communication scenarios where reliable Mbus unicast communication is required, this is often the case. For example, in request/response scenarios, a sender would send the request reliably, and the receiver would piggyback the acknowledgment directly onto the response message that would typically be sent reliably as well.

In a series of request/response interactions, the acknowledgment retention in combination with piggybacking can thus effectively avoid the necessity of sending empty acknowledgment messages. In these scenarios the requests may not always directly follow the reception of a response, instead they may be sent asynchronously, e.g., triggered by external events. If a subsequent request is sent before the timer acknowledgment retention expires, the acknowledgment will still be piggybacked onto the new request.

Figure 6.11 depicts an Mbus message exchange where entity A sends two requests to entity B (message bodies are not shown). The replies from entity B carry the acknowledgment for the corresponding request according to the acknowledgment piggybacking rules. In the second request from entity A, the preceding response from entity B (that had been sent reliably as well) can still be acknowledged by piggybacking the acknowledgment onto the new request message, although it is not sent directly after the reception of the first response message from B.

In Section 6.3.2.2, we describe an Mbus RPC mechanism that provides a standardized message format for RPC-like communication, e.g., in order to correlate responses to requests. This mechanism benefits directly from the optimizations that acknowledgment piggybacking and retention allow for. In Section 8.2, we present some simulation results that show the effects of acknowledgment piggybacking for certain Mbus interactions.

It should be noted that acknowledgment piggybacking and retention are *optional* optimizations, i.e., it is legitimate for simple implementations not to implement these (as senders of acknowledgments). However, they are still able to inter-operate with more advanced implementations, which are in any case required to accept empty acknowledgment messages.

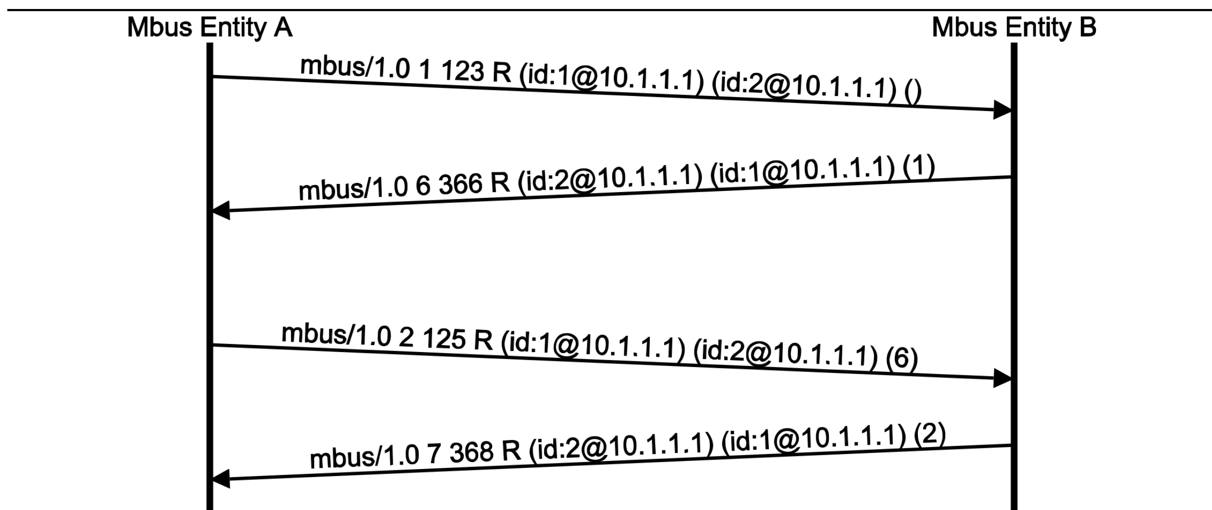

Figure 6.11: Acknowledgment piggybacking and retention in request/response scenarios

6.2.2 Membership Information Service

The Mbus is intended for dynamic, rather loosely coupled environments where the set of session members is neither fixed nor known in advance. Instead Mbus entities can join and leave sessions arbitrarily without the need for explicit membership control operations such as admission requests and indications on leaving a session.

When we think of mobile and ad-hoc communication scenarios, we can state that this liberal membership policy is actually desirable as it tolerates the temporary disruption of connectivity and the dynamic joining of members that enter a wireless network's range.

In order to accommodate these dynamics, we have designed a light-weight membership information service that informs applications about changes in the group membership, thus enabling applications to continuously track the set of available entities. In order to meet the requirements of scalability and robustness, this membership information service cannot be centralized but has to be a distributed service. The Mbus membership information service relies on periodic Mbus broadcasts and serves three main functions:

- it provides each session member with the current set of Mbus entities and their Mbus addresses;
- it enables each session member to maintain an up-to-date mapping of Mbus addresses to UDP/IP endpoint addresses, which can be used for implementing the unicast optimization as described in Section 6.2.1.4; and
- it serves as a *rendezvous* function upon session initiation and joining of sessions: A new session member immediately starts sending its periodic heartbeat messages, thus informing existing other members of its existence. At the same time, the new member learns of the existence of the other members by receiving their heartbeats, without the need for active queries.

Before Mbus entities can communicate with one another, they need to mutually find out about their existence. After this bootstrap procedure that each Mbus entity goes through, all

other entities listening to the same Mbus know about the newcomer and the newcomer has learned about all the other entities. Furthermore, entities need to be able to notice the failure (or leaving) of other entities, e.g., in a decomposed application, some modules may provide crucial functions. If a controlling module detects that a crucial entity has left, it could notify the user or terminate the whole application.

Each Mbus entity maintains an estimated number of Mbus session members and continuously updates this number according to any observed changes. The existence and the departure of other entities can be detected by dedicated Mbus messages for entity awareness: `mbus.hello` and `mbus.bye` (Section 6.2.6.1). An Mbus protocol implementation periodically sends `mbus.hello` messages that are used by other entities to monitor the existence of that entity. If an entity has not received `mbus.hello` messages for a certain time from an entity, the respective entity is considered to have left the Mbus and is excluded from the set of currently known entities. Upon the reception of an `mbus.bye` message the respective entity is considered to have left the Mbus as well and is excluded from the set of currently known entities immediately.

The interval for sending `hello` messages is dependent on the current number of entities in an Mbus group and can thus change dynamically in order to avoid congestion due to many entities sending `hello` messages at a constant high rate.

Section 6.2.2.1 specifies the calculation of `hello` message intervals that are used by protocol implementations. Section 6.2.6.1 presents the command synopsis for the corresponding Mbus messages.

6.2.2.1 Hello Message Transmission Interval

Since the number of entities in an Mbus session may vary, care must be taken to allow the Mbus protocol to automatically scale over a wide range of group sizes. Without adaptation, the average rate at which hello messages are received would increase linearly with the number of entities in a session. Given an interval of 1 second, this would mean that, in an Mbus session with `n` entities, each entity would receive n hello messages per second. Assuming all entities resided on one host, this would lead to `n*n` messages that have to be processed per second — which is obviously not a viable solution for larger groups. It is therefore necessary to deploy dynamically adapted hello message intervals, taking varying numbers of entities into account. In the following, we present an algorithm that is used by implementors to calculate the interval for hello messages considering the observed number of Mbus entities. The algorithm features the following characteristics:

- The number of `hello` messages that are received by a single entity in a certain time unit remains approximately constant as the number of entities changes. For an Mbus session on a single host, the processing effort for the host system would increase linearly (instead of quadratically without adaptation) with the number of entities.

- The effective interval that is used by a specific Mbus entity is randomized in order to avoid unintentional synchronization of hello messages within an Mbus session. The first `hello` message of an entity is also delayed by a certain random amount of time.

- A timer reconsideration mechanism is deployed in order to adapt the interval more appropriately in situations where a rapid change of the number of entities is observed. This is

useful when an entity joins an Mbus session and is still learning of the existence of other entities or when a larger number of entities leave the Mbus at once.

The details of the timer calculation algorithms are described in [RFC3259].

6.2.3 Message Syntax

Figure 6.12 depicts an example of a complete Mbus message providing multiple commands. In addition to the message header and the message body, each Mbus message provides a leading authentication line that contains a message digest (see Section 6.2.4 for details).

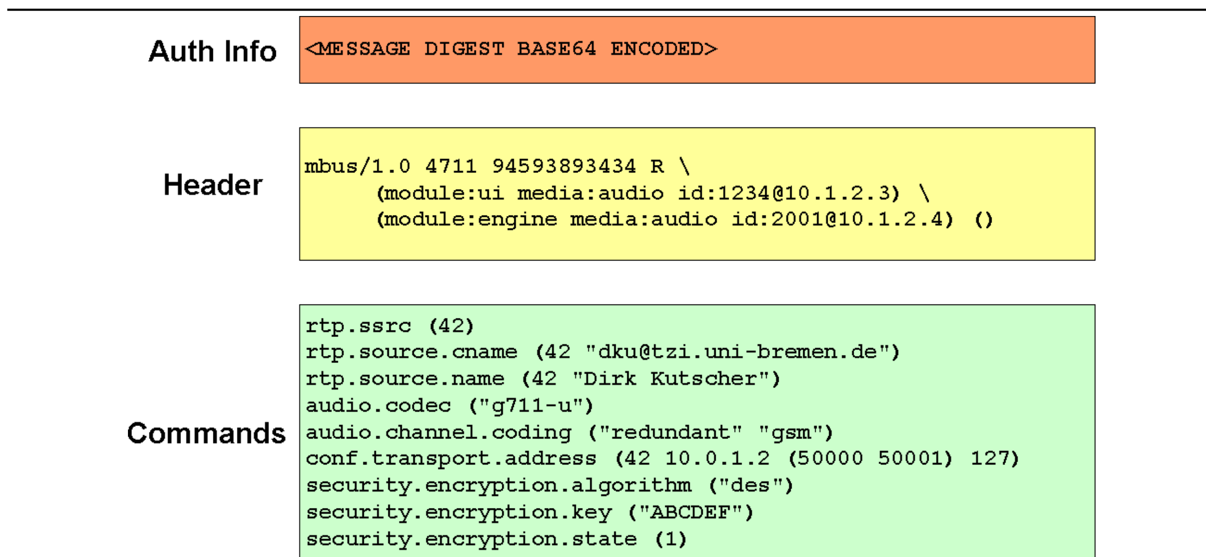

Figure 6.12: Mbus message syntax

An Mbus message comprises a header and a body. The header is used to indicate how and where a message should be delivered and the body provides information and commands to the destination entity. The following pieces of information are included in the header:

Protocol identifier: A fixed ProtocolID field identifies the version of the message bus protocol used, currently `mbus/1.0`.

Sequence number: A sequence number is contained in each message. Sequence numbers are decimal numbers in ASCII representation. The first message sent by a source provides a sequence number of zero, which is incremented by one for each message sent by that source. A single sequence number is used for all messages from a source, irrespective of the intended recipients and the reliability mode selected. The value range of a sequence number is (0,4294967295). An implementation must re-set its sequence number to 0 after reaching 4294967295. Implementations must take into account that the sequence number of other entities may wrap-around.

Timestamp: The Timestamp field is also contained in each message and contains a decimal number representing the time of the message construction in milliseconds since 00:00:00, UTC, January 1, 1970.

Message type: A message type field indicates the kind of message being sent. The value R indicates that the message is to be transmitted reliably and must be acknowledged by the recipient, U indicates an unreliable message, which must not be acknowledged.

Source address: The source address field identifies the sender of a message. This is a complete Mbus address, with all address elements specified.

Destination address: The destination address field provides a single Mbus address that identifies the intended recipient(s) of the message. The address may be wildcarded, i.e., partly qualified, by omitting address elements and hence address any number (including zero) of application entities. The destination address list may also be empty, i.e., denoting the group of all Mbus entities in a session.

Acknowledgment List: The acknowledgment list field comprises a list of sequence numbers for which this message is an acknowledgment. If the message does not acknowledge any other message, this list is empty.

The header is represented by a single line of text, and the header fields are separated by whitespace. The detailed syntax specification is provided by RFC 3259. Figure 6.13 depicts a sample Mbus message header.

```
mbus/1.0 32 127330077 U (app:mbussh id:2090-0@192.168.2.1) \
(id:2091-0@192.168.2.1 app:foo) (17)
```

Figure 6.13: Sample Mbus header

The header is followed by the message body, which contains zero or more commands to be delivered to the destination entities. If multiple commands are contained within the same Mbus message payload, they must be delivered to the Mbus application in the same sequence in which they appear in the message payload.

An Mbus command provides a command name and a parameter list (which may be empty). We have defined a context-free grammar that allows uniquely encoding different parameter types in text representation. The notation has been inspired by *Symbolic Expressions* (*S-expressions*) [McCarthy60] [Rivest97], which are used by the Lisp programming language and its derivates such as Scheme for representing expressions in programs. S-expressions are a data structuring mechanism for representing arbitrary complex data structures. They are suited for a text-based coordination protocol, because they allow for efficiently notating hierarchically structured expressions. Hence, expressions can be parsed by receivers without having to understand the semantics of a message — implementing the concept of *self-describing messages* that we have also noticed in our description of TIBCO Rendezvous (Section 4.3.1). For Mbus expressions, the fundamental data type is the `list` type. A `list` is a collection of parameters of different types, including `list` parameters of arbitrary length. Other data types that are defined are `string`, `symbol`, and the numerical types `integer` and `float`. Opaque binary data can be encoding using the type `data`, and values of this type are represented using

the Base64-encoding [RFC1521]. The syntax specification for Mbus commands and command parameters is provided by RFC3259.

```
tool.rat.codec ("GSM" "Mono" "8-kHz")
sample.msg     ("string" SYMBOL 1 3.14 <bdbdhj>)
sample.nested  ((key1 "value 1") (key2 "value2"))
```

Figure 6.14: Sample Mbus commands

Figure 6.14 depicts three sample Mbus commands with parameters. The command `tool.rat.codec` provides a parameter list of three parameters that are all of type `string`. The command `sample.msg` provides a parameter list of five parameters. The first parameter is of type `string`, the second parameter is of type `symbol`, the third parameter is of type `integer`, the fourth parameter is of type `float`, and the fifth parameter is of type `data`. The command `sample.nested` provide a parameter list of two parameters of type `list`. Each of the two list parameters provides an element of type `symbol` and an element of type `string`.

6.2.4 Security

In Section 3.2.10, we have described security threats that we have to consider for the Mbus protocol, and we have derived the following security services that have to be provided by the transport protocol:

- message authentication;
- message integrity; and
- confidentiality.

Obviously, providing these security services is associated with a cost — the main factor is the computational overhead required by cryptographic algorithms, and another cost factor is a potentially increased message size for additional header fields. In order to meet the *efficiency* requirements (Section 3.2.5) and in order not to preclude the usability of Mbus for systems with limited computational resources, we have created a security framework providing fundamental mechanisms for security functions that can be employed by applications requiring the corresponding services. In general, all Mbus security functions rely on symmetric (shared-key) cryptography and not (computationally more demanding) public-key cryptography. For example, we provide mechanisms for message encryption however, we do not *require* protocol implementations to use encryption, and we do not restrict the protocol to use specific algorithms.

Authentication and *message integrity* are the most critical Mbus security features. The Internet multicast service model provides a rather loose coupling of members in a multicast session and does not provide any group security or membership management functions at all. For example, any host on the Internet can send to a multicast group without even being a member of the group — provided that global multicast routing is in place.

The fact that the Mbus protocol is only intended to be used in local networks, does not exclude the possibility that some other hosts send messages to the same multicast group and the corresponding port. This is especially a problem when administrative multicast scoping is not in place, and any host can send to the global scope by setting the multicast-TTL field correspondingly. For an application coordination protocol such as Mbus, it is an absolute necessity to guarantee (by the use of strong cryptography) that only authorized communication peers can gain control of application components that are executed on behalf of a user. In order to prevent accidental or malicious disturbance of Mbus communications through messages originated by applications from other users and in order to be able to detect third-party manipulation of messages in transit, message authentication is deployed as a mandatory feature that has to be supported by every Mbus implementation. For each message, a digest is calculated that is based on the value of a shared secret key value. Receivers of messages check if the sender belongs to the same Mbus security domain by re-calculating the digest and comparing it to the received value. The messages are only processed further if both values are equal. In order to be able to tolerate different simultaneous Mbus sessions at a given scope, e.g., belonging to different users that accidentally use the same address/port number configuration, and in order to compensate defective implementations of host local multicast, message authentication must be provided by conforming implementations. We describe the details of the Mbus authentication mechanism in Section 6.2.4.1.

Privacy of Mbus message transport can be achieved by optionally using symmetric encryption methods (Section 6.2.4.2). Each message may be encrypted using an additional shared secret key and a symmetric encryption algorithm. Encryption is optional for applications, i.e., it is allowed to configure an Mbus domain not to use encryption. But conforming implementations must provide the possibility to use message encryption (see below). The exact order of applying authentication and encryption functions at senders and receivers is described in Section 6.2.4.3.

Message authentication and encryption can be parameterized by parameters such as the algorithms to apply and the keys to use. These and other parameters are defined in an Mbus configuration object that is accessible by all Mbus entities that participate in an Mbus session. In order to achieve interoperability, conforming implementations typically use the values provided by such an Mbus configuration. Section 6.2.5 defines the mandatory and optional parameters as well as storage procedures for different platforms. Only in cases where none of the options mentioned in Section 6.2.5 is applicable, alternative methods of configuring Mbus protocol entities may be deployed.

6.2.4.1 Message Authentication

For authentication of messages, hashed message authentication codes (HMACs) as described in RFC 2104 [RFC2104] are deployed. In general, implementations can choose between a number of digest algorithms. For Mbus authentication, the HMAC algorithm is applied in the following way:

- The keyed hash value is calculated using the HMAC algorithm specified in RFC 2104. The specific hash algorithm and the secret hash key are obtained from the Mbus configuration (see Section 6.2.5).
- The keyed hash values are truncated to 96 bits (12 bytes).

- Subsequently, the resulting 12 bytes are Base64-encoded, resulting in 16 Base64-encoded characters (using the Base64-encoding algorithm as specified in RFC 1521 [RFC1521]).

Either MD5 [RFC1321] or SHA-1 [NIST95] is used for message authentication codes (MACs). SHA-1 is the mandatory baseline implementation.

Message authentication is also the basis for preventing replay attacks. Aura has presented a set of design principles for avoiding replay attacks in [Aura97]. Essentially, replay attacks can be prevented by binding the messages to their correct context, thus enabling receivers to detect and discard replayed messages. This can be achieved by including enough information into the messages that allow a receiver to correlate them to a specific state of a protocol run, i.e., by tagging messages with an unique identification.

For Mbus sessions that can be quite long-lived, we rely on the sequence number and on the timestamp of the Mbus message header as unique identification characteristics. Both header fields change for every message that is generated by a sending entity. A receiver can thus use this information to correlate a received message to a protocol run context and detect a replayed message, e.g., by comparing the sequence number to the sequence number of the last message that has been received from the respective sender.

Note that even though the sequence numbers of received messages per source do not necessarily have to increase by one for each received message (because the sender may send unicast messages to other entities between any two messages and because message may simply be lost or intercepted), this mechanism still provides sufficient protection against replay attacks, because a receiver would never deliver a message twice. The message authentication provides integrity, i.e., it is not possible for an attacker to change the sequence number unnoticed. Replay attacks *between* sessions cannot be completely excluded, as the sequence number context is local to a specific session for each entity.[1] However, the message timestamp allows receivers correlating messages to a protocol run with respect to the sending time, which reduces the possibilities for an attacker to inject recorded messages from a previous session.

6.2.4.2 Message Encryption

Encryption of messages is optional, which means an Mbus session may be configured not to use encryption. Implementations can choose between different encryption algorithms. The mandatory baseline encryption algorithm is AES (*Advanced Encryption Standard*, [NIST01]). In addition, the following algorithms can optionally be supported: DES (*Data Encryption Standard*, [NIST99]), 3DES (*triple DES*, [NIST99]) and IDEA [Schneier96].

6.2.4.3 Applying Authentication and Encryption to Messages

In the following, we describe the operations that senders and receivers perform with respect to encryption and message digest calculation. In many protocols that employ encryption and authentication by the use of message digests, a digest is calculated for the *unencrypted* message, and the encrypted payload contains the original message and the calculated message digest.

[1] RFC 3259 requires implementations initially setting their sequence number counter to zero. In order to make the protocol more robust against replay attacks, a random initial sequence number would be more appropriate. This will be adopted by a future specification.

The motivation is that a receiver can then decrypt the message and, by re-calculating the digest for the un-encrypted message, verify that the decryption was successful without analyzing the message itself.

For the Mbus protocol, we have deliberately chosen another approach. The Mbus protocol relies on a standardized default multicast address and port number. These parameters are used unless a session is explicitly configured differently. This means, the probability that more than one user on a given network link is using the same transport address configuration cannot be neglected, which is also the reason why authentication is mandatory for Mbus implementations. If encryption is used and a receiver had to decrypt every message first before being able to check the authenticity of the message, many messages would have to be decrypted before eventually being discarded.

For efficiency reasons, we are therefore calculating the digest for *encrypted* messages, thus enabling receivers to verify the authenticity of a message before having to decrypt it first. In order to verify that the message has been decrypted successfully, a receiver can look at the first 4 bytes of the message that is the protocol identifier as described in Section 6.2.3.

A sender performs the following operations:

1. If encryption is enabled, the message is encrypted using the configured algorithm and the configured encryption key. Padding (adding extra-characters) for block-ciphers is applied as described in Section 6.2.4.2. If encryption is not enabled, the message is left unchanged.

2. Subsequently, a message authentication code (MAC) for the (encrypted) message is calculated using the configured HMAC algorithm and the configured hash key.

3. The MAC is then converted to Base64 encoding, resulting in 16 Base64-characters as described in Section 6.2.4.1.

4. At last, the sender constructs the final message by placing the (encrypted) message after the base64-encoded MAC and a `CRLF` (carriage return/line feed).

A receiver applies the following operations to a message that it has received:

1. Separate the base64-encoded MAC from the (encrypted) message and decode the MAC.

2. Re-calculate the MAC for the message using the configured HMAC- algorithm and the configured hash key.

3. Compare the original MAC with re-calculated MAC. If they differ, the message is discarded without further processing.

4. If encryption is enabled, the message is decrypted using the configured algorithm and the configured encryption key. Trailing octets with a value of 0 are removed. If the message does not start with the string “mbus/” the message is discarded without further processing.

Figure 6.15 depicts the operating sequence for sending and receiving an encrypted and authenticated message.

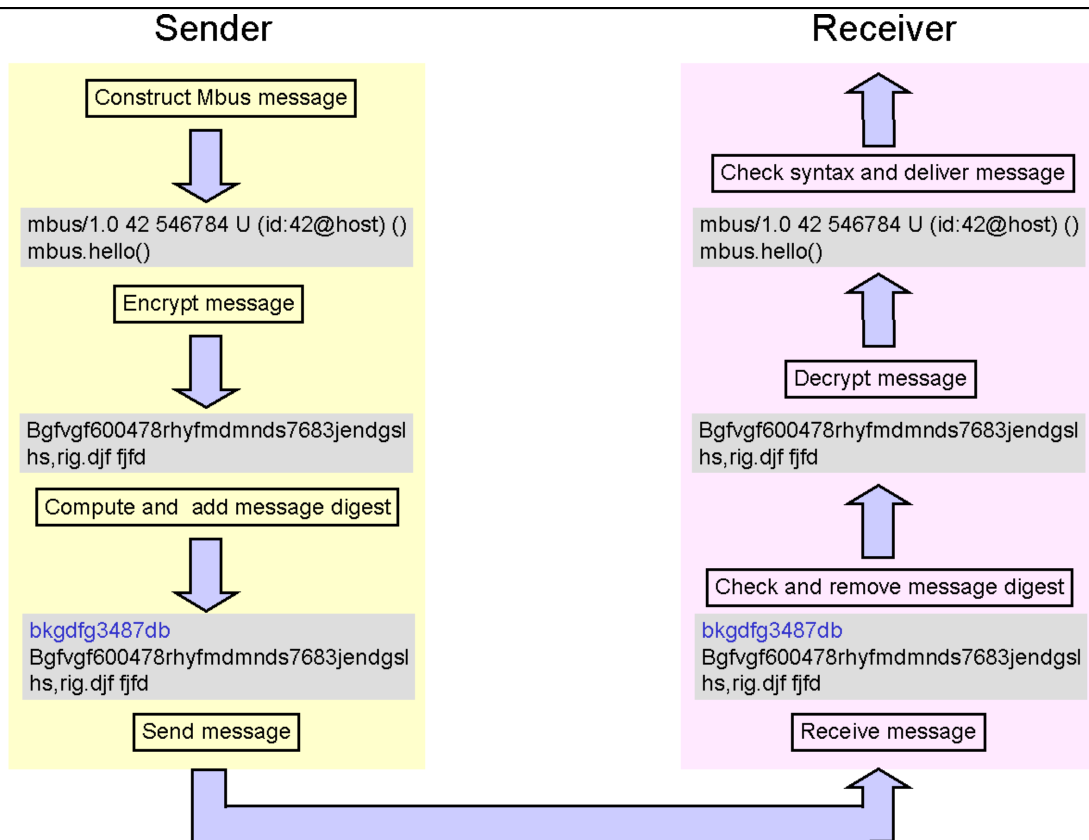

Figure 6.15: Applying the Mbus security functions

6.2.5 Configuration of Mbus Transport Parameters

An Mbus protocol implementation requires a few Mbus session parameters that must be known in advance, i.e., before the Mbus session is joined. Any implementation has to be configurable with the following parameters:

Configuration version: The version number of the given configuration entity. Version numbers allow implementations to check if they can process the entries of a given configuration entity. Version numbers are integer values. The version number for the version specified in RFC 3259 is 1.

Encryption key: The secret key used for message encryption.

Hash key: The hash key used for message authentication.

Scope: The multicast scope to be used for the sent messages.

In addition for these mandatory parameters we have defined the following optional parameters:

IP-address: The non-standard multicast address (IPv4 or IPv6) to use for message transport.

Use of broadcast: It can be specified whether IP broadcast should be used. If broadcast has been configured, implementations use the network broadcast address instead of the standard or a explicitly configured multicast address for all messages that are not sent to a fully qualified destination address.

UDP port number: The non-standard UDP port number to use for sending messages via IP multicast or IP broadcast.

Currently available Mbus implementations obtain these parameters either from a configuration file or from a registry database (e.g., for the Microsoft Windows family of operating systems). RFC 3259 specifies the syntax of the configuration fields. Figure 6.16 depicts an sample Mbus configuration file.

```
[MBUS]
CONFIG_VERSION=1
HASHKEY=(HMAC-MD5-96,MTIzMTU2MTg5MTEy)
ENCRYPTIONKEY=(DES,MTIzMTU2MQ==)
SCOPE=HOSTLOCAL
ADDRESS=224.255.222.239
PORT=47000
```

Figure 6.16: Sample Mbus configuration file

It should be noted that this configuration must a) be known to all anticipated Mbus session members in advance and that b) the configuration cannot be changed *during* a session. In Section 6.4, we present an approach for distributing such as a Mbus configuration, e.g., in order to integrate other entities into a session that do not know the current configuration beforehand.

6.2.6 Mandatory Mbus Commands

The Mbus transport specification defines a set of mandatory baseline commands that are supported by every implementation. Table 6.2 provides an overview of the Mbus commands that are briefly described in Section 6.2.6.1 and Section 6.2.6.2. [RFC3259] provides the complete specification.

6.2.6.1 Commands for Membership Management

The Mbus command for the periodic heartbeat messages is called `mbus.hello`. It is sent to the empty Mbus address, i.e., to all entities, using the time interval calculation rules that are specified in Section 6.2.2.1.

In larger Mbus sessions, the timer interval calculation will yield intervals in the order of several seconds, which is useful for scalability reasons but can delay the coupling of two entities on start-up, e.g., in scenarios, where one entity joins a large Mbus session and is depending on the existence of another entity to continue.

Table 6.2: Mandatory Mbus commands

Command Name	Purpose	Described in
mbus.hello	periodic heartbeat message	Section 6.2.6.1
mbus.ping	explicit member query message	Section 6.2.6.1
mbus.bye	leave indication	Section 6.2.6.1
mbus.quit	termination request	Section 6.2.6.1
mbus.waiting	synchronization request	Section 6.2.6.2
mbus.go	synchronization acknowledgment	Section 6.2.6.2

The `mbus.ping` command can be used to solicit other entities to signal their existence by replying with an `mbus.hello` message. The reply `mbus.hello` message is delayed for `hello_delay` milliseconds, where `hello_delay` is a randomly chosen number between 0 and `c_hello_min` (defined as 1000 ms in [RFC3259]). Several `mbus.ping` messages can be answered by a single `mbus.hello` message: if one or more further `mbus.ping` messages are received while the entity is waiting `hello_delay` time units before transmitting the `mbus.hello` message, no extra `mbus.hello` message need to be scheduled for those additional `mbus.ping` messages. The `mbus.ping` message can be sent to an Mbus multicast address in order to query a group of entities.

When an entity leaves an Mbus session, other entities will notice this by the expiration of their timers for the entity's `mbus.hello` messages. Again, in larger session the timeout will be several seconds and can thus delay an entity that is waiting for another entity to leave. Moreover, depending on the `mbus.hello` timer expiration alone does not allow for distinguishing proper termination of entities and failures such as system crashes or network link failures.

Mbus entities can therefore send the `mbus.bye` message as an indication that they are leaving a session. The `mbus.bye` message is broadcast to all entities. In addition, an entity can request other entities to terminate themselves (and detach from the Mbus) by sending the `mbus.quit` message. Whether this request is honoured by receiving entities or not is application specific. The `mbus.quit` message can be multicast or sent reliably to a single Mbus entity.

6.2.6.2 Commands for Synchronization

The Mbus transport layer provides a simple synchronization mechanism that can be used to synchronize two entities, e.g., at an application start-up phase, where one entity suspends its operation until another entity has signaled that a certain condition has been satisfied.

The `mbus.waiting` message can be sent to a single entity or a group of entities, expressing that the sender is waiting for a certain condition to be satisfied. The condition is specified as an argument of the `mbus.waiting` command (of type `symbol`). The name of the symbol is application dependent. An entity that receives the `mbus.waiting` message and can fulfill the condition specified by the argument returns an `mbus.go` message to the original sender that is sent reliably (using Mbus unicast). The `mbus.go` command provides the same condition symbol as the original `mbus.waiting` message.

The `mbus.waiting` message can be sent as a soft-state message, i.e., it can be sent re-

peatedly until the condition has been satisfied by another entity. For example, an entity that does not know, which other entity can satisfy its waiting condition can simply broadcast or multicast periodic `mbus.waiting` messages until the condition can be satisfied. Figure 6.17 in Section 6.2.7 depicts an Mbus message exchanges, where an entity uses `mbus.waiting` to synchronize with another entity.

6.2.7 Sample Mbus Session

Figure 6.17 depicts a set of Mbus message exchanges between two Mbus entities `(app:video-tool module:control)` and `(app:video-tool module:engine)` during an Mbus session. Note that `mbus.hello` and `mbus.bye` messages are actually broadcast to all entities and not sent explicitly to a single entity.

In this example, the control module is sending an `mbus.ping` to query the engine module. For a better presentation in the diagram, we have provided the destination address of the `mbus.ping` message as a command parameter to clarify that is neither broadcast nor unicast to the engine module. In reality, the `mbus.ping` message does not provide any parameter; instead the corresponding Mbus message simply provides an appropriate destination address. In many cases, this will be a partly qualified address, as the sender will usually not know the complete Mbus address of the desired addressee — otherwise there would be no need to send an `mbus.ping` query.

The engine module answers with an `mbus.hello` message (that would be broadcast to all entities). The control engine sends an `mbus.waiting` indication, expressing that it waits for the `video-device` condition to be fulfilled.

The condition cannot satisfied immediately, so the control module repeats the corresponding `mbus.waiting (video-device)` message until the engine module answers with an `mbus.go (video-device)`. The message exchange provides the regular sending of `mbus.hello` messages by both parties (that would be broadcast).

After some time, the control module requests the engine module to terminate itself by sending it the `mbus.quit` message. The engine module honors this request and sends a `mbus.bye` message to indicate that it is about to leave the Mbus session.

6.3 Higher Layer Interactions

In Section 6.2, we have described the Mbus transport protocol. This includes the basic protocol behavior, representation of messages and message elements and operational aspects, such as Mbus configuration. While it is possible to build applications that layer directly on the basic Mbus transport protocol, we have identified a set of frequently used interaction schemes that we have generalized and described in more detail in a document entitled *Guidelines for Application Profile Writers* [Kutscher01a].

Building on the basic Mbus transport protocol, we have developed and specified a set of higher-layer interactions for peer-to-peer and for group communication in a guidelines documents for Mbus application designers that is intended to be used by applications. For example, we have defined procedures for implementing commonly used interaction models with the Mbus messaging layer, such as RPC communication (see Section 4.1.1) and event subscription and notification.

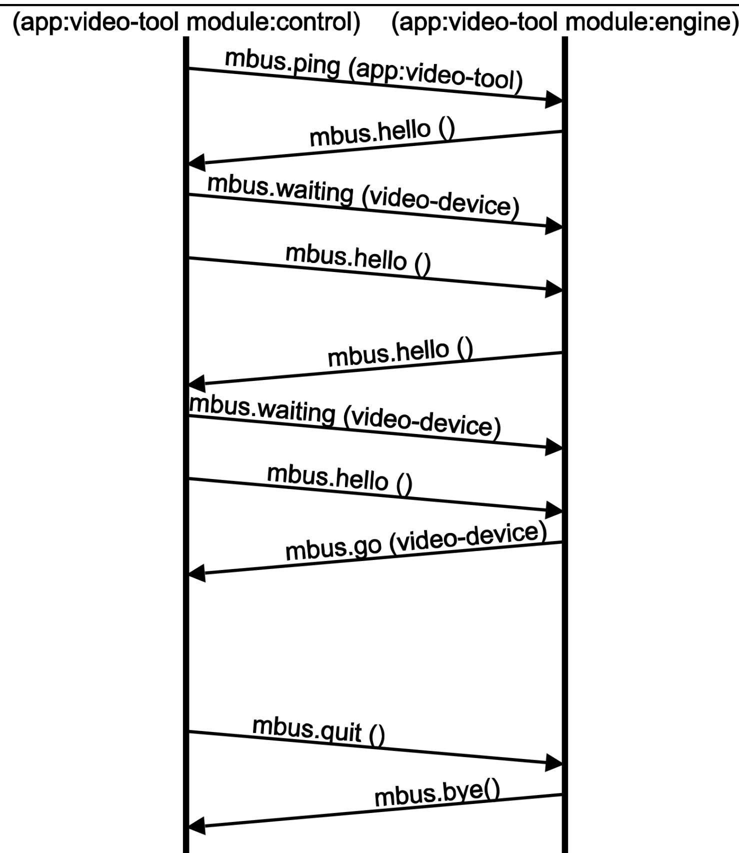

Figure 6.17: Sample Mbus message exchange with synchronization

The main objective for these higher layer services is to provide a foundation for the development of so-called *Mbus application profiles* (*Mbus profiles*): application-specific descriptions of the usage of the Mbus mechanisms that typically provide a list of command definitions, a description of address element semantics etc. For example, we have defined the *Mbus Call Control Profile*, specifying how to interact with call control engines of conferencing systems that we describe in Section 9.3. The *Mbus Guidelines* cover four main aspects:

Terminology and representation conventions: Building upon the representation of values given in the Mbus transport specification we have defined additional representations for more complex data types that are frequently used for the specification of Mbus application profiles. These definitions are documented in [Kutscher01a].

Notational conventions: In addition, [Kutscher01a] provides a set of conventions that can be used for writing *Mbus application profiles*, such as recommendations how to specify the characteristics of an Mbus command.

Usage of Mbus Addresses: In Section 6.3.1, we describe the usage of Mbus addresses and addressing schemes for specific applications and provide a list of conventions that we have followed in specifying application profiles.

Interaction models: The transport specification essentially defines one interaction model for the Mbus: Message passing (with support for group communication). In Section 6.3.2 we describe conventions and protocols for additional interaction models that can be implemented with the basic Mbus message passing mechanisms.

Management of control relationships: In order to increase robustness for Mbus-based group communication scenario, we have formalized the notion of *control relationships*. For example, a service providing entity can require another entity to explicitly register as a *controller* before being allowed to send control commands and to receive event notifications. In Section 6.3.3, we present different types of control relationships and corresponding Mbus commands for managing these relationships.

We define the following common Mbus related terms that are used in the following discussion:

Mbus application profile: An *Mbus application profile* is a specification of Mbus addressing conventions, Mbus commands, their semantics and characteristics for a specific application context.

Fully qualified Mbus address: A *fully qualified Mbus address* is a unique Mbus address for an Mbus session that denominates exactly one specific existing Mbus entity at a given time and can thus not be expanded further.

Mbus addressing scheme: An *Mbus addressing scheme* is a set of Mbus address keys and possible address values. An *Mbus application profile* definition typically provides a specification of a corresponding addressing scheme.

6.3.1 Usage of Mbus Addresses

As discussed in Section 6.2, Mbus addresses are lists of arbitrary key/value pairs and every Mbus entity can choose its own Mbus address. Destination Mbus addresses can be partly qualified to allow for group addressing or selecting receivers by certain application specific key elements that represent a certain application or service type. Except for the mandatory `id-element` the Mbus transport specification (Section 6.2.1.1) does not define any other elements. *Mbus application profiles* define application-specific address elements and describe useful values for the address elements that allow for identifying application components in a meaningful way and allow for group communication between the intended set of application components. For example, for development of Mbus-based conferencing systems, we have defined the address elements

- `app` for denoting the application;
- `media` for denoting the media type (within a multimedia conference) the application is used for; and
- `module` for distinguishing individual modules of a single application component.

In this architecture, a complete multimedia conferencing endpoint would consist of multiple application entities, each of which possibly provides different modules that perform individual functions. For example, the audio tool RAT (see also Section 10.1.2) is an application that consists of three components: an audio data transport and rendering/recording engine (`module:engine`), a user interface engine (`module:ui`) and a coordinating controller (`module:control`). In addition, all three components provide the address elements (`app:rat media:audio`). In a conferencing system that employs this addressing concept, it is possible to rely on Mbus group communication mechanisms, e.g., for the following purposes:

- a destination address of `(module:engine)` can be used to send a message to all transport and rendering engines, e.g., for synchronizing playback or for implementing floor-control;
- a destination address of `(module:ui)` can be used to send a message to all user interface components, e.g., in order to coordinate graphical user interface components for providing an integrated user interface;
- a destination address of `(app:rat)` can be used to send a message to all components of the RAT application, e.g., in order to coordinate these components by a super-ordinate controller.

Figure 6.18 depicts a fully qualified Mbus address and three partly qualified Mbus addresses that could be used to communicate in a corresponding Mbus session.

```
(conf:test media:audio module:engine app:rat id:42-1@10.10.1.1)
(media:audio module:engine)
(module:engine)
()
```

Figure 6.18: Sample fully qualified Mbus address and three partly qualified Mbus addresses

In addition to the addressing concept itself, a profile definition would typically also associate interface definitions to address elements, e.g., by specifying a set of mandatory commands that must be understood by all components of type `(module:engine)`. As depicted by the example above, Mbus application profile definitions typically specify a set of useful address element names and values for a specific application context. These address elements might be used to offer a particular service to other entities and to disambiguate entities sufficiently. Address elements might also be used to express membership of a certain Mbus address group. When it is known that an entity will always send certain messages to a specific address group, an entity will have to provide the corresponding address elements (with proper keys and values) to become a member of that group. Note that while a single Mbus entity is bound to exactly one Mbus address, an application program may choose to instantiate several Mbus interfaces, i.e., appear as multiple Mbus entities on the Mbus. In summary, Mbus addresses can be used for the following purposes:

1. to signal affiliation to an application context;

2. to offer a certain service; or

3. to receive messages for a certain subgroup (to *tune* into a specific sub-channel on the Mbus).

It is possible that, for a given application context, not every address element is to be used by every involved Mbus entity. Instead some elements (or values) might be reserved for use by *service providing entities* while others might be required in order to receive messages that are addressed to a certain subgroup. For example, an Mbus entity could be configured to always send event notifications to a specific destination address, e.g., `type:client`. In order to use this entity in new application contexts and in order to receive corresponding event notifications, interested receivers would have to adopt this address element for their own Mbus address.

Moreover, it should be noted that it might make sense for entities to adopt more than one command profile and thus make use of more than one addressing scheme. An entity could provide all address elements that are required by command profile `A` and additionally provide all the required elements for profile `B`. For example, if an audio engine module in a conferencing application does not only support the audio-specific Mbus command interface but also an *RTP Mbus interface*, it is conceivable that this entity provides the address elements for conferencing applications and additional elements that are specified in a corresponding *RTP Mbus profile*. Of course, it is also conceivable that such an application component has multiple Mbus interfaces with unique addresses (depending on the specific application requirements). The way the Mbus addresses are used for an Mbus-based application is described by an *Mbus addressing scheme*. In the following, we present a list of guidelines how to specify an Mbus addressing scheme for an Mbus application profile definition. In addition to these notational guidelines, we have provided the following algorithms for processing Mbus addresses in applications or protocol implementations:

Aggregation of Mbus Addresses: In an Mbus session Mbus entities learn of the existence of other entities by the entity awareness mechanisms specified in Section 6.2.2. An implementation can thus maintain a set of Mbus addresses (and, potentially, the mapping to endpoint addresses). In order to send a message to a subset of all Mbus entities, an implementation has the following options:

- It can send the message via Mbus-broadcast to all entities, taking into account that the message is received and potentially processed by entities that are not a member of the subset;
- it can send the message via Mbus-unicast to each entity that is a member of the desired subset, taking wasted bandwidth and (potentially) additional processing times for receiver-based filtering (if the messages are not sent via IP unicast) into account; or
- it can determine a single Mbus group address that represents the entities of the subset and send the message once using this group address as a destination address.

Determining a single Mbus group address can be done by *aggregating* a set of Mbus addresses into a single address that represents the original sets through wildcarding. Essentially, such an address can be obtained by determining the intersection of the input Mbus addresses, each of which is a set of address elements. Figure 6.19 depicts an algorithm that can be used to aggregate an arbitrary number of Mbus addresses into a single address. The C++ function `aggregate` takes an iterator range as an input parameter (the range denotes a collection of Mbus addresses) and counts the occurrences of all Mbus address elements in the Mbus addresses. Only those address elements that occur in every Mbus address are adopted for the resulting Mbus address.[2] Note that this algorithm can generate Mbus addresses representing more Mbus entities than specified by the iterator range. For example, if the addresses to be aggregated do not provide at least one common Mbus address element, this algorithm will yield the empty Mbus address `()` as a result.

An extended variant of this algorithm (not shown here) could determine a list of addresses as a result, where the list would describe the desired subset of entities unambiguously. For a set of addresses that do not provide a single common address element, this algorithm would yield the input set of addresses as a result.

```
template <class Input>
MAddress aggregate(Input start, Input end)
{
  typedef map<MAddressElement,int> elements;
  elements ae;
  int count=0;
  MAddress res;

  // get all address elements:
  for(;start!=end;start++) {
    count++;
    for(MAddress::Const_Iterator ai(*start); ai; ++ai) {
      ae[*ai]++; // count occurences of AddressElements
    }
  }
  // keep all elements that occured in every address:
  elements::const_iterator ei;
  for(ei=ae.begin();ei!=ae.end();ei++) {
    if(ei->second==count) {
      res.setElement(ei->first.key(),ei->first.val());
    }
  }
  return res;
}
```

Figure 6.19: C++ function to aggregate a set of Mbus addresses into a single Mbus address

Expansion of Mbus Group Addresses: Applications or protocol implementations that want to send a message via an Mbus-unicast to a group of entities that is represented by an Mbus group address, i.e., a partly qualified address, need to determine the set of fully

[2] We have chosen this iterative algorithm because it has shown to be more efficient than a recursive algorithm, e.g., using the C++ standard library function `set_intersection`.

qualified Mbus addresses that is represented by the group address. An application of a corresponding algorithm could be the implementation of a (simplistic) reliable Mbus-multicast protocol, based on a sequence of reliable Mbus-unicast messages.

Figure 6.20 depicts an algorithm that can be used to expand an Mbus group address to the corresponding set of fully qualified Mbus addresses enclosed within the group address. The C++ function `expand` takes the group address (`a`) and an output iterator (`out`) for the result addresses as arguments. The current set of known entities is stored in the variable `entities`. All the addresses in `entities`, for which `a` is a subset, are adopted for the resulting set of fully qualified addresses.

```
typedef std::map<MAddress, Entity*, maddress_less> EntityMap;

EntityMap entities;

template<class Output>
inline int MBus::expand(const MAddress& a, Output out) const
{
  EntityMap::const_iterator start;
  int total=0;

  for(start=entities.begin();start!=entities.end();start++) {
    if(a.isSubsetOf(start->first)) {
      *out++=start->first;
      total++;
    }
  }
  return total;
}
```

Figure 6.20: C++ function to expand an Mbus group address into a set of corresponding fully qualified addresses

6.3.2 Interaction Models

The general semantic model for Mbus commands is that commands are sent as payload of messages from one peer to another receiving (group of) peer(s) in order to trigger some kind of operation on the receiving side or to distribute the current soft-state of a given variable. On a low level of abstraction, every Mbus command can be modeled like this. However on a higher level of abstraction some classes of commands can be identified that are used to implement specific Mbus communication scenarios. The following list describes these command classes briefly:

Remote commands: Remote commands are used to trigger an asynchronous operation on the target system. The command has a name that is associated with a certain operation on the receiving side and can be sent together with a list of arguments (that can be empty) that are interpreted by the receiver. The name and the type definition of the command are specified in an Mbus application profile definition. See Section 6.3.2.1 for a discussion of generic remote commands.

Mbus Remote Procedure Calls: RPC-commands allow associating an operation at a remote entity with an Mbus command and are used when a caller expects a result message from

the callee that can return result parameters of the remote procedure/function call. We have defined different types of RPC-commands that are discussed in detail in Section 6.3.2.2.

Transactions: Transactions are similar to remote commands because they are also used to trigger a remote operation. Additionally, transactions are used in scenarios where the caller is interested in how/whether the remote operation has been performed. In general, characteristics of transactions are atomicity (recoverability), consistency, isolation (serializability) and durability (see Section 4.1). We have defined a protocol that supports atomicity, consistency, and isolation. Durability cannot be provided by the protocol itself, i.e., applications have to provide additional mechanisms to guarantee durability. See Section 6.3.2.3 for a detailed discussion of transactions.

Properties: Obtaining the value of a named property of another Mbus entity is a variant of an RPC-style command: One command is sent that represents a query and one command is returned to the caller that contains the value. Setting the value of a named variable is a simple remote command with a parameter for the new value.

Event notifications: An entity that frequently sends messages to inform other entities of certain events sends a command for each state change (or after a certain interval) to a (group of) receiver(s). These commands are similar to the simple remote commands because they are also sent asynchronously.

Contexts: Instead of short time interactions between entities that can be accomplished by RPCs and other command classes, contexts allow for more persistent relationships between entities. Contexts are scopes for coherent commands that are to be exchanged within a long-term interaction. Contexts provide the service of assigning a name to an interaction context that allows disambiguating Mbus interactions that use the same commands but refer to different contexts at the same time.

Note that, in the following, we will only describe Mbus RPCs and Transactions in detail. For a description of Mbus Properties, Event Notifications and Contexts, the reader is referred to [Kutscher01a].

6.3.2.1 Remote Commands

Simple remote commands (that do not belong to any of the other classes) require no special procedures or conventions besides the general recommendations for Mbus command definitions: They are defined in self-contained profile definitions, where their applicability, the command name and the command arguments are documented.

Figure 6.21: Sample Mbus remote command interaction

Figure 6.21 depicts a sample Mbus command (the Mbus message headers are not shown). The command `tools.rat.volume()` could be used as a soft-state update of the current output volume and could be sent periodically to a sub-group of entities in an Mbus session. The message depicted in Figure 6.21 is an Mbus unicast message, however remote commands can also be sent to entity groups.

6.3.2.2 Mbus Remote Procedure Calls

There are Mbus commands that are not intended as soft-state updates, which are sent either on request or without solicitation to a group of potentially interested receivers. It is often required to trigger a well-defined remote operation on a well defined Mbus entity. Mbus commands that are used to trigger remote operations on a receiver's systems fall into the class of RPC communication mechanisms that we have described in detail in Section 4.1.1. For the development of Mbus-based applications, we have identified the following characteristics of Mbus commands that are used to trigger remote operations:

- There is a *static binding* between a command's name and the associated operation. A corresponding Mbus command name is associated with a well-defined effect that would, for example, be defined in an Mbus profile definition.
- A sender must be able to send parameters together with the command name.
- A sender is interested in the result of the remote procedure invocation. There are different classes of results that have to be distinguished:
  - The Mbus does not rely on a strict static binding of command names to procedure implementations as, e.g., ONC RPC does (see Section 4.1.1). Instead, a sender cannot verify that another entity really provides a procedure implementation for a given command name, unless it tries to invoke the corresponding procedure. This means, a sender has to be informed whether a remote procedure that has been specified in a corresponding Mbus message has been called at all or whether it is not associated with a procedure implementation.
  - When the remote procedure has been called, the sender is typically interested in results of the invocation. The results can be classified as *status information* and other result parameters. For example, in a database query, the procedure would yield a status information indicating that the query has been successful and the requested database record as additional parameters.
- The Mbus provide a completely *asynchronous* message passing service. Because in asynchronous communication scenarios, a request message is not necessarily immediately followed by a corresponding response message, there is a need to *correlate* RPC answers to the original requests. For example, when a sender sends out multiple RPC messages to possibly different entities, there must be a way for a sender to correlate a received answer to the correct command.

During the development of different Mbus-based applications, we have learned that many Mbus commands with RPC-semantics provide these properties. These commands are often

implemented as an Mbus command pair, i.e., one command for the remote procedure invocation and one command for sending back results and status information, where the answer provides some information that allows to refer to the original request. We have therefore generalized this behavior and defined an Mbus command class named *Mbus RPCs*. Mbus RPCs are implemented by a command-pair: One command (with arguments) triggers the remote procedure call and one command represents the result. There are RPCs for Mbus "point-to-point" communication and RPCs for Mbus group communication.

RPCs for Unicast Communication

RPCs commands are regular Mbus commands with a special parameter list. The first parameter of an RPC command is a protocol management list that is used for specifying the RPC type (`UNICAST, ANYCAST` etc.) and a unique RPC ID for identifying RPCs and for correlating answers to requests. Unicast RPCs are sent using reliable Mbus messages (Section 6.2.1.5). Multicast RPCs are described below.

The names of commands for returning RPC results are constructed using the name of the request command and appending the string `.return`. The first parameter of an RPC return command is also a management list. In addition to the RPC-ID the list provides general RPC status information (whether the operation has been performed at all). The second argument of an RPC return command is of type list and is a list of return parameters of the invoked procedure. The return command is typically sent via reliable transport; however, the Mbus RPC specification does not require this. Figure 6.22 depicts a sample Mbus RPC interaction (the Mbus message headers are not shown).

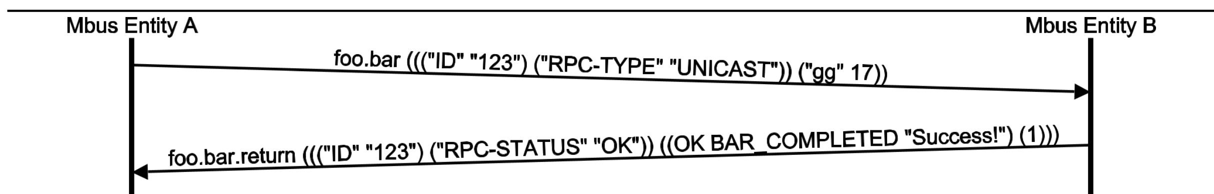

Figure 6.22: Sample Mbus RPC interaction

Although the processing of Mbus RPCs and answers to RPCs is implementation specific, it can be expected that an RPC message is treated like any other message that is sent to the Mbus: The message is sent asynchronously (using reliable message transport) and the protocol implementation (respectively the application) continues with its normal operation, e.g., sending of `mbus.hello()` messages, receiving and delivering other messages etc. until the answer to the original RPC message has been received, which is then correlated to the request and delivered to the application. It should be noted that, in the presence of *Mbus unicast optimization* (Section 6.2.1.4) and *acknowledgment piggybacking* (Section 6.2.1.5), the message exchange for an Mbus RPC is quite efficient:

1. The sender sends the RPC in a reliable message to the destination address. The rules for sending messages reliably specify that the destination address must be fully qualified and thus denote a single entity. In this case, the *Mbus unicast optimization* applies and the message is sent as a reliable Mbus message via IP unicast to the destination entity.

2. The destination entity receives the message. The acknowledgment rules for reliable message transport require the entity to send back an acknowledgment. However, the optional *acknowledgment piggybacking* rules allow for deferring the sending of the acknowledgment for a short time. After the protocol implementation has delivered the RPC to the application and the procedure has been called, the RPC return message is generated and sent to the original sender.

3. Because the return message is by definition addressed to the original sender only, the *Mbus unicast optimization* applies again and the message is sent via IP unicast to the original sender. In addition, the *acknowledgment piggybacking* mechanism applies as well, as the return message will (probably, depending on the implementation) be the first unicast message that is sent to the original sender after the reliable RPC message has been received. As a result, the acknowledgment is piggybacked onto the RPC return command.

Summarizing, we can state that an Mbus RPC interaction can effectively be realized by a single message exchange, provided that an implementation supports the optional mechanisms for unicast transport and acknowledgment piggybacking. In Section 8.2, we validate this proposition by some simulation results.

Although the RPC paradigm is a fairly simple model, there are issues with *idempotency*, especially in the presence of asynchronous communication and unreliable message transport.[3] In general, remote procedure calls cannot be treated as function calls in the mathematical sense that perform a calculation that is solely based on the function arguments and does not generate any side-effects. Instead, remote procedure calls are typically dependent on state within their execution environments and can change this state during their execution. As a result, it can mean a huge difference whether a remote procedure in invoked once, twice or not at all. For example, let us consider a remote procedure that is used to initiate a call at a call signaling engine of a distributed IP telephony endpoint: a user who issues a corresponding RPC would certainly want to establish exactly one call to the specified callee, not multiple (or none).

Another important property in this regard is *consistency*: not only do we want to exclude the possibility that an RPC is called multiple time accidentally, but we also want to ensure that both caller and callee have the same view about the number of invocations. For example, a caller who issues the call initiation RPC has to know whether the corresponding message has resulted in a call set-up or not.

When we analyze the Mbus RPC mechanism with respect to these requirements, we can make the following observations. Note that the term *RPC message* refers to an Mbus message containing an Mbus RPC command, whereas the term *RPC command* refers to the Mbus command of an Mbus RPC message that is structured as described in the aforementioned rules for Mbus RPC communication.

- In general, the Mbus message sequence number prevents message duplication and unintended multiple invocations of a remote procedure.

- In the presence of transmission failures for the RPC message, the Mbus reliable transport mechanisms would trigger re-transmissions of the RPC message. Even if a retransmission is caused by an acknowledgment that has been sent but not been received, the Mbus

[3]It should be noted that these problems do not only apply to RPC communication, but to any interaction type, where requests are sent that are associated to some form of remote operation.

message identification as describe above would prevent multiple deliveries of the corresponding message.

- In case the RPC message cannot be delivered at all, i.e., when all re-transmissions have failed, the sender application will be notified by the Mbus transport layer and the RPC can be classified as failed.
- Only in cases where the RPC message *has* been received at the callee but the caller has received none of the acknowledgments, inconsistencies can arise. In this case, the caller will assume that the RPC message transmission has failed and that the corresponding procedure has not been invoked, whereas in fact in has been invoked, but the result message (and the acknowledgment) have never arrived at the caller's side. In this case, the caller will retransmit the RPC message but the callee will ignore each of the retransmissions because the message has already been received and processed. In these scenarios, sending the RPC return command in a reliable Mbus message can increase the reliability. In this case, the RPC callee can at least verify that a return command *has* been received. (It can *not* definitely tell that it has *not* been received when this command has not been acknowledged, because of the same difficulty in determining failure situations on the basis of missing acknowledgments in two-way-handshake scenarios.
- In those cases, where inconsistencies arise because a caller has considered an RPC command as failed, as the return command and/or the acknowledgment has not been received, robustness can be increased by imposing the following additional rules:
  1. An RPC callee maintains a list of received (and processed) RPCs by storing their identifiers.
  2. An RPC caller that has considered a sent RPC command as failed because no return command or acknowledgment has been received, uses *the same* RPC identifier in subsequent attempts to invoke the RPC.
  3. The callee notices the new attempt by comparing the RPC identifier with the ones that are stored in its RPC ID table. When the callee detects a new attempt to invoke an RPC that has already been invoked, it notifies the caller by sending a corresponding RPC return command.

  This enhancement is essentially a second layer of duplicate detection and does of course not solve the fundamental problem that we have to face here: the difficulty of reaching consistency of a message delivery status in asynchronous communication scenarios with a limited number of handshakes. We have not implemented this last enhancement but rely on another strategy for the detection of communication problems in the context of RPC communication: In cases where a series of retransmissions or retransmitted acknowledgments cannot be delivered, there is a high probability that the communication between the two involved entities itself is disturbed. An Mbus entity is likely to notice this on the basis of irregularities in the reception of the periodic `mbus.hello()` messages and can thus notify the application that in turn can run specific connectivity tests (e.g., by the means of `mbus.ping()` messages) or terminate the session completely.

In Section 6.3.2.3 we present another approach for increasing the robustness for RPC-based communication in Mbus sessions.

RPC Communication with Multiple Entities

Relying on the general model of remote procedure invocation, where an Mbus command is used to trigger a remote operation and a corresponding return command needs to be correlated to the original RPC command, we have considered different scenarios for RPCs that are not addressed to a single entity, but to a group of entities. This means, we are using the general Mbus RPC command syntax, but deviate from the principle that an RPC can only be sent to exactly one entity. We distinguish three different command classes for RPC communication with multiple entities: *Anycasts*, *Group RPCs with multiple responses*, *Group RPCs with a coordinated result*:

Anycast: In general, *anycast* is communication between a single sender and the topographically nearest of several receivers in a group. For example, IPv6 has an anycast mechanism to locate routers that are closest to a given host [RFC2373]. For Mbus communication, we have adopted the *anycast* paradigm as follows: A sender of an anycast RPC wants the corresponding operation to be performed by *any* of the addressees, and it does not care which particular entity of the destination address group performs the operation. However, the RPC shall be invoked exactly once. The representation of the RPC and the return command is the same as for the unicast RPC model, however the exact procedure differs as follows:

- The RPC command is sent in a message that is addressed to a group of receivers, and as multicast message, it is sent using unreliable Mbus transport. The destination address should represent at least one entity in the current Mbus session.
- The management information list of the RPC command provides an entry with key `RPC-TYPE` and value `ANYCAST`.
- Those of the receiving entities that want to respond to the RPC and are able to perform the requested operation return a `standby` command in order to signal their disposition to provide the service. The name of the command is the RPC command name concatenated with `.standby`. The first argument is again a management information list that contains the original `RPC-ID`. The destination address of this command is an aggregate of the sender address and the destination address of the RPC and is therefore sent unreliably. See Section 6.3.1 for a description of an address aggregation algorithm. It should be noted that *suppression* of standby commands is not provided by the Anycast RPC protocol; instead callers are required to send Anycast RPC to a sufficiently qualified destination address that limits the group of receivers to an appropriate size (which is application-dependent).
- After a timeout `T_anycast`, the entity that originally sent the RPC message selects one of the entities that answered with a `standby` command and sends the RPC again (in a new message). This message is sent using reliable Mbus message transport. The management information list of the command contains an additional entry with a key `REFERENCE`. The value is the sequence number of the received standby message.
- The entity that receives the second RPC message now operates as specified for the regular unicast RPC case.

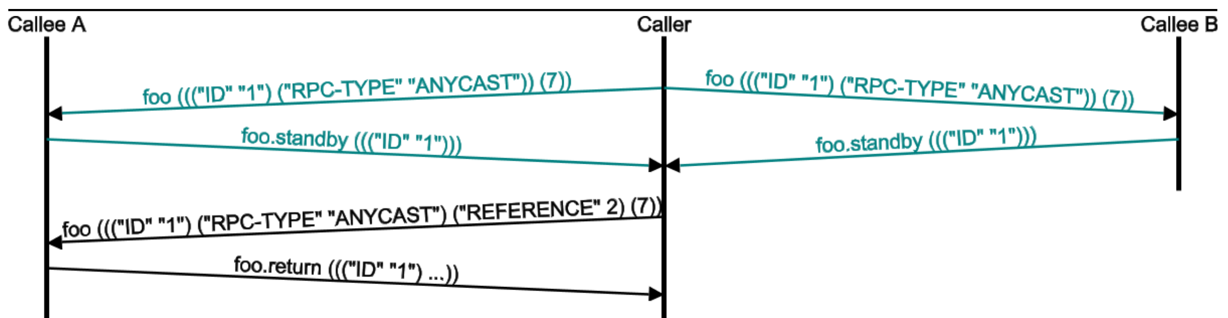

Figure 6.23: Sample Anycast interaction

Figure 6.23 provides an example of a simplified anycast interaction. Note that the first RPC message and the corresponding `standby` messages are actually multicast messages. The final `return command` from callee B is abbreviated.

One RPC, more than one result command: While Mbus RPCs and Mbus Anycast RPCs are used to trigger exactly one invocation of a remote procedure, we also allow for RPC commands that are sent to a group of entities and that trigger a procedure invocation at each of the receivers. In essence, the interaction model is similar to a *Remote Command* (Section 6.3.2.1) that is multicast to an entity group, where each of the receivers responds to the original sender by sending a unicast message with the individual result. The difference to this latter model is that we use the Mbus RPC command syntax in order to correlate responses to requests and to represent parameter and result lists. We have defined the following conventions for RPCs that are sent to a group of entities where each entity responds independently:

- The sender uses a group address as the Mbus destination address of the RPC message.
- The command meta-information list provides an entry with key `RPC-TYPE` and value `MULTICAST`.
- The sending entity sends the RPC in a message addressed to an Mbus address group using unreliable Mbus message transport and calculates the set of unique Mbus addresses (using the algorithm described in Section 6.3.1) of the entities that are enclosed in the destination address group. This list is used to determine the set of Mbus entities that should answer the message.
- The receiving entities operate as specified for the regular unicast RPC case, i.e. they try performing the operation and report the results to the sender of the RPC. The destination address of the result message is the address of the sender, and the message is sent reliably.
- After an application dependent timeout, the entity that originally sent the RPC evaluates the received results. The procedure of how return parameters are gathered, collapsed and presented to the user is application/implementation specific. In addition, the application specific rules have to define, how many responses have to be received in total. For example, not every intended receiver may have received the

request message, and not every receiver may support the corresponding operation. For some application scenarios, it can be sufficient to receive at least one response, whereas for others, all addressed Mbus entities should respond (based on the address list generated at the time of sending).

In order to have each entity of a group of entities invoke a remote procedure, a calling application could alternatively determine the set of Mbus entities that are described by a group address and send an RPC `UNICAST` message to each of the entities. While this imposes the overhead of sending each RPC multiple times, it also provides a higher degree of reliability and a more precise control of the effective group of receivers:

- each RPC message is sent independently, using reliable Mbus transport; and
- the caller can identify the set of receivers precisely, where for `MULTICAST` RPCs that rely on receiver-based filtering, it is always possible that an entity joins or leaves the destination group just after the RPC message has been sent. This does not necessarily have to be problematic, but applications have to be prepared that the exact group of receivers cannot be determined.

In summary, `MULTICAST` RPCs are a simple adaptation of the RPC command syntax to Mbus multicast scenarios and provide a less reliable way to invoke remote procedures. For reliable remote procedure calls at multiple entities, the sending of multiple `UNICAST` RPC messages is preferable, albeit less efficient.

One RPC, coordinated result: We have defined one other class of RPC-based interactions that also relies on the sending of RPCs to a group address: the RPC class `COORDINATED`. For this type of RPC interaction, a sender sends an RPC message to a group of entities and expects each entity to perform the operation but expects only one result message that represents all results of the addressed group. For example, this RPC class could be employed by a controller in a signaling gateway that wants to decide how to process an incoming call. The signaling gateway provides a set of *policy modules*, each of which can make independent decisions based on an individual rule set. Instead of locating each policy module, issuing a `UNICAST` RPC and evaluating the results of all RPCs, the controller could send a `COORDINATED` RPC to the group of all policy modules in the system, say to the Mbus address `(module:policy)`, and have the policy modules coordinate themselves and provide the caller with an aggregated result of all RPCs.

This approach simplifies the development of components such as the controller in the example, because the application logic for distributed decision making does not have to be hard-wired into the controller (i.e., the component that is only interested in the decision) but can be provided by the components that have a knowledge of the RPC's semantics anyway. The operations that are associated with `COORDINATED` RPCs are typically operations that generate an "interesting" result. They may have side effects, but these side effects are typically not the main reason for issuing the corresponding RPC. Instead, they can be viewed as *functions* that execute some internal operations in order to generate a result that is then processed in the coordination and aggregation step.

The procedure for processing and forwarding the results of multiple RPCs to the original caller is highly application specific. For example, if we assume that the afore-mentioned

policy modules each generate a single Boolean value, where `true` indicates that the incoming call should be accepted and relayed and `false` indicates that it should be canceled, the composition of multiple RPCs would probably be a Boolean operation on the results. In this case, the final (coordinated) result could be the disjunction of the individual results, which means that the aggregated result would be `true` as long as one policy module returned `true` itself. If the result is determined by a conjunction, one would effectively implement a *veto model*, because the result would be `false` as long as only one policy module returned `false` itself.

For other applications and other data types, other operations would be required. The interesting feature of `COORDINATED` RPCs is that the caller does not have to know these operations as the coordination is done in a distributed fashion among the modules that provide the RPC service. We have defined the following conventions for RPCs that are sent to a group of entities where each entity performs the operation but only one result message that represents all results of the addressed group is sent to the original sender of the RPC:

- The sending entity sends the RPC in a message addressed to an Mbus address group using unreliable Mbus message transport.
- The command meta-information list provides an entry with key `RPC-TYPE` and value `COORDINATED`.
- The receiving entities try to perform the operation and then send intermediate result commands to a coordinator in an Mbus unicast message using reliable transport. After a timeout `T_Coordination`, the coordinating entity aggregates all intermediate results and sends an aggregated RPC-result message to the original sender. The coordinator does not send an intermediate result.

  The coordinator sends the aggregated result regardless of the number of intermediate result messages it has received, i.e., it does not verify that every entities that is a member of the destination group sends a result message. If no intermediate result message has been received, the coordinator sends the final result based on its own result of the procedure invocation.
- The coordinator is determined implicitly by sorting the entities of the destination group with respect to a lexicographic order of their Mbus addresses (see below for a discussion of consistency issues with this approach). Each of the recipients can determine the set of destination entities by applying the address expansion algorithm of Figure 6.20 to the destination address of the RPC message. The first entity in this address list is the coordinator. The lexicographic order of Mbus addresses is defined by sorting the address elements of an Mbus address lexicographically, concatenating the address elements into a single character string, and then sorting these character strings lexicographically.
- As described above, the rules for processing the intermediate results are application specific.

Figure 6.24 depicts a sample `COORDINATED` RPC interaction. The initial RPC request is multicast to all entities. The entity `name:A` is chosen as a coordinator because it is

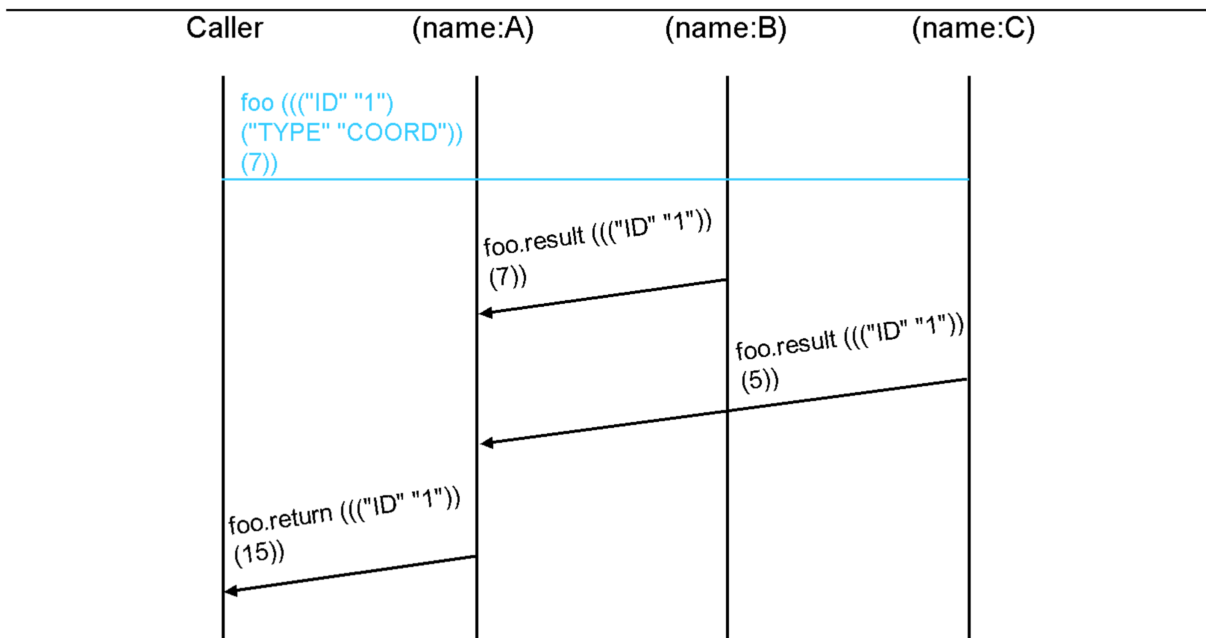

Figure 6.24: Sample coordinated RPC interaction

the first entity in the lexicographically sorted list of entities in the destination group. The entities `name:B` and `name:C` send their intermediate results to `name:A` that aggregates the results and sends a single result in a single RPC return message to the original caller. In this example, the aggregation algorithm would be the addition of integer values. We assume that `A` has obtained `3` as a result of its own procedure invocation and adds this to the results `7` and `5` that have been received from `B` and `C`, thus resulting in `15` for the final return message.

It should be noted that `COORDINATED` RPCs are as sensitive to changes in the group membership as `MULTICAST` RPCs. For example, the joining of a new entity during the `COORDINATED` RPC procedure can lead to inconsistent views about the coordinator. As a result, we recommend that this mechanism be only used in situations where the session membership is stable. In many applications, entities are started at the session initiation, so that the membership can be considered stable, unless communication failures, link partitions or similar problems arise.

In addition, it should be noted that the current protocol for coordinated RPCs relies on an optimistic model with respect to transmission of intermediate results. Because the coordinator will send the final result in any case, i.e., regardless of the number of intermediate results, the original sender cannot determine the number of contributing entities. The coordinated RPC mechanism is intended as a simple way to send a coordination request to a group of entities, where the exact group membership does not have to be known to the sender. Hence, we assume that for many cases, the sender would not be interested in the concrete set of destination entities anyway. If this assumption does not hold, the sender should rather use unicast RPCs that are explicitly addressed to a single destination.

6.3.2.3 Mbus Transactions

`UNICAST` RPCs provide point-to-point communication for invoking remote procedures at a remote Mbus entity. In Section 6.3.2.2, we have described some issues that have to be taken into account when using this interaction model:

1. There is no static binding between a procedure "stub" at the caller and the remote procedure at the callee. Entities that want use to `UNICAST` RPCs rely on informal interface descriptions and cannot verify in advance (i.e., before calling a certain RPC) that the remote entity supports the RPC.

2. In the presence of transmission failures, consistency problems can occur (this can be amended by increasing the number of handshakes).

Mbus Transactions are a command class that provides additional mechanisms to improve the robustness and possibilities for failure detection for remote procedure calls. Mbus Transactions rely on the general RPC model of a command/response pair, and use the same RPC message syntax. However, we have decoupled the receiving of an invocation request from the invocation of the procedure on the receiver. Mbus Transactions provide an additional handshake after the caller has sent the invocation request. In this handshake, the callee can confirm that the requested procedure is supported and that the current disposition of the callee allows for invoking the procedure. The original caller can answer this confirmation by sending a *commitment*, i.e., a final request to invoke the procedure. Alternatively, the caller can also cancel the operation, which prompts the callee to abort the transaction without invoking the procedure.

Transactions are implemented by defining a command that triggers an operation and an additional acknowledgment command that is sent after the operation has completed (or failed). Acknowledgment commands refer to the initial trigger command and this relation is expressed by a special reference parameter that is generated by the caller. Similar to RPCs, the acknowledgment command is different from acknowledgments on the Mbus transport level: Those only ensure that messages are really received by the addressees, whereas transaction acknowledgments inform the original caller about the result of a certain operation that the callee should have performed upon reception of the transaction command. Transaction commands are only allowed for unicast messages, they may not be sent to an address group. They are sent using reliable Mbus messages. Senders of transaction commands are called clients; receivers of transaction commands are called servers.

It should be noted that means for concurrency control, e.g., to achieve consistency in the presence of parallel transactions, have to be provided by the application itself and are not part of these conventions. Transactions follow the model we have presented for RPCs, i.e., they provide an initial request command and a corresponding acknowledgment, but require additional commands for implementing the commit and rollback functionality. We have defined the following conventions for transaction commands:

A transaction has the same structure as an RPC command, i.e., it provides a management list providing a request ID and a parameter list providing transaction parameters. After receiving a transaction command, an entity responds with an acknowledgment. Acknowledgment command names are constructed using the name of the request command and appending the string `.ack`. The first parameter of a transaction acknowledgment is again a management list providing the request ID. Any action that may be performed by the receiver must be reversible

and should only be executed in a non-reversible way after a commit command is received for the corresponding transaction. If a cancel command for the corresponding transaction has been received before a commit command, the entity will rollback any performed actions. In addition, the entity will also rollback any performed action if no commit message has been received after a certain timeout (which is application-dependent and should be specified in the description of the corresponding transaction command).

After a transaction command has been sent, the sender can either cancel or commit the transaction: A transaction cancel command consists of the original transaction command name and an appended `.cancel` as a command name and provides the original transaction id as a management information parameter. A receiver will cancel or rollback any actions initiated by the original transaction message after receiving a transaction cancellation and delete any state related to the transaction. A transaction commit command consists of the original transaction command name and an appended `.commit` as a command name and provides the original transaction identifier as a protocol management information parameter. A caller can send the commitment after having received the acknowledgment from the callee. A receiving entity finishes the outstanding action initiated by the original transaction command after receiving a transaction commit command and deletes any state related to the transaction. After a commit has been received, cancel commands for the corresponding transaction are not honored anymore. After receiving the commit message, the entity that performs the transaction sends a final acknowledgment constructed from the original transaction command name and an appended `.completed`.

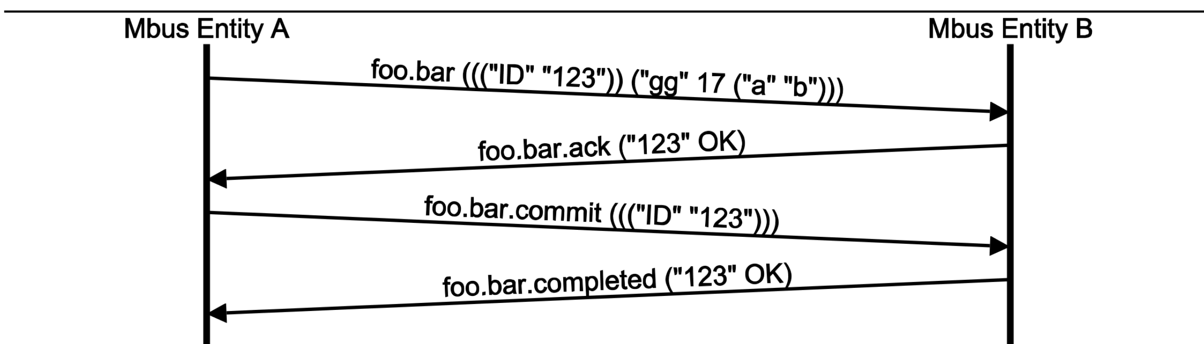

Figure 6.25: Sample Mbus transaction

Figure 6.25 depicts a sample Mbus transaction (the Mbus message headers are not shown), and Figure 6.26 depicts a canceled Mbus transaction. With respect to the *ACID properties* described in Section 4.1, Mbus Transactions can be assessed as follows:

Atomicity: In group communication scenarios, where an entity (“sender”) wants a group of entities to perform a certain transaction atomically, it can send each of the entities the transaction command. After having received the corresponding acknowledgments from all entities (thus verifying that the transaction can be performed by all of them), the sender can commit the transaction.

Consistency: The first handshake of a transaction sequence can be viewed as *probing* for the callee’s disposition to perform the transaction. If the callee acknowledges the initial request, the sender has ensured that the callee supports the transaction command in general,

and is currently able to invoke the corresponding command. Both sender and receiver have a consistent view of the receiver's current disposition and have a consistent view on the state of the transaction.

Isolation: Mbus Transactions allow a receiver to serialize concurrent transactions: during a transaction, a receiver can suspend new transactions with the same command, until the sender has committed the first transaction. Transactions that have been issued simultaneously can thus be processed in isolation. For isolated transactions, an application program should specify an application-dependent timeout for a transaction after which the transaction is canceled unless a `commit` message has been received. If the application semantics allow for a simultaneous processing of transaction requests, i.e., if the transaction has no side effects and is isolated due to its semantics, the application can also allow the parallel invocation of transaction requests with the same command name.

Durability: Although Mbus Transaction are defined to be not reversible (after having been committed) durability needs additional support from the application: for example, Mbus Transactions cannot provide any support with respect to durability in scenarios where systems crash after the Mbus interaction. Obviously, the application must provide appropriate means, e.g., employ persistent databases to guarantee the durability of transactions.

In general, Mbus Transactions are intended to increase the robustness and consistency of RPC invocations: receiving an intermediate acknowledgment after sending the first transaction message indicates to the caller that the remote entity is alive, that the requested procedure is defined, and that the entity is willing to invoke the procedure. The caller can then, in a second step, request the invocation of the procedure with a reduced probability that this invocation fails. If a transmission failure occurs and the second message cannot be delivered (or the caller does not receive an acknowledgment), it can either retry to send the second request or cancel the transaction, respectively do nothing and let the transaction time out. In each of the latter two cases, both caller and callee have a consistent view about the execution state of the transaction.

It should be noted that the second handshake (for committing the transaction) is sensitive to communication failures for the final acknowledgment: if the sender does not receive the final acknowledgment, it will assume that the transaction has not been performed. However, in these scenarios, robustness can be increased by sending both the commitment and the final acknowledgment as reliable Mbus messages and by applying the same enhancements as described for unicast RPCs, i.e., allowing a sender to re-issue a transaction command.

6.3.3 Mbus Control Models

In general, the Mbus is a communication channel for message passing within a group of modules, and Mbus implementations provide mechanisms to enable applications modules to pass messages to other Mbus entities. Each Mbus entity can provide a number of different services: It may perform certain operations for other entities that are triggered by the reception of Mbus commands or it may notify one or more entities of events. In the simplest case, an entity will simply receive Mbus messages and perform the operations that are requested by the commands contained in the messages. Sometimes, however, entities will only process certain commands if they are received from an entity that has registered as a client, e.g. a controller, before. Entities

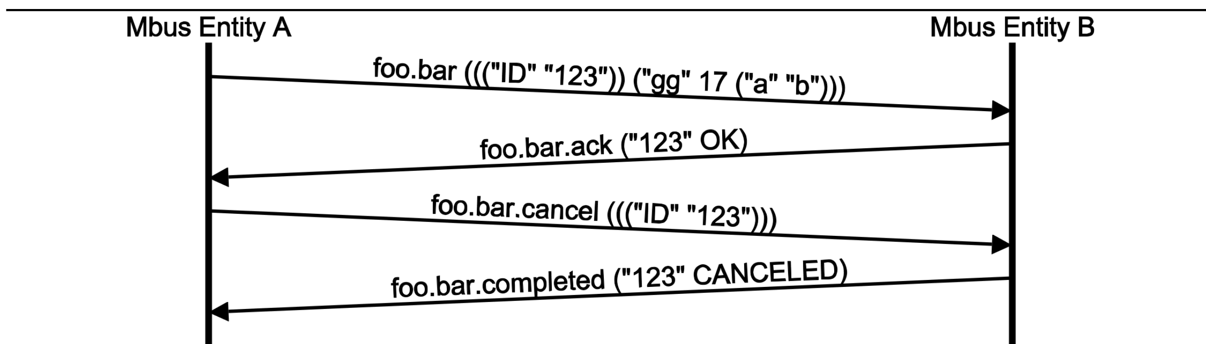

Figure 6.26: Canceled Mbus transaction

that are remote-controlled via their Mbus interface can restrict the number of controlling entities to one (at a time) in order to ensure consistency. Also, there are event notifications that are sent to a certain dedicated controller only, as well as there can be notifications that can be sent to a group of receivers, each of which having subscribed to this event source before. Again, in simple scenarios, entities may just broadcast all event notifications to the whole Mbus.

For example, a call signaling engine in an Mbus-based conferencing endpoint could provide several RPCs to control active calls, and it could generate different event indications, e.g., for forwarding call signaling indications from a specific call signaling protocol such as SIP. The deployment of such an Mbus module would only make sense in conjunction with a controlling entity that reacts to event notifications such as *incoming call*. In this example, the call signaling module could report these events to a default destination address as long as there is no registered controller, but in order to provide some useful functionality, e.g., answering calls or initiating new calls, a controlling module would be required. This controlling module would be interested in receiving the relevant event notifications and would thus register with the call signaling module. In this particular example, a controller would typically want to allocate the call signaling engine *exclusively*, i.e., the call signaling engine can only be controlled by exactly one controller at a time. The *exclusive control* of an entity prevents the entity from accepting control commands from another entity that has not registered as a controller before. For other applications, *multi-controller* scenarios are conceivable, where multiple controllers control another module in a distributed fashion. In this case, all controllers would register with the module, and the controlled module would multicast event notifications to the group of controllers.

We have formalized these different *control relationships* by the *Mbus service models* that are described in this section. Mbus profile definitions can specify the type of control model that applies to a set of commands. Typically, a service model applies to a command prefix that is used by several commands. In the following, the different control models (control relation classes) are described in detail and a list of conventions and recommendations for writing Mbus command definitions is presented. The following different classes of control relations are defined:

- *no control*;
- *tight control*; and
- *exclusive tight control*.

These different classes of control relations are usually applied to a command set that is implemented by some Mbus entities, and a control relation type is assigned to command sets in the command set definition. The motivation for defining different control models is to accommodate different applications with different requirements concerning flexibility and the level of control for their Mbus communication.

For managing control relations, we have defined a set of commands that allow a client to register (and de-register) with a service providing entity. Entities of the control class `tight control` and `exclusive tight control` require an explicit `registration` of a controlling entity before accepting commands from that entity. In the following sections, we first present the commands for managing control relations and subsequently, we present the different control classes.

6.3.3.1 Managing Control Relationships

Redirection commands belong to the class of RPC-commands. The following commands are defined:

`mbus.register:` This command is of type RPC and is sent by an interested client entity to a service providing entity for registering as a controller and for changing its default destination address for the given command (prefix). It provides two parameters:

- the name of the Mbus command (or command prefix) that the sender wants to register for; and
- the Mbus address that should be registered, i.e., used as the new destination address for event notifications. This allows an entity to redirect notifications to other entities than itself.

The application specific return parameters are `OK` (the calling entity has been added to the address list), `NO_SUCH_COMMAND` (the command prefix that has been specified in the request is unknown) and `DENIED` (the requesting entity is denied to register the given command prefix).

The return parameter is a list of Mbus addresses that represents the list of currently registered controllers.

`mbus.deregister:` This command is of type RPC and is sent by a registered client entity to a service providing entity in order to de-register from a command or service subscription. It provides two parameters:

- the name of the Mbus command (or command prefix) to de-register for; and
- the Mbus address that should be de-registered.

The return parameter is a list of Mbus addresses that represents the list of currently registered controllers.

`mbus.registered:` This command is an event notification and is sent by a service providing entity after a new controller has registered for a command (prefix). The first parameter specifies the name of the Mbus command, and the second parameter contains the new list of registered controllers for the given command. The notification is sent to the old list of clients (or to the default destination address if no other clients have registered before).

`mbus.get-registered:` This command is of type RPC and can be used in order to obtain the current list of registered clients for the specified command (prefix).

These four commands are used to manage controller relationships for all control models, i.e., for `no control`, `tight control` and `exclusive tight control`.

6.3.3.2 No Control

The control model `no control` is intended for entities that do not require the establishment of an explicit control relationship with another entity in order to accept commands from it. All Mbus commands, variables etc. of the respective command set can be used directly, and there is no regulation of the number of entities that may interact using the respective commands at a time.

A command set that is classified as `no control` may provide commands for unsolicited event notifications or even RPC-style commands that can be sent by an entity conforming to a specific Mbus command set definition. These Mbus commands that are originated by a conforming entity may be addressed to a default destination address. There may be a default destination address for all commands of a command set but each command may be associated with a specific default destination address. Commands of the `no control` class that may be sent without prior solicitation, such as event notifications, are usually assigned a default destination address.

The default destination address that an entity sends unsolicited commands to, may be changed by other entities. Entities may add themselves to a list of clients (controllers) that is maintained by another service providing entity. The effect of having the service providing entity add another entity to a list of clients is that the default destination address is no longer used but the respective messages are directed to the client entity. If more than one entity tries to add itself to the destination address list, it is up to the application to allow or deny this. Generally, entities of the `no control` class are expected to accept multiple clients. When multiple clients are present, each message that would otherwise just be sent to the default destination address is sent either to an Mbus group address that uniquely represents the registered clients or is sent independently to all clients. Clients may also de-register. When all clients have de-registered the entity uses the default destination address for the respective command again.

The changing of the default destination address is called *redirection*. Redirection may take place on single commands or a complete command set. If a command or a command set uses a default destination address that can be redirected by clients, it is marked as `REDIRECTABLE` and the default destination address is given.

Figure 6.27 depicts a message flow that involves an entity of the `no control class` (`module:call-control`). In the initial state, the entity uses its default destination address (in this case, we assume an Mbus broadcast address), and sends the event notification `call-control.incoming-call()` to all entities in the current Mbus session. A controller (`module:control`) redirects the destination address by sending the register command

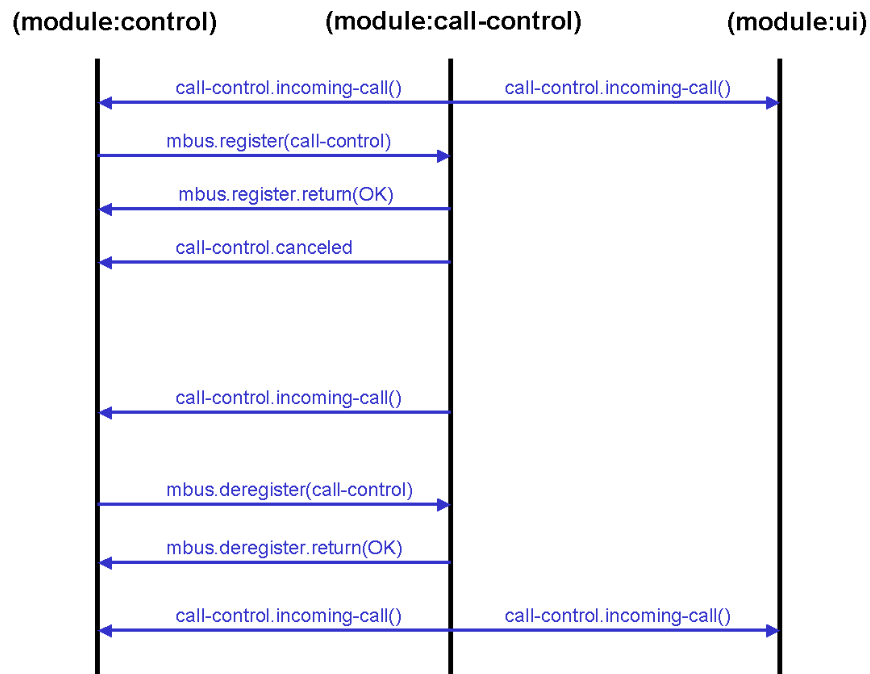

Figure 6.27: Usage of default destination addresses

for the Mbus command hierarchy `call-control`. As a result, subsequent event notifications such as `call-control.canceled` are sent explicitly to the `module:control` entity. In this case, the call control engine sends two notifications, `call-control.canceled` and `call-control.incoming-call()`. After the registered controller has de-registered, the default destination will again be used for all event notifications.

6.3.3.3 Tight Control

An entity that requires `tight control` for some or all of its Mbus controllable resources will only accept commands from an entity that has established a control relationship before. This means that Mbus commands, variables etc. can only be accessed by another entity after it has registered itself as a *controller* at the entity that provides the resources. Upon this registration, the controlled entity adds the new controller's Mbus address to a controller-address-list that is used for authorization and for sending event notifications etc. The complementary de-registration command enables entities to end the control relationship. Again, there is no regulation of the number of entities that may register themselves as a controller at a time.

Entities that conform to a command set definition marked as `tight control` do not send commands or event notifications to a default destination address for resources of that set.

Figure 6.28 depicts a message exchange for the registration of a controller `(module:control)` with a call control engine `(module:call-control)` that implements the control class *tight control*. In this case, the call control engine accepts multiple registered controllers and disseminates event notifications to all of them. In addition, the call-control

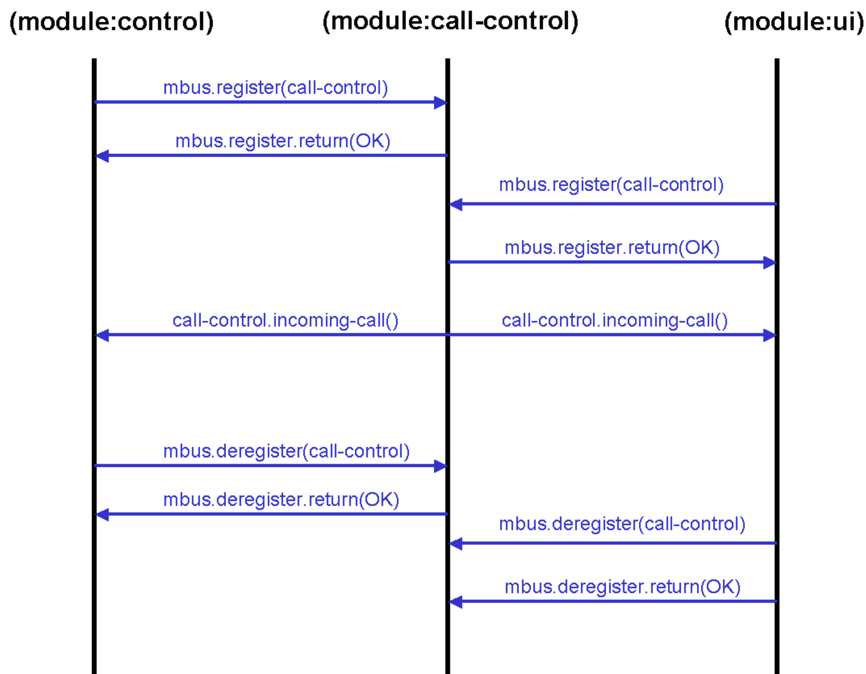

Figure 6.28: Registration of a controller with an entity implementing tight control

entity would accept control commands such as RPCs from all currently registered controllers.

In these *multi-controller* scenarios, there are typically application-specific requirements for guaranteeing consistency across the controllers and the controlled entity. For example, for the coordination of a distributed state that is manipulated by the exchange of RPCs sent by controllers to the controlled entity (i.e., any controller can send an RPC message), the controlled entity must update the state at the other controllers upon receiving an RPC from a specific controller entity. In Section 9.3.4, we discuss a possible solution for Mbus multi-controller systems for Mbus Call Control applications.

6.3.3.4 Exclusive Tight Control

The control model `exclusive tight control` has the same semantics as `tight control`, except for the number of controllers at a time: An entity that provides an Mbus command set that has been marked as requiring exclusive tight control will only accept one controller at a timer and reject register requests once a control relation with another entity has been established.

When a register request is received while another entity is currently registered as a controller the receiving entity returns the value `DENIED`.

Figure 6.29 depicts a message exchange for the registration of a controller (`module:control`) with a call control engine (`module:call-control`) that implements the control class *exclusive tight control*. After the registration message exchange, the call control engine does not accept further registration requests and also rejects RPCs and other

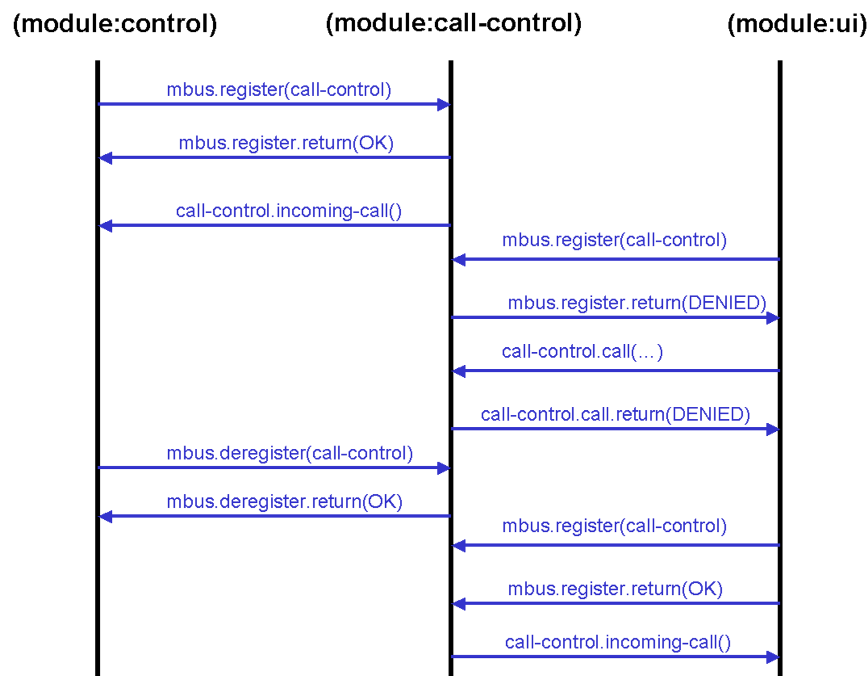

Figure 6.29: Registration of a controller with an entity implementing exclusive tight control

commands for the Mbus command hierarchy `call-control`.

6.4 Mbus Bootstrapping

The transport protocol mechanisms and implementation requirements described in Section 6.2 relied on the existence of a well-known Mbus configuration (consisting of Mbus transport and security parameters). This configuration has to be available for all application components that are to be integrated into an application session.

For many applications, it is appropriate to couple the Mbus session to a user's presence, i.e., to consider a set of Mbus entities as application components that provide a certain service for a user or on behalf of a user. This is reflected by the Mbus configuration concept that we have described in Section 6.2.5: In general, we assume that a user's Mbus components have access to a common file or registration database or can be configured in similar ways prior to establishing an Mbus session. This is feasible for many desktop computer and workstation operating systems. For applications that are to be distributed on multiple hosts that do not have access to a common file system or database, it is still conceivable that a user provides the required Mbus configuration herself.

Because the user would never distribute this configuration to other users, it does not have to be changed frequently, e.g., to exclude other entities at a later time. In summary, we can assume that a user relies on one single Mbus configuration that she distributes on all her personal devices and computing environments. The lifetime of this configuration can (and will often) be greater than the lifetime of a single application session. Of course, it is nevertheless possible for a

user to employ multiple Mbus configurations, e.g, in order to use different Mbus sessions for different applications.

Of course, the group communication features can be applied to more dynamic scenarios as well, where application components from different user/owner domains are integrated into a common session. A simple example is the integration of a new device into a user's Mbus session, where we assume that the device, say an RTP-enabled speaker-box, does not provide an appropriate user interface for configuring the required Mbus parameters in a convenient way. Another example is a user who is mobile and carries her Mbus environment with her, e.g., her personal communication devices. As soon as she enters a desktop computing environment, say in a non-territorial office environment, she wants to integrate applications on a desktop computer into her personal Mbus environment.

In both scenarios, we have to conduct a *bonding* process, where the new device and the existing entities agree on a common Mbus configuration. In order to be able to conduct this process, the involved entities have to locate each other (in the network) and associate with each other, which involves the exchange of Mbus configuration parameters. Because these parameters contain crucial keying material that is the basis for the integrity of a user's distributed Mbus application, the design of the association process has to address security requirements.

In order to allow for such dynamic associations of Mbus entities, we have defined the *Dynamic Device Association* framework (DDA, [Kutscher03b]) — a framework for the discovery and association of devices that is explicitly targeted at mobility of users and does not compromise security. We have designed the framework to be general enough to be applicable to other session protocols besides Mbus but have implemented it for Mbus first.

The Dynamic Device Association concept generalizes the ideas of service location and bootstrapping of communication sessions. The service location and service selection functionality is intended to be usable in different network environments: statically configured enterprise networks, ad-hoc-networks, and networks with both mobile user agents and mobile service agents. The device association functionality is not limited to specific application protocols and provides secure authentication to dynamically located services, including user (and service) authentication and confidential transport of application session parameters.

In Section 6.4.1, we present a more detailed sample scenario for dynamic device association, and in Section 6.4.2 we discuss the concepts of some existing approaches with respect to service discovery and the dynamic establishment of application sessions. In Section 6.4.3 we describe the general design of the DDA framework, and in Section 6.4.4 we present our specific implementation for the initiation of Mbus sessions. Section 6.4.5 summarizes the main results and findings of our DDA work.

6.4.1 Sample Scenario

In a sample mobile user scenario, Alice and her visitor Bob meet in a conference room of Alice's company. Alice brings her laptop, Bob his PDA, and both use the available WLAN. The conference room is equipped with a SIP-based conference phone and a regular SIP phone. Alice's laptop and Bob's PDA find two devices offering telephony services and obtain their locations and labels. Alice attaches to both devices, establishes an Mbus session and uses the conference phone to place a call to Carol with whom they are supposed to have a tele-chat. Alice and Bob use the speakerphone for the voice conversation and they use Alice's laptop to add a slide presentation to the call. During the teleconference the speakerphone is not accessible

for others. Bob is only allowed to access the regular SIP phone to which he also attaches. Both may receive incoming calls through the regular SIP phone, but only Alice may place outgoing ones. After the conference call completes, Alice and Bob dissociate from the two phones and leave. From this scenario, we can roughly identify the following five phases of dynamic device association:

1) Service and Device Discovery: Initially, the mobile devices need to find services offered by other devices. The discovery obviously involves an identification of the services, their availability (free vs. in use), their location information (to determine physical proximity), and a rendezvous address.

2) Device selection: Once a set of suitable devices has been found, the user may select a particular device. This selection process may also be fully automated (by user preference, physical proximity, or some other algorithm).

3) Service and Device Association: As soon as the user has picked a particular device, the mobile device invokes an association process. The selected service is contacted and an authentication procedure is carried out, after which the application protocol is bootstrapped: all necessary configuration parameters are exchanged and the application protocol is initialized.

4) Application Protocol Operation: The application protocol is run in the context of the dynamically established association. This may involve all kinds of interactions between the mobile and the associated device.

5) Service and Device Dissociation: When the associated device is no longer needed, the user's device dissociates from the device, freeing all allocated resources, and potentially making the device fully available again to the public.

Different protocols may be employed to implement the necessary mechanisms for each of the above phases. The following section discusses related work in this area, particularly regarding phases 1 and 3. Section 6.4.3 will then build on this foundation and discuss the concrete system architecture that we have developed and the protocols that we have chosen (and enhanced) for our design of dynamic device association.

6.4.2 Service Location and Device Association

In Section 4.3.2, we have described different solutions for service discovery and ad-hoc communication, including the *Service Location Protocol* (SLP) and *Universal Plug and Play* (UPnP).

When we compare SLP and UPnP, we can state that SLP (Section 4.3.2.1) provides mechanisms for service discovery and service selection in static enterprise networks and does not address the service association problem. It focuses on simplicity and scalability with respect to the number of user agents and can (optionally) rely on the presence of service directory agents as part of a network infrastructure. We believe that SLP's exclusive usage of service requests by user agents can lead to sub-optimal behavior in the case of ad-hoc communication and mobility, which we will discuss in more detail in Section 6.4.3.1.

UPnP (Section 4.3.2.3) is intended as a complete solution for service discovery (using SSDP) and device control (using SOAP). It is explicitly targeted at service discovery in "dynamic" environments such as home networks where devices (both user agents and service agents) can be connected dynamically. For this reason, it relies on periodic service announcements, however without addressing scalability properly. The device association and device control using SOAP does not address security, and UPnP as such is tied to SOAP as an application protocol.

These observations have led us to develop a new design for a dynamic device association framework that is usable in dynamic ad-hoc environments, addresses the necessary scalability and security issues and is not tied to any particular application protocol. This design is presented in the following section.

6.4.3 DDA System Design

In this section, we describe the general DDA solutions for the five phases that we have introduced in Section 6.4.1: Service and Device Discovery (Section 6.4.3.1), Device Selection Section 6.4.3.2, Service and Device Association (Section 6.4.3.3), Application Protocol Operation (Section 6.4.3.4) and Service and Device Dissociation (Section 6.4.3.5).

For the presentation of the design concepts in the following sections, we emphasize the *general* DDA model, whereas in Section 6.4.4, we describe a specific implementation for the dynamic establishment of Mbus sessions. Figure 6.30 depicts the DDA process schematically. The concepts we present in the following refer to the five phases of device association that we have discussed in Section 6.4.1.

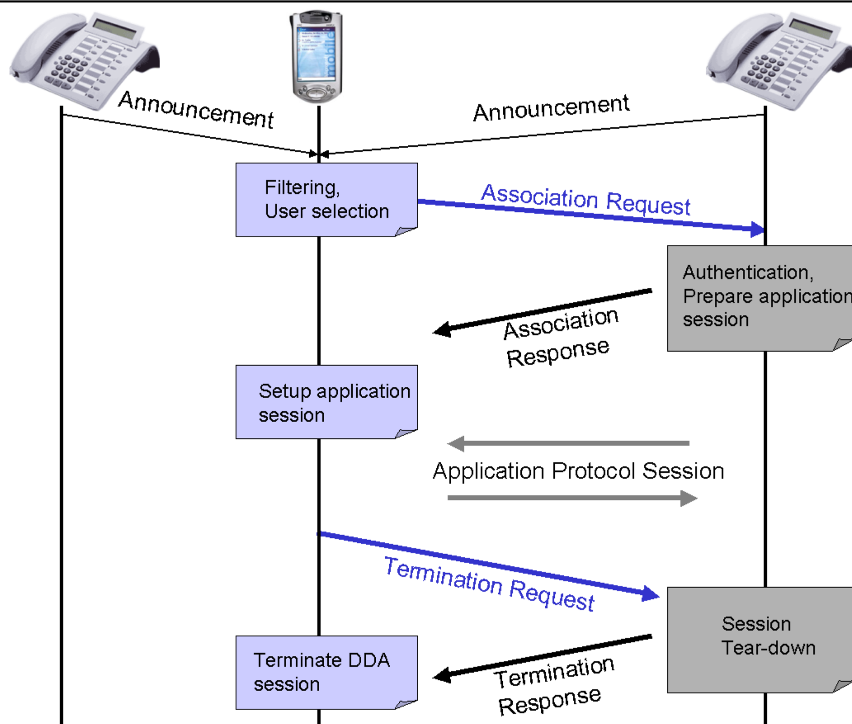

Figure 6.30: Overview of the DDA process

6.4.3.1 Service and Device Discovery

When discovering service providing devices, a user agent is interested in the following information: the existence and availability of devices (in range), the characteristics of the services they offer, and service association parameters.

In principle, the discovery of existing and available services is addressed by both SLP (Section 4.3.2.1) and SSDP (Section 4.3.2.3). However, to enable mobile user agents to discover (possibly dynamically changing) services offered by (potentially mobile) devices, the SLP mechanisms are not directly applicable:

SLP has been designed for enterprise service location where services are more or less permanently available and do not change their service attributes frequently. In this model, a user agent sends a service discovery request upon start-up (or when the respective service is needed) and uses it. The protocol is designed to minimize network traffic and to scale with respect to the number of services. As a result, a service itself is not advertised by announcements. Instead, user agents have to query actively (only directory agents announce their existence periodically). For the scenarios that we have described above, this model imposes some problems:

- For mobile devices that roam between networks, we cannot assume a static enterprise network with constant connectivity. A mobile device can enter and leave the *scope* of services quite frequently (and so may the devices providing services). For example, consider a user with a PDA that is carried to different rooms (potentially belonging to different organizational scopes).

- The user agent would have to initiate new queries every time it enters a new network. Relying on the query-response model would also inhibit the user agent from detecting timely that a service is no longer reachable and thus not available anymore. In cases of intermittent connectivity a user agent would thus have to send out queries periodically to validate its view of the available services in a network.

- A query-response model does not allow a mobile device to build up and maintain an ephemeral "directory" of available services: this would require either iterating through all conceivable service names (which appears infeasible) or using wildcard queries (which would need to be defined in SLP) at the risk of response implosions.

- When entering a new network, it can take quite some time until a host's IP interface is completely configured, depending on the configuration mechanism in place. For DHCP-based configuration that is frequently found in IPv4 networks, the DHCP service needs to be located and queried (in potentially multiple iterations). For IPv4 auto-configuration, it must be ascertained that no DHCP server is available and subsequently the uniqueness of the chosen address must be verified. IPv6 auto-configuration is a more efficient solution here but is not applicable to all scenarios. In either way, applications would have to deal with interrupted connectivity and failures when sending queries.

- Finally, relying on directory agents announcing their availability is not a suitable fallback: in some wireless ad-hoc networks, we can face the *hidden terminal* problem: this could lead to situations where we can communicate with the directory agent but not with the corresponding service agents.

As a solution, we have developed an announcement-based scheme, where services actively announce their availability and user agents receive and filter announcements depending on the service description. This approach allows user agents to discover services more easily, especially in environments with mobile user agents and ad-hoc-networking characteristics. SSDP also relies on an active announcement model but does not provide a rate-adaptation scheme and thus does not scale to large numbers of services. In our approach service agents announce themselves by multicasting advertisements, and service agents participate in a *rate adaptation process*: Each service agent observes the announcements of other service agents per time interval and adapts its own transmission rate so that the total announcement data rate roughly remains constant, regardless of the number of service agents. Service agents also announce discontinuation of their service. User agents receive service announcements but can optionally also send explicit service requests, and service agents include their current estimate of the group size in their announcements in order to inform user agents and other agents about their view of the group.

Using active announcements with a well-defined rate-adaptation scheme has several advantages: User agents do not have to poll to detect the availability of services, instead they receive announcements automatically when connecting to a network. User agents can also detect when services are no longer available by monitoring the group size and calculating the retransmission interval themselves, automatically expiring services that cease sending announcements after some time.

Instead of static service descriptions, we rely on a *soft-state* approach, where the latest announcement is conveying the current service description, which is more practical for accommodating dynamic parts in the service description as well, e.g., information on service availability. In order to provide user agents with enough meaningful information as a basis for the service selection, the service descriptions should include at least the following information in addition to the service type:

- the session protocol that is used to access the service;
- information on the current availability status of the service (e.g., for services that can only be used by a limited number of users at a time);
- information on the geographic position of the service (if relevant);
- service attributes that describe the capabilities of the services in a meaningful way that enables user agents to select an appropriate service; and
- a service URI that a user agent can traverse in order to associate with a service.

The DDA framework provides all of this information in the periodic service announcement by service agents and also in the service replies that are sent in response to explicit service queries. Note that this approach differs significantly from the SLP model, where two message types for describing services are distinguished: *Service Replies* that are sent as answers to *Service Requests* and *Attribute Replies* that are sent as answers to *Attribute Requests*.

With SLP, a user agent usually issues a service request first in order to locate suitable services (the service request may contain predicate strings to specify the required service capabilities) and then queries the attributes in a second step by sending an attribute request directly to selected service agents.

In the announcement-based model that we propose, this two-stage request-response model is not adequate; therefore we propose to include all the potentially interesting information in the announcement (and in the responses to explicit requests) and not to provide a second *attribute request*.

The service announcement must also include a *service URI*, i.e., a URI that identifies to a service association point where the user agents can authenticate itself and request the session parameters. The details of the device association step are discussed in Section 6.4.4.2.

6.4.3.2 Device Selection

The device selection can be realized in different ways, depending on the application, the network characteristics, and user settings:

- The user agent can be configured with a filter expression to consider only services of a specified type and with certain attributes. E.g., user agents will probably know which session protocol they support and thus neglect service announcements for services using unsupported session protocols. A preference-based ranking may be included as well.
- For some applications, physical proximity — if provided sufficiently fine-grained and reliable — may be used as a criteria for automated device selection.
- If the device selection process cannot be automated, the list of currently available services is presented to the user. For example, if a user employs a PDA to locate IP telephony services, the PDA software may offer the user to select a device based on its name, position, or other attributes.

  Mobile devices may continuously monitor their environment and present a (structured) service directory to the user from which she can select services to access. The choice of services presented may of course be subject to prior filtering.

6.4.3.3 Service and Device Association

After a user agent has selected an appropriate service, the actual *device association* takes place. The user contacts the service URI that is specified in the announcement, authenticates itself to the service and requests the session parameters for the actual service session. The following requirements can be identified for these steps:

Authentication: Some services such as telephony and printing services may only be made available to authorized users. Therefore, DDA must support authentication of user identities. The specific mechanisms depend on the available infrastructure and the application scenario. For corporate environments with a set of well-known users and an appropriate security infrastructure, public-key based mechanisms as well as shared secrets (i., passwords) may be used. Guests may be authenticated using one-time (or one-day) credentials, which may take either shape. With an inter-domain public-key infrastructure, guests may also be authenticated as individuals.

Our DDA scheme supports certificate-based authentication as well as password-based authentication. If no personalized authentication is possible, access passwords can be provided, e.g., at a help desk.

Confidentiality and Integrity: After the user has been authenticated the parameters for the actual session protocol have to be negotiated. Since this may include sensitive information such as transport parameters that should not be disclosed and keying material for securing the service session itself, this data exchange may need to be secured with respect to confidentiality and message integrity. Confidentiality is achieved by encrypting the communication; integrity is accomplished by relying on hashed message authentication codes (HMACs).

Description of session protocol parameters: One or more services can be described in a session description. The DDA protocol is not tied to any specific session protocol, it must be possible to describe session parameters for different protocols such as Mbus, SIP, HTTP, and SOAP.

Independent of the specific protocol in use, the following parameters are described: an explicit lease duration, a service URI for re-associating and a service URI for dissociating.

6.4.3.4 Application Protocol Operation

After the user agent has obtained the session description it can start to use the corresponding service, e.g., by joining the Mbus session or by sending SOAP requests. The device association has been established, and the communication session that has been used to exchange the parameters is terminated. The service agent and the user agent communicate over the application specific session protocol. The application protocols that DDA intends to support can differ significantly in their session semantics. We can classify the different types of application protocols as follows:

Protocols with aliveness control: Some application protocols, such as Mbus, support the constant monitoring of the aliveness of communication peers. E.g., some multicast-based protocols rely on periodic control messages that can be used to track the aliveness of peers, but the general principle is applicable to unicast protocols as well. These protocols achieve a comparatively tight coupling between the communicating entities and can thus usually provide indications to the application when a communication peer does not seem to be alive anymore. The application can use this information to conclude that the application session is terminated.

Protocols without aliveness control: For request-response-oriented application protocols, e.g., HTTP-based protocols, the whole application session consists of individual transactions, where the number of transactions is usually not limited. There is no upper time limit for the interval between requests, so it is not possible to reliably infer any aliveness-information from a sequence of transactions. For this class of protocols, it is extremely difficult to determine the current state of the communication peers and of the protocol session. Services could be blocked indefinitely, if no other mechanisms exist to terminate application sessions.

In order to accommodate both types of application session protocols we rely on a *lease concept*, i.e., on explicitly limited usage durations: a service agent specifies a maximum lease time and if a user agent wishes to use the service beyond this time, it has to *re-associate* with the service agent, i.e., it has to go through the association process again and obtain a new lease

(and potentially new parameters). If the user agent does not re-new the lease the service agent can assume the session to have terminated when the lease expires. The concept of leases is used by other protocols as well such as DHCP [RFC2131] and the Jini architecture (Section 4.3.2.2).

6.4.3.5 Service and Device Dissociation

For service dissociation, we also have to take the characteristics of application session protocols in account. An important aspect is the provision of *membership control functions*, such as the Mbus membership information service. We can distinguish the following different types of protocols:

Protocols with explicit membership control: Some application protocols support the explicit termination of sessions, e.g., Mbus provides the `mbus.bye()` command and RTP-based applications can make use of the RTCP-BYE packet. These protocols enable applications to explicitly terminate sessions. However, there are protocols that support explicit termination but are asymmetric by nature, i.e., only one side can terminate the sessions. This applies to all request-response-based protocols where the "server" has no way to signal the termination of a session independent of answering requests.

Protocols with implicit membership control: Protocols with aliveness control can provide implicit membership control: when the aliveness of a communication peer can no longer be ascertained it can be considered to have left the session and the session can be terminated.

Protocols without membership control: Protocols without any mechanisms for signaling the termination of a session can only rely on external mechanisms, e.g., on the explicit termination by traversing a service dissociation URI.

It is evident that we cannot rely on the application protocol for realizing a controlled termination of the application session in all cases. Even protocols that provide explicit membership control can fail to work, e.g., in the presence of communication errors or system failures. The lease concept helps to terminate sessions eventually. In addition to the implicit termination through the lease-expiration user agents can also explicitly terminate the session by traversing a service dissociation URI. Such a URI can optionally be specified in the service description that a user agent obtains from the service agent during the association phase.

6.4.4 DDA Implementation

We have filled the DDA framework described in the previous section with two main protocols: Phase 1 (service and device discovery) is realized by the Session Announcement Protocol (SAP, [RFC2974]), which we use for multicasting service announcements. The service description is expressed in SDP, with some DDA-specific extensions. A simple SLP-style request scheme using unicast SAP responses allows for service queries. For device association and dissociation (phases 3 and 5), we have used HTTP (optionally with TLS), with HTTP-digest-authentication as a minimal user authentication mechanism. The application session parameters are contained in HTTP bodies and are also described with SDP (`application/sdp`). Device and service selection (phase 2) has been implemented by developing a GUI-based service directory running

on a PDA. Mbus is used as the application protocol (phase 4). The remainder of this section focuses on the conceptually relevant phases 1 (Section 6.4.4.1) and 3 (Section 6.4.4.2).

6.4.4.1 Service and Device Discovery

The discussion in Section 6.4.3.1 has shown that in order to meet the requirements of efficient service location for different scenarios — including ad-hoc networks and mobility of user agents and service agents — an announcement-based protocol is preferable, with additional provision for rate-adaptation in order to maintain scalability. Entities that want to announce a communication session that can be joined by interested parties, must announce this session using SAP (that we have described in Section 2.2.1.1).

For dynamic device association, SAP is used according to the rules specified in RFC 2974 [RFC2974]. In particular, the standard SAP multicast group and port number are used. In addition, all rules concerning the calculation of announcement intervals are adopted. The scope for the announcement packets is application specific and should be determined with respect to the session protocol that should be used for the device communication. For example, if a link-local coordination protocol such as Mbus is to be used and the announcing entity wants to announce an Mbus-based service on its local link, the scope for the announcement packets should also be link-local.

In order to allow for shorter announcement intervals for the link-local DDA announcements, we have increased the bandwidth limit for DDA announcements over SAP to 64 kbit/s and require that the interval for sending DDA announcements should not be shorter than 5 seconds. Based on SAP's algorithm for determining the announcement interval [RFC2974], the base announcement interval can therefore be calculated as follows:

$$interval = max(5; \frac{8 * no_of_ads * ad_size}{64000})$$

where

no_of_ads is the total number of advertisements received; and

ad_size is the average message size in bytes.

It is possible that DDA servers change the description of the announcement from time to time, e.g., to publish up-to-date utilization information. When a DDA server updates the description for an announcement, it does not send a SAP deletion packet for the announced session. Instead, it increments the version field in the session description and calculates a new SAP session hash value. By relying on periodic re-transmissions of the announcements every DDA client that is in range of the server will eventually receive the current service description.

When a service that has been announced by a DDA server is no longer available, e.g., when the server is going down, the DDA server sends a SAP deletion packet for the SAP session. The following information must be expressed in an announcement for dynamic device association:

- The HTTP-URI that must be used by clients to initiate the parameter negotiation (see Section 6.4.4.2).

- A service identifier to describe the service type.
- The session protocol that is used for the communication session. This information is required because it allows clients to select an announced service by considering the used session protocols, i.e., if a client determines that a server uses an unsupported session protocol, the client does not have to enter the negotiation phase and can simply ignore this announcement.
- Meta-information about the communication session. These parameters are application-specific.

The server uses the same description language for the announcement as for the description of the session parameters.

An SDP description describes configuration setting for one more multiple application sessions and is structures as follows: the description consists of a *session level section* and one or more *media level sections*. The *session level section* provides general information about the session and configuration parameters that are common to all application sessions. Each `m=` line starts a *media level section* that provides configuration parameters for a single application session.

```
v=0
o=sip-phone-1 3235206722 3235206722 IN IP4
s=
i=DDA session
c=IN IP4 10.1.2.3
t=0 3235293122
a=app:dda mbus
a=dda-device-id:32778be73ef9823097d22957b3e5809a
a=dda-device-location:MZH%205160
a=ip-phone:SIP dku's%20phone Example.com%20Phone-X
a=dda-stats Local 42
m=dda-control:443 HTTPS -
a=dda-connect:https://10.1.2.3/connect
a=dda-device-type:ip-phone
a=dda-device-status:AVAILABLE
a=mbus:224.255.222.239 47002
```

Figure 6.31: Example for a DDA service announcement

Figure 6.31 depicts a complete example for a DDA announcement of an Mbus-based service. We have used some of SDP's elements such as `o` (originator) for specifying a device identifier as well as the version of the advertisement and `m` (media name and transport address) for specifying protocols and transport addresses. In addition, we use the SDP description to provide a number of DDA-specific parameters such as the application type, the name of the session protocol and

device identifiers. These additional parameters are represented as SDP *attribute* lines (a=) — SDP's extension mechanism that allows for the inclusion of arbitrary session description attributes. The order of attribute lines in a specific section is not significant.

For DDA session announcements, the following session attributes are used:

app:<application-type> <session-protocol>: The app attribute is used to express the type of announcement. DDA receivers ascertain that a received announcement is a DDA announcement by checking for the presence of this attribute. The session protocol is given as a second parameter to allow the selection of appropriate announcements.

dda-device-type:<device-type>: This attribute is used to specify the type of device that is offering the service. This information can be used by DDA clients to distinguish devices of different types.

In the example of Figure 6.31, the device type is specified as ip-phone. The type identifier is application-specific.

dda-device-id:<device-identifier>: This attribute is used to specify a unique identifier for the device that is offering the service. This information can be used be DDA clients to distinguish different devices.

dda-device-location:<location>: This attribute is used to specify the geographic location of the device that is offering the service. The type of the location specification is application-dependent (e.g., latitude, longitude or room number). This information can be used be DDA clients to locate a suitable device based on the announced location.

dda-stats: This attribute is used to communicate the number of service agents that are believed to be active in the given scope.

dda-device-status:<status>: This attribute is used to specify the availability status of the device and can take the values AVAILABLE or BUSY.

The example of Figure 6.31 features an attribute mbus that is used to describe the IP address and the port number that are to be used for establishing an Mbus communication session with the device. This attribute is protocol specific and only provided for informational purposes. The attribute informs other service providing devices of the communication parameters that the announcing device intends to use or is using (the attribute must be specified multiple times, e.g., when a device supports multiple parallel sessions) in order to facilitate an efficient usage of addresses and port numbers: Other service providing devices can monitor the usage of multicast addresses and can select unique addresses for their own advertised sessions, thus minimizing the possibility of address clashes. The mbus attribute only applies to the description of Mbus sessions. The first parameter is the IP address and the second parameter is the port number.

The media field indicates the media type dda-control, and the protocol field specifies the used application session protocol, e.g., HTTP, HTTPS, SIP, or MBUS. Figure 6.31 illustrates the use of SDP for HTTPS and Mbus. The service URI that a client should use to authenticate itself and request the session parameters is specified in the attribute dda-connect that occurs in the media section.

6.4.4.2 Service and Device Association

After a user agent has selected an appropriated service, it associates with the service by authenticating itself and by obtaining the application session parameters. In our implementation we have mapped this to an `HTTP-GET` request: The user agent requests the session configuration, possibly providing authentication credentials for the user. After the user has been authenticated, the service agent replies with a HTTP-response providing the configuration data in the message body. We have considered two mechanisms for user authentication (that can either be used alternatively or in combination):

- The user agent can either connect to the server using HTTP or HTTP/TLS. In the case of HTTP/TLS, the user agent and the service agent can authenticate themselves using certificates.

- Alternatively, e.g., if a public key infrastructure is not available and certificates cannot be validated, the service agent can authenticate the user using HTTP digest authentication. TLS can also be used without user agent and service agent authentication, just to establish an encrypted communication channel and to provide message integrity. In this case, HTTP authentication as described above is used for user agent authentication.

The DDA service agent provides the Mbus session description (as an SDP description) in the message body of the response message to the `GET` request. Figure 6.32 depicts such an Mbus session description. For DDA session descriptions, the parameters for the session are described in a media level section of the SDP description. The `m=` line must specify the type `control` and followed by a port number (if applicable) and a protocol identifier.

```
v=0
o=sip-phone-1 3235206722 3235206722 IN IP4
s=
i=DDA session
c=IN IP4 10.1.2.3
t=0 3235293122
a=dda-connect:https://10.1.2.3/connect/4367-abc4-9786
a=dda-disconnect:https://10.1.2.3/disconnect/4367-abc4-9786
a=dda-lease:600
m=control 47000 MBUS -
c=IN IP4 224.224.224.224
a=mbus:HASHKEY=(HMAC-MD5-96,T2l/U6/0RLxKF/0a)
a=mbus:ENCRYPTIONKEY=(NOENCR)
a=mbus:SCOPE=LINKLOCAL
```

Figure 6.32: Example of a DDA session description

The session level section includes the lease time (`a=dda-lease`), a URI for lease-renewal (`a=dda-connect`), and may also provide a URI to explicitly terminate the application session (`a=dda-disconnect`). All other parameters are given as protocol-specific session attributes.

The example of Figure 6.32 uses the attribute `mbus` that is used multiple times to specify different Mbus communication parameters. Other session protocols will have other requirements for specifying session parameters. We have defined DDA fields for the description of the required Mbus parameters as described in Section 6.2.5.

Figure 6.33 shows the complete operation of a DDA process: the user selects an IP phone, connects to it using TLS, thereby authenticating the device and establishing a secure communication link (which is continuously kept open). HTTP digest authentication is used to verify the mobile user's identity and the application session parameters are conveyed in the HTTP body as is further DDA information. The application protocol operates and, when the lease expiration time nears, the DDA process is re-invoked to refresh the lease. Eventually, the application session and the device association are terminated.

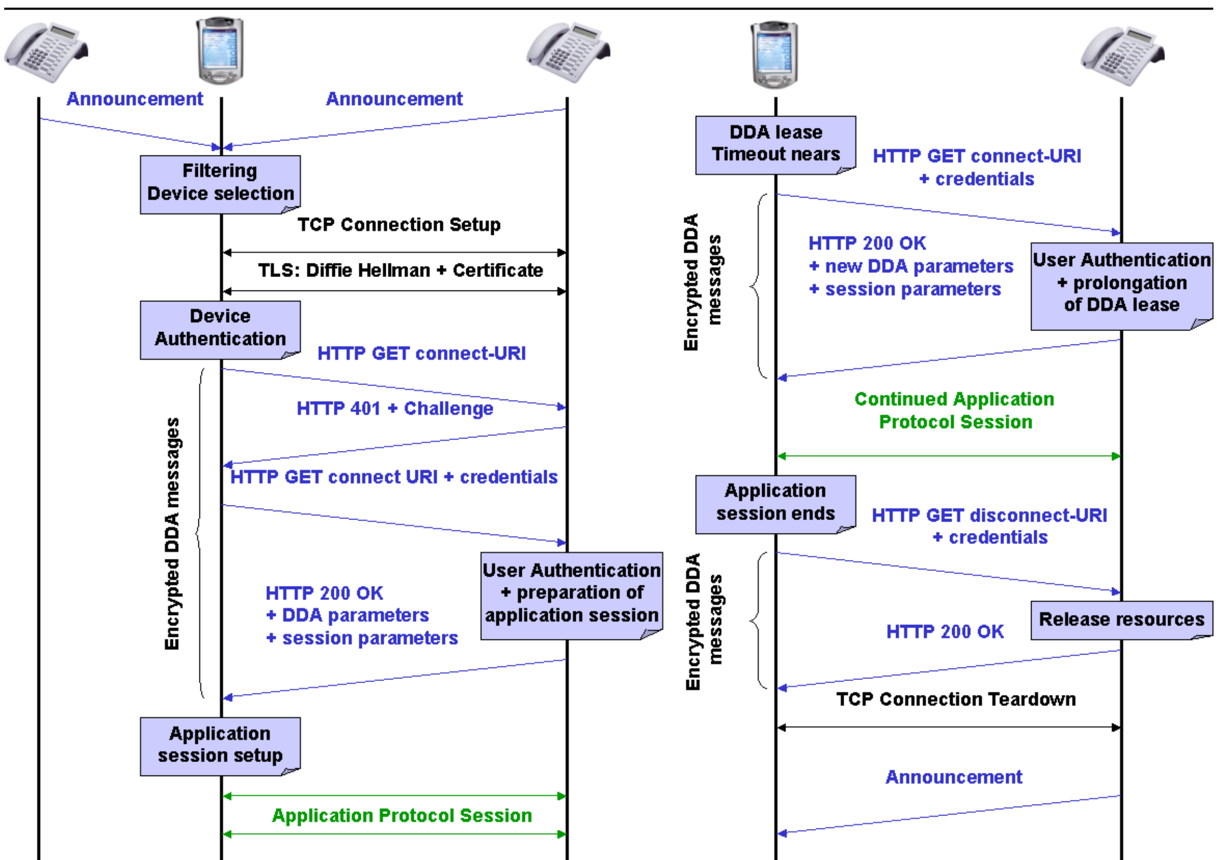

Figure 6.33: The complete DDA process

6.4.5 Summary

The Dynamic Device Association concept generalizes the ideas of service location and bootstrapping of communication sessions and provides a mechanisms for dynamically associating

Mbus entities to existing application sessions, without the need for prior configuration. We have presented some sample scenarios for which DDA can be useful, e.g., the dynamic association and control of *public* IP-telephony systems.

DDA builds on some concepts that have been implemented in other protocols before — in this section we have especially considered the SLP and UPnP. When analyzing these protocols and architectures some shortcomings could be observed. These observations have led to a different approach, with a scalable service announcement mechanism and a generic, secure service association procedure. We have implemented these concepts in a protocol specification and have applied the DDA protocol to the dynamic association of IP phones using PDAs with *DDA browsers*. The PDA applications can locate available IP phones, personalize and control these devices. The concrete implementation (relying on SAP and HTTP) is certainly not the only way to implement the DDA concepts. For service discovery, it is also conceivable to adapt the mechanisms of SLP and SSDP to provide similar functions. The combination of SAP and HTTP with SDP turned out to be a good compromise yielding a scalable, secure, and extensible solution that meets the requirements of our target application scenarios.

In Section 10.2, we present two applications of the DDA framework, which is used for the dynamic establishment of Mbus sessions between a user's personal device and communication services in her desk area environment.

6.5 Mbus and Ad-hoc Communication

The DDA approach that we have described in Section 6.4 provides a solution for the *dynamic establishment of Mbus sessions*, e.g., in scenarios where it is not possible or not practical to rely on shared configurations that are available to all session members up-front. The main application for DDA is thus the *bootstrapping* of Mbus sessions, and the typical usage scenario consists of an Mbus-based service that is being advertised, discovered by a user device and then utilized by a user who associates with the services and joins an Mbus session that is *owned* by the service. One sample application is to provide services to mobile users who dynamically discover and associate with services.

The DDA approach relies on a stable network connection between user device and Mbus service once the session has been established, and requires native IP multicast support for all participating entities. In this section, we extend the Mbus transport layer and the DDA framework in order to support *multiparty peering for ad-hoc communication*, i.e., the association with multiple Mbus entities in *ad-hoc networking environments*. We consider scenarios, where both user devices and Mbus services may utilize multiple, possibly heterogeneous network links and where the reachability and connectivity of Mbus entities is not constant but may vary, e.g., due to mobility. One application in this context is a user device that provides multiple network interfaces, e.g., a WLAN interface and a Bluetooth interface (using IP/PPP/RFCOMM or IP/BNEP). The user device wants to associate with multiple services that it discovers on both interfaces and wants to establish a common Mbus session between all desired services taking into account that devices themselves may move from one network link to another.

The *multiparty peering* extensions that are described in this section therefore have to provide the following functions that are required for the establishment and operation of Mbus sessions in *ad-hoc communication* scenarios:

- DDA advancements for the initiation of multiparty sessions; and

- Mbus and DDA mechanisms for multi-link sessions with heterogeneous capabilities.

One key element that we describe in this section is a multi-link concept for Mbus and a Mbus message relaying function that allows to bridge multiple of such links independent of their individual characteristics, e.g., with respect to multicast transport capabilities. In Section 6.5.1, we present more detailed scenarios for group communication in ad-hoc environments and derive some requirements. Section 6.5.2 introduces extensions of the DDA approach for accommodating multiparty peering scenarios, and Section 6.5.3 enhances the Mbus protocol to provide group communication services in dynamically changing network topologies.

6.5.1 Requirements for Group Communication in Ad-hoc Environments

Many existing coordination protocols such as UPnP's SOAP rely on point-to-point communication between components and apply classical *remote procedure call* style interactions. Group communication is only used as a rendezvous mechanism for service discovery, e.g., multicast service discovery in UPnP (see Section 4.3.2.3). This is motivated by the fact that unicast communication is usually more manageable in today's production networks. Multicast is often restricted to isolated parts of the network, e.g, for security or bandwidth limitation reasons. Furthermore, many protocols are designed for static scenarios that do not change with respect to availability and reachability of services after the discovery process has been finished. This makes these protocols difficult to deploy in ad-hoc communication scenarios, where services can appear and disappear dynamically and where mobility of both service providers and clients might be involved.

However, we argue that group communication is an important element for service coordination because it is a natural solution for many coordination scenarios and may help simplifying several kinds of interactions. For example, event notification and the dissemination of soft-state information — two elements that can be useful for service coordination — can often be implemented efficiently by employing group communication mechanisms. In addition, group communication as implemented by the Mbus transport protocol can be a useful tool to enable communication between referentially uncoupled entities: sender do not have to know the exact addresses of potentially interested receivers but can send messages to a group address that serves as a *subject-based address*. This is especially interesting for the development of systems that are intended to be used in ad-hoc environments where network addresses do not necessarily have to be persistent for the duration of a session. Group communication in conjunction with interaction types such as *soft-state communication* can help to make these systems more robust against intermittent connectivity and changing availability of services.

The component-based conferencing system scenario that we have described in Section 3.1.1 provides a set of different application entities that dynamically become aware of each other through Mbus mechanisms and that can be combined for specific a call — according to a user's needs. This scenario that is based on a set of dedicated, statically pre-configured components of a conferencing system, can be extended to include mobile (nomadic) users who do *not* have an associated desk area environment — at a certain time or permanently. Examples include employees visiting co-workers in other offices or subsidiaries who want to stay connected as well as environments with non-territorial offices. Such users need to dynamically locate the devices or services they want to use in an ad-hoc fashion: in the simplest case they may just require access a single device in a foreign environment, e.g. to place a phone call controlled

from a portable device, so that only point-to-point communication needs to be established dynamically. If more application components need to be involved as for multimedia conferencing, group communication sessions need to be set up among a number of components in an ad-hoc fashion. These different devices may reside on different network links, e.g., on a WLAN link and on an Ethernet link. In many corporate production networks, direct multicast connectivity (and, due to layer three security mechanisms, even unicast connectivity) is not always available between these different links.[4]

In addition to the DDA bootstrapping mechanisms, such scenarios require model to allow for the integration of *multiple* service entities into existing user-initiated Mbus sessions and the integration of multiple entities that are not directly connected, i.e., that do not reside on a common network link.

6.5.2 DDA Extensions for Multiparty Peering

Similar to other approaches, the DDA model (Section 6.4) relies on the concept of a one-to-one relationship of service providers and service clients. A service client contacts exactly one DDA service and requests the necessary configuration data to establish a point-to-point communication session. Note, that when using DDA to initiate group communication sessions such as Mbus sessions, the DDA process does not preclude the possibility that the session the DDA client is joining, provides other session members in addition to the service providing entity. However, this is not a result of the DDA session set-up itself and thus orthogonal to the actual association process.

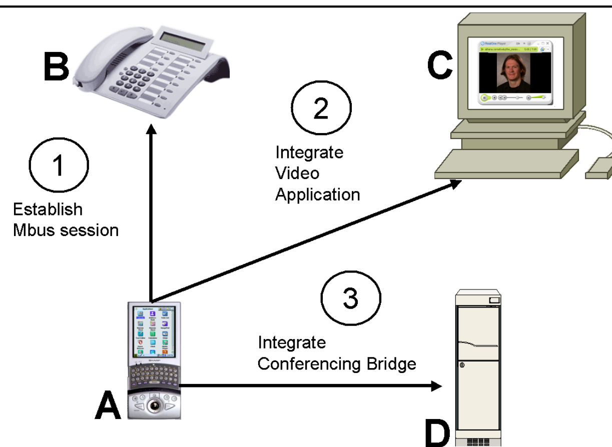

Figure 6.34: Sample multiparty peering process

For multiparty peering, there can be more than a single association step for a given session: multiple devices have to be provided with the session configuration and brought into a common application session. Applying a series of DDA association steps would not solve the problem,

[4]For example, large-scale WLAN installation often provide a design where the WLAN segment is considered as an access network only, without general Internet connectivity, and layer three security mechanisms such as VPN tunneling are in place to enforce user authentication and privacy.

as the DDA client (representing the user who wants to control a set of services in a common session) would potentially obtain different session configurations from each of the DDA services thus resulting in several independent, unrelated application sessions.

Figure 6.34 depicts a sample multiparty peering process, where a user device `A` (the DDA) establishes a session with an DDA-enabled IP phone `B` and later integrates additional services, such as a video application entity `C` on her workstation and a personal conference bridge `D`. While the first association can take place using the regular DDA mechanisms, the associations in step two and three must follow a different model, because the user device has to *invite* the video application and the conferencing bridge into the existing Mbus session.

The main requirement for a generalized DDA concept is the usability for both client-server and multiparty associations. In the following, we describe a few selected association scenarios first and then derive the most important requirements before describing the modified DDA association in Section 6.5.2.2.

6.5.2.1 Requirements for Multiparty Peering

We assume a scenario of 4 DDA enabled systems `A,B,C` and `D` that we want to bring into a common session. Let `A` be the user device and `B,C` and `D` service providing entities.

`A,B,C` and `D` reside on the same (multicast-enabled) link and `B,C` and `D` send out periodic service announcements as described in Section 6.4. We assume that DDA services do not associate unless a user device has triggered an association, i.e., we maintain the concept of a user-initiated session that performs a certain service for an identifiable user.

Obviously, the first requirement for a successful multiparty peering is that all DDA services and the user device support the same (group-communication) application protocol. For the rest of the discussion in this section, we assume that a group communication protocol such as Mbus is being used.

`A` will send an association request to the first DDA service (say `B`) and obtain a session configuration. Now, in order to bring `C` and `D` into the same session, there must be a way for `A` to associate with `C` and `D` and make them use the configuration of the existing session that has been initiated by the DDA process between `A` and `B`.

After the entities have been brought into a common session, the application-specific coordination can commence. For example, the controlling entity of the phone can determine the set of available application entities, query their capabilities and provide them with an initial configuration. When a conference has been initiated, e.g., through the user's extended phone GUI on the PDA, the controller coordinates the phone's call control engine, the application entities and, in case a multiparty conference should be established, the conference bridge (as depicted in Figure 6.34). Hence, the Mbus group communication session has been established through the DDA multiparty peering mechanisms, i.e., multiple entities, that have been in a coordination context before are associated dynamically.

Another requirement is that for any of the associations, both partners will need to know the application protocol specific address of the other party. E.g., when `A` associates with `B` it cannot rely on `B` being the only other entity in the group communication session, and `B` cannot assume that `A` has not already brought other peers into the session. There must be a way for both peers in an association process to indicate their application protocol specific and (possibly) transport protocol specific addresses to enable both peers to identify each other in the session and to verify that the association has been successful.

Taking this a step further, the associating entities must not only be able to learn each other's application protocol addresses during the association step, it must also be ensured that each of the addresses that is chosen by the communicating parties in each of the individual associations is unique in order to avoid address clashes.

In a different scenario, `A` may be a host providing multiple network interfaces that are used for listening to DDA announcements. The DDA association itself will not change, i.e., `A` will still receive DDA service announcements on its different interfaces and will invite the selected service entities into the Mbus session. Note, that although all entities might reside in different separated network links, it is nevertheless required to provide them with the same Mbus configuration, i.e., have them use the same multicast group in order to support mobility and changes in the network topology. For example, when a service entity that has been connected via WLAN becomes available on the Bluetooth link of a user device, the Mbus entity awareness mechanisms can allow for a seamless continuation of the Mbus session, if all entities use the same Mbus multicast configuration. Section 6.5.3addresses the Mbus specific implications for these ad-hoc changes in connectivity and network topology.

In scenarios where a user devices creates Mbus sessions dynamically, it has to determine usable configurations for theses sessions. One important aspect is the selection of a properly scoped and currently unallocated IP multicast address in order to avoid unwanted multicast traffic leaking and multicast address clashes. Hence, allocation infrastructure as described by the *Internet Multicast Address Allocation Architecture* (MALLOC, [RFC2908]) or distributed allocation and defense protocols such as the *Zeroconf Multicast Address Allocation Protocol* (ZMAAP, [Catrina02]) should be used for the dynamic allocation of multicast addresses. When inviting multiple service entities over different links, a user device has to verify the availability of the multicast address on all links in advance before it starts to invite the first service entity.

6.5.2.2 Modified DDA Association

The first extension to the DDA process is to generalize the association and to enable DDA clients to not only request a configuration from a DDA service but to optionally *invite* a service into a session by providing it with the required session parameters. Where applicable, service entities should provide both forms of association, i.e., be able to offer a session configuration and to be invited into a session. Service entities that are restricted to either mode should indicate their preferred association mode in their service announcement to avoid unnecessary requests/response cycles.

For the HTTP-based DDA protocol, we have implemented the *invitation* mode with an `HTTP-POST` [RFC2616] request. Note that the authentication requirements do not change for association invitations, e.g., for digest-based authentication, the DDA client would still provide the credential in the request message.

For DDA for Mbus sessions, we have defined additional attributes for the session description that allow both parties to express their Mbus and their corresponding UDP/IP endpoint address (IP address and port number). Because both parties have to know each other's addresses in advance, we allow for both the request and the response in every DDA HTTP request (`GET` and `POST`) to contain a message body. For example, when a DDA service is invited and has received a corresponding association invitation, it will send an SDP fragment in a response to that request that provides the required address attribute.

Figure 6.35 depicts the message exchange for the DDA invitation mode. After the DDA

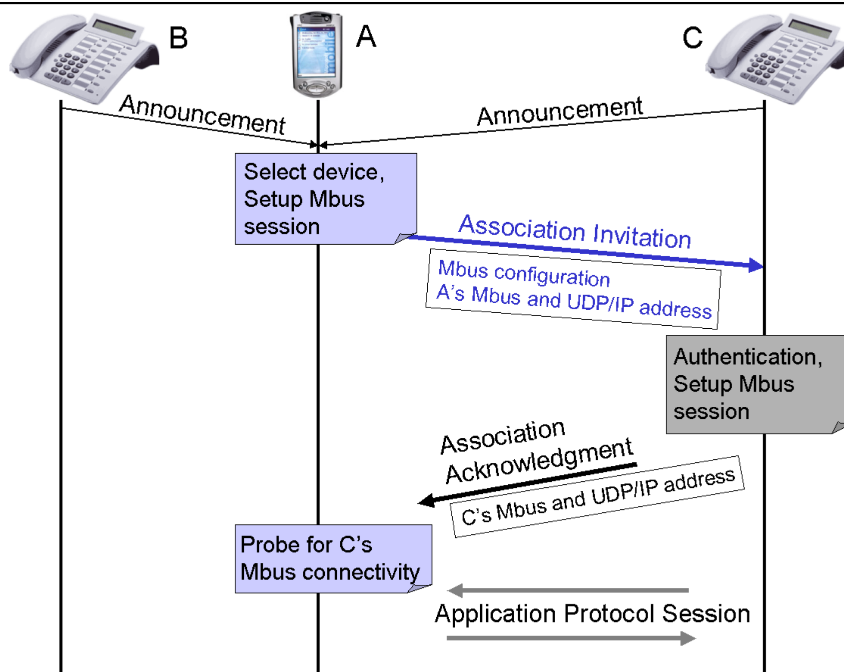

Figure 6.35: DDA in invitation mode

association has been completed, both peers can join the Mbus session and check for the availability of the other side, using the Mbus address information that has been exchanged in the DDA process. For scenarios where IP multicast connectivity is not available we have defined a *probing* process that helps to determine the optimal communication mode (e.g., multicast or direct unicast) between the initiator (the user device) and the invited service entity. The probing process is described in section Section 6.5.3.2.

It should be noted that the DDA announcement process relies on the availability of IP multicast. For special scenarios with limited IP host implementations or other restrictions it is conceivable to distribute the announcement using other means. E.g., for the communication between a PDA and a user's mobile phone dedicated short-range communication technologies such as Irda and Bluetooth could be applied. Although multicast connectivity is not available, the application on the PDA could obtain the service description out-of-band, e.g., via Bluetooth service descriptions. As long as unicast connectivity can be established, the session can be initiated, and the probing procedure described in Section 6.5.3.2 would yield that both entities can communicate via unicast. In Section 6.5.3, we also describe how the invited service entity can be brought into an existing Mbus session with other service entities by relying on the user device acting as a message relay.

For ad-hoc communication, the network topology and the connectivity for individual entities is likely to change during a session. For example, a user might roam from one network to another, and a device that has been visible over a short-range radio link is now visible over the corporate WLAN infrastructure. Mechanisms such as Mobile-IP can, in general, provide durable IP unicast reachability, but require corresponding infrastructure (e.g., home agents) and are not always sufficient to maintain the local multicast connectivity. For example, the new network may be in another multicast zone that is out of the local scope of one of the multicast

sessions that pertain to the Mbus session. It is also possible that the device has moved into a non-multicast capable network. Some of the topology and visibility changes can be dealt with on the basis of adapting the Mbus message forwarding as described in Section 6.5.3.4. In those scenarios where the Mbus connectivity is lost completely, we perform a new DDA association step (if the devices still is or has become available over DDA) and perform the regular exchange of addresses and probing procedures.

6.5.3 Mbus Extensions for Multiparty Peering

For the initial Mbus scenario in Section 6.5.2, we have implicitly assumed full multicast connectivity between all application entities — a safe assumption for an environment built up around a single link of a local area network with today's operating systems. However, this assumption is unlikely to hold as soon as mobile devices (such as PDAs or laptop computers) become involved which are likely to make use of WLAN infrastructure or even engage into ad-hoc communications with other devices using dedicated Bluetooth or infrared links, in addition to their connection to the (W)LAN.

The security risks of WLANs may force institutions to deploy dedicated routers and firewalls to protect their corporate network from external access, thereby (accidentally) preventing the propagation of link-local multicast traffic and possibly blocking multicast traffic explicitly. Also, because of the limited bandwidth compared to wired LANs, WLAN access points may have multicast forwarding disabled altogether. I.e., even if link-local multicast communication was possible (because the WLAN access point operated as a bridge to an multicast-enabled Ethernet link), it can be administratively disallowed in order to save bandwidth for unicast communication.

As Mbus-based applications rely on Mbus session-wide multicast connectivity, Mbus extensions are necessary that preserve this communication property in spite of the potential issue listed above. We have chosen to keep Mbus simple and straightforward and have devised minimal enhancements to support non-uniformly connected Mbus entities based upon our target application scenario (rather than designing a sophisticated generic application-layer message routing and forwarding overlay). This section presents the Mbus enhancements designed and implemented to enable robust ad-hoc multiparty peering using Mbus.

Key to the Mbus enhancements is the concept of a *coordinator entity* (Section 6.5.3.1), represented by the same device that has also initiated the DDA process and brought together all the Mbus entities. The coordinator probes the connectivity to all the associated Mbus entities (Section 6.5.3.2). All entities report the peers visible to them (Section 6.5.3.3) and, based upon this information the coordinator determines when to forward messages between links (Section 6.5.3.4). The proposed enhancements also deal with topology changes and failures (Section 6.5.3.5) and require minimal changes to our existing Mbus implementation.

6.5.3.1 Coordinator Concept

As discussed in Section 6.5.2, the establishment of an Mbus session is initiated by a user device that contacts the required service entities after a phase of service discovery. In our previous example, the user has manually selected the service entities that provided the required services for constructing a distributed conferencing session. Obviously, this very entity — subsequently referred to as the *coordinator* — has a complete overview of which other components it has

sought (and found) to realize the intended communication scenario. Furthermore, as the coordinator may have used entirely different link layer technologies to contact the various peers, it is the only one to take up the responsibility of initially establishing reachability between all involved parties and also to act as a hub in case native multicast connectivity cannot be established. Finally, the coordinator is the one entity capable of re-invoking the DDA procedures, e.g. to update Mbus configuration parameters or to prolong service leases.

Usually centralized architectures are deemed risky as they introduce at least a potential bottleneck and a single point of failure. With the DDA setting in mind, however, there may be no other way to establish connectivity between the entities in the first place. And, as the Mbus is used to communicate control messages only (rather than large data volumes), processing power and communication bandwidth are not considered to be problematic. In principle, the coordinator could obviously be a single point of failure. However, for sessions that are initiated by the coordinator system, the operation of the whole distributed system depends on the availability of the user device and the coordinator function anyway — thus we achieve fate-sharing with the intended application as the coordinator is also in control (at least initially) of the other devices. Future work in this area could address this robustness issue by investigating possibilities to re-organize the network by determining a new coordinator. However this would impose new requirements on every Mbus implementation involved and is therefore considered to be too costly for the operation of user-initiated Mbus sessions.

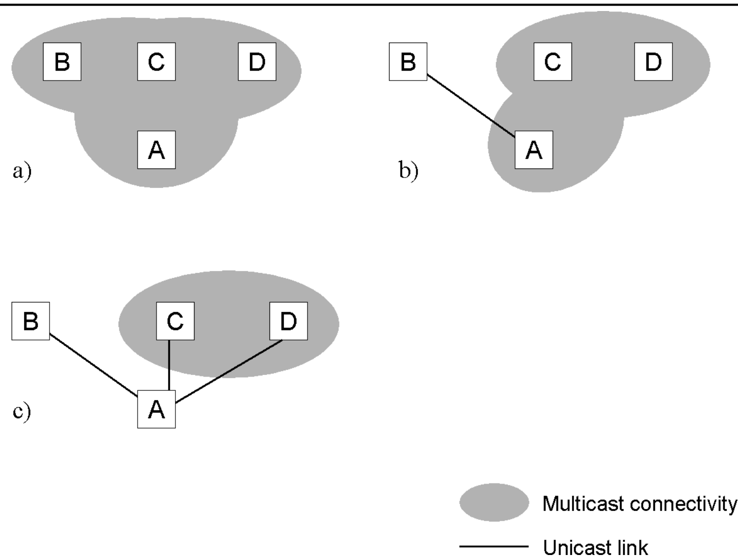

Figure 6.36: Three multiparty peering scenarios

Figure 6.36 shows three conceivable settings with a coordinator *A* and three devices *B*, *C*, and *D*. Note that, in the following, the term *multicast* refers to layer three multicast, i.e., IP multicast and *not* to Mbus group communication.

- In setting a), full multicast connectivity is available so that there is no need for the coordinator to perform any kind of message forwarding.
- Setting b) shows device *B* being on a separate link: *A* and *B* can communicate via unicast

and multicast, and so can *A*, *C*, and *D*. But *B* has no way to talk to *C* and *D* and vice versa, neither with unicast nor with multicast.

- Setting c) depicts a scenario with *B* on a separate link again, with *C* and *D* sharing the same link while *A* is connected via a router (or some other entity blocking link-local multicast). As a result, *A* and *B* can talk via unicast and multicast and so can *C* and *D*. *A* and *C* as well as *A* and *D* can only communicate via unicast.

Those three settings can be taken representatively for a majority of connectivity variants that one may experience in local ad-hoc communications. Note also that even though we do not explicitly discuss asymmetric connectivity, we have verified that the algorithms presented below will work in those cases, too.

6.5.3.2 Connectivity Discovery

As outlined above, the (mobile) coordinator needs to determine what kind of connectivity is available to its peers. As it was able to initially contact them and create an association, IP connectivity is available. The next step is to determine whether the respective entities can be reached via multicast or via unicast only.

For this purpose, we introduce `mbus.probe(m|u seq-no)` messages that are parameterized with a flag indicating whether this message has been sent, at the IP layer, via unicast (`u`) or multicast (`m`) and with a sequence number (for matching probes and their responses). The coordinator starts sending `mbus.probe` messages to each of the newly associated entities using their Mbus unicast addresses (learned from the DDA association messages, see Section 6.5.2.2). These messages are sent once via IP unicast (using the `u` flag) and once via IP multicast (using the `m` flag) with only minimal delay. For each message sent, regardless whether unicast or multicast, the sequence number (which starts at a random number) is incremented by one. The coordinator may retransmit these messages up to three times to deal with possible packet loss.

A receiver of such a message responds to each of the messages received after a short delay, once by unicast and once by multicast — so that six messages are exchanged in total. Each response message `mbus.probe.ack (m|u seq-no*)` again contains a flag indicating whether the message was sent via unicast or multicast and contains a list of `mbus.probe` message sequence numbers received from this sender (the coordinator) for the last two seconds. Figure 6.37 depicts a corresponding probing process between a coordinator and a single entity. Unicast messages provide the `u` parameter, and multicast messages provide the `m` parameter.

If the coordinator receives responses to `mbus.probe` messages via unicast and via multicast acknowledging both unicast and multicast probes, full unicast and multicast connectivity is available. Otherwise, the combination of response messages received (via unicast and/or multicast) and their acknowledged sequence numbers reveal in which direction multicast connectivity is available if at all. The result of this process serves as input to the configuration of message routing for both the coordinator and the Mbus entity. Based on the result of the probing process, both entities can determine, how to communicate with each other on the current link. For simplicity, multicast communication is only enabled if full multicast connectivity has been determined. In all other cases, the Mbus entities will resort to unicast communication.

Connectivity probing may be repeated in case network topology changes are suspected, e.g., when an entity has become invisible on a link.

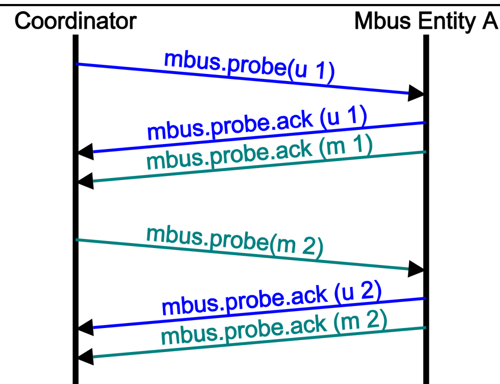

Figure 6.37: Mbus-based connectivity discovery

6.5.3.3 Visibility Reporting

Mbus entities announce their presence in regular intervals by means of `mbus.hello()` messages; thereby, for a multicast-enabled link, all members of a session become aware of each other. In a setting with potentially disjoint communication links, the coordinator needs to determine which Mbus entities can talk directly to each other and which require its help to forward messages.

To establish this view, each Mbus entity regularly transmits *visibility reports*, i.e., Mbus messages containing a list of other Mbus entities it is aware of. We distinguish two kinds of visibility: *native visibility* refers to Mbus entities whose `mbus.hello()` messages were received directly, i.e., without the help of the coordinator forwarding them; *effective visibility* refers to all Mbus entities from which `mbus.hello()` messages have been received recently. An Mbus visibility report is defined as `mbus.visible ((<Mbus address> [native])*)`, i.e., it contains a list of peers, each indicating the Mbus address being reported and a *native* flag showing whether or not the message has been received directly from the respective entity. If an Mbus entity receives both native and relayed messages from another, the native reporting takes precedence.

Initially, when all Mbus entities start communicating only native visibility reports are possible. If the coordinator observes that all entities can see all others natively no further actions are necessary on its part. Otherwise, the coordinator can determine from the visibility reports how the Mbus session is partitioned and start forwarding messages between those partitions. All Mbus messages except for `mbus.hello()` are forwarded unchanged. `mbus.hello()` needs to receive special processing to allow distingushing native messages from relayed ones: a single parameter pair `(via <Mbus coordinator-address>)` is inserted into the message so that one yields `mbus.hello ((via <coordinator-address>))`.

Figure 6.38 depicts the exchange of visibility reports between a coordinator (entity `A`) and two entities `B` and `C` that reside on separate *Mbus links*. After an initial exchange of `mbus.hello()` messages all entities report the current visibility status on each Mbus link they are attached to. `A` can see both `B` and `C` and sends a corresponding visibility report on each Mbus link. `A` receives the visibility reports from `B` and `C` and infers that `B` cannot see `C` and vice versa. Based on this observation `A` start to *relay* messages from `B` and `C`, as depicted by the relayed `mbus.hello` messages that are colored in teal and provide the `via` parameter.

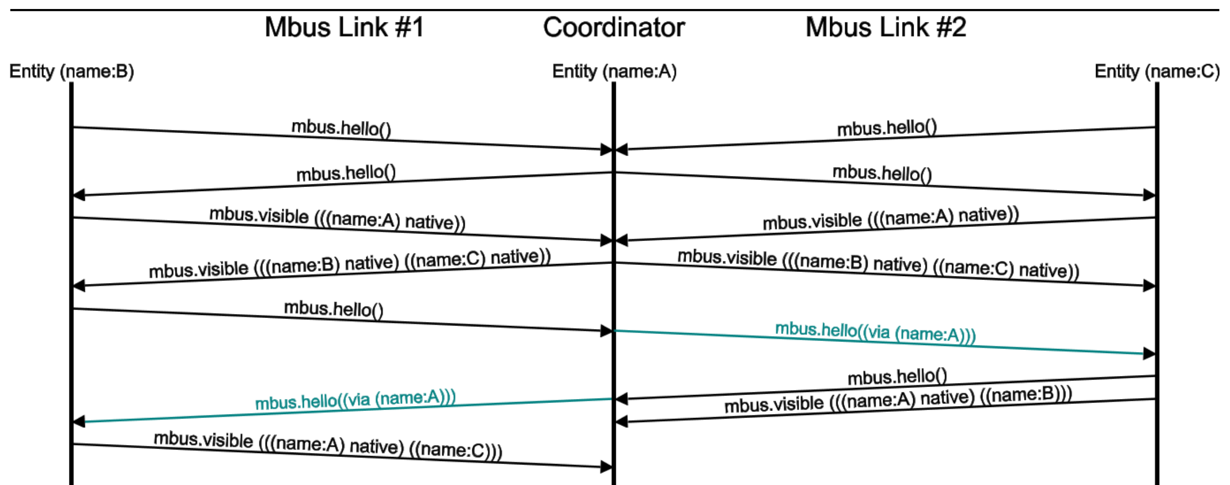

Figure 6.38: Visibility reports and relaying of mbus.hello messages

As a result, `B` becomes aware of `C` and vice versa, which is announced by an updated visibility report.

The coordinator uses the effective visibility to determine when full connectivity of the Mbus session has been achieved. It continues to use the native visibility to constantly monitor the overall connectivity and adapt its forwarding behavior when necessary.

6.5.3.4 Message Transmission and Forwarding

Mbus message transmission is conceptually extended to support multiple *interfaces* per Mbus entity. A regular Mbus entity (i.e., not the coordinator) provides a multicast interface and may have one or more unicast interfaces and uses only a single link. Each interface can be thought of similar to a link layer interface with routing table entries (based on Mbus addresses) pointing to this interface. The original Mbus design has a default route for all traffic pointing to the multicast interface and may have one unicast interface per known Mbus entity for the unicast optimization.

For Mbus sessions with partial multicast connectivity and a coordinator acting as a *hub*, the transmission behavior of Mbus entities needs to be adapted only slightly. Mbus entities that have full multicast connectivity with their coordinator do not need to change; the above rules just work. Mbus entities that have only unicast connectivity to their coordinator and no multicast connectivity to other entities (i.e. do not see any native visibility reports except from the coordinator) use their unicast interface to the coordinator as default interface. Mbus entities that have directly reachable multicast peers but only a unicast interface to the coordinator, create two default routes and thus duplicate their outgoing Mbus messages (except for those using the unicast optimization) transmitting them via multicast and sending them to the coordinator.

The coordinator is responsible for relaying Mbus messages and modifying `mbus.hello` messages in transit. Its forwarding functions are configured based on the visibility reporting and the connectivity discovery. It may have any number of unicast and multicast interfaces on different links. Each incoming Mbus message is examined with respect to its target Mbus address. If this is a multicast address, the coordinator forwards the message to all interfaces (except for

the one it has been received on) and forwards a local copy to its own application. Otherwise, the coordinator examines — based upon native visibility reports — to which interface (if any) the message needs to be forwarded; optionally, it hands the message to its local application.

6.5.3.5 Change and Failure Handling

As described in the Section 6.5.3.3, the coordinator permanently monitors effective and native visibility as reported from each endpoint. In case multicast connectvity improves, the coordinator will notice further entities reporting native visibility of each other and so the coordinator can reduce forwarding. If multicast connectivity is lost, incomplete effective visibility reports indicates that additional forwarding needs to be installed.

If the coordinator looses contact to an entity (e.g. by missing `mbus.hello()` messages), it may need to re-enter the *connectivity discovery* again. If this does not reveal ways to re-establish connectivity (e.g. because the entity is not longer reachable), the coordinator may attempt a DDA re-association or, ultimately, go through the entire service location procedure again to look for a different device offering the same services.

6.5.4 Summary

We have described scenarios and solutions for multiparty peering of service entities, considering the aspects *service discovery* and *group communication* and the special requirements for ad-hoc communication scenarios, such as changing network topologies and peer mobility. The discussion of these scenarios has shown that group communication is a desirable feature for many component-based services in local networks. However, it has also been evident that its implementation is not always trivial, because general multicast connectivity cannot be assumed and because ad-hoc communication scenarios require concepts that address potential changes in network topology while maintaining a continuous group communication session on the application layer.

We have extended the DDA service association protocol to support multiparty peering and have discussed the use of these extensions for establishing Mbus sessions. Using the Mbus as a basis, we have developed a group communication model that provides the concept of a group communication session that can encompass multiple underlying multicast and unicast sessions (that can be dynamically re-configured without affecting the overall group communication session).

One key aspect of this model is a central coordinating entity that manages the individual communication sessions, monitors entity visibility and provide message relay functions where appropriate. In summary, a coordinator would perform the following steps to provide Mbus connectivity for a group of entities that are distributed on multiple separate links:

1. Perform DDA-based session initiation, which includes the exchange of UDP unicast endpoint addresses with each entity.

2. Establish Mbus link to each entity and probe for multicast and unicast connectivity using the endpoint addresses exchanged before.

3. Report and determine visibility and update message relaying table.

4. For each message that is received and that is exclusively addressed to the coordinator, determine intended receivers (based on destination address) and relay message (based on message relaying table).

By adding some minimal changes to the Mbus protocol (adding new membership information messages) and by extending the Mbus implementation requirements slightly, Mbus entities can accommodate changing multicast connectivity, dynamic changes to the group membership and mixed multicast/unicast environments — characteristics that are frequently experienced in ad-hoc communication scenarios. While the central role of the coordinator implies that the group communication session depends on its existence, this is not considered an issue when the fate of the coordinator is coupled to that of the actual user application.

The link-state monitoring and forwarding functions that we have described are not to be misinterpreted as elements of a general layer 3 ad-hoc networking routing protocol such as AODV [Perkins02]. While routing protocols for ad-hoc networks provide multihop routing between potentially mobile hosts in order to establish and maintain an ad-hoc IP network, our approach is much simpler and highly efficient:

- The main goal is to provide group communication in scenarios where no comprehensive multicast connectivity between the intended group members can be established;
- the forwarding is restricted to specific messages of a selected application protocol (Mbus); and
- we rely on a special case, where there is always a central entity (the coordinator) that has a direct link to each of the session members.

We believe that our approach is useful for a variety of scenarios where multiple entities have to be brought together in a common session in order to provide a certain service for a user — and that its simplicity and efficiency provide significant deployment advantages over more sophisticated schemes.

Finally, there are some aspects that we have not discussed in detail in this section that are subject of our current and future research. Those particularly include:

Group security: As well as being able to add services to a running session, a user might want to exclude an entity from a session. While it is no problem to request the entity to leave, using application protocol specific means (e.g., by sending the `mbus.quit()` message or to issue a DDA dissociation request), the user cannot enforce the entity to leave. Group security approaches could be used to enforce that a former entity cannot participate anymore by changing the group security configuration, i.e., by initiating a *re-keying* process. For Mbus and DDA, there are two alternative solutions to re-configure the Mbus configuration: by designing corresponding Mbus extensions or by relying on existing DDA mechanisms. In the latter case the coordinator would have to dis-associate from all entities and initiate a new session with a new configuration. While this would solve the problem of excluding the former member, it would always imply a termination and re-initiation of the session.

Association protocol: The initial version of DDA (Section 6.4) uses HTTP for sending association and dissociation requests, e.g, an association request is implemented with an `HTTP-GET` request. For our first prototype implementation of the generalized DDA version that we have described in this section, we have mapped the invitation request to an `HTTP-POST` request. We are aware that there are existing protocols that are specifically designed for session initiation, such as the *Session Initiation Protocol* (SIP, [RFC3261]). Since most of the features of SIP such as user location and call routing are not required in the DDA context, the selection of an appropriate protocol needs more discussion.

Integrating with existing security infrastructures: The authentication step of a DDA association represents the authentication of a user to the service entity. The service entity uses the authenticated user identification for deciding whether the user is authorized to access the given service. In our current implementation we have implemented the authentication by verifying a user-name/password pair, relying on HTTP's digest authentication as a mechanism. In this implementation, the user-names and passwords of authorized users had to be configured on each service entity. This is obviously not a viable solution for large scale deployment. We are thus currently investigating possibilities for leveraging existing security and authorization infrastructures and AAA protocols such as Diameter [Calhoun02]. One issue that has to be considered is that not every service device will be provided with a permanent connectivity that would allow for accessing external AAA servers.

The presented multiparty peering mechanisms have been employed for the dynamic composition of conferencing endpoints, where a user can associate with multiple multimedia conferencing components in her environment and bring them into a common Mbus session. In Section 10.2.2, we describe a specific system that makes use of this concept.

6.6 Summary

In this chapter, we have presented the Mbus framework, consisting of the Mbus transport layer definition (Section 6.2), protocols and extensions for bootstrapping of Mbus sessions and for establishing Mbus sessions across network link boundaries (Section 6.4 and Section 6.5) and the higher layer Mbus interaction models (Section 6.3).

The Mbus transport layer has been designed as a group communication mechanism that provides a basic messaging service. It does not address services such as RPC or transaction semantics, let alone virtual synchrony as described in Section 4.1 that are costly to implement and not required by every application (in the application area of local coordination). The Mbus transport protocol is strictly asynchronous, in order to allow for a high degree of parallelism and a loose coupling of communication peers.

The basic messaging service employs the *soft-state* principle and can thus accommodate interrupted connectivity, dynamic group membership changes and does not rely on a strict client-server architecture. In addition to soft-state communication the protocol also provides a reliable, acknowledgment-based transport service. The *entity awareness* mechanism that is based on periodic multicasts provides a simple membership information service that allows to discover new entities and monitor their liveness during a session. Care has been taken to allow this mechanism to scale to a large number of entities.

We have tried to achieve a compromise between three design goals: *feature-richness*, *efficiency* and *simplicity* (allow for simple implementations). This has been realized by defining a basic transport service with mandatory protocol features, such as IP multicast transmission of messages, the awareness mechanism, the reliable transport service and baseline security mechanisms. Additional efficiency can be achieved by implementing enhancements such as the *unicast optimization* and the *acknowledgment piggybacking*. Care has been taken that implementations with different complexity levels can still inter-operate, thus allowing for a vast heterogeneity of Mbus-enabled devices that can communicate in a single session.

The security mechanisms can be tailored with respect to functionality (i.e., whether encryption should be used or not) and with respect to the employed cryptographic algorithms. Only message authentication is mandatory, because it is required for avoiding collisions and provides protection against malicious control messages that are injected into a session. Since the Mbus is likely to be used in isolated networks or for host-local communication where remote listeners cannot intercept messages, encryption is optional, again accommodating basic devices and simple implementations.

Taking the mechanisms for Mbus session set-up and the Mbus message passing and group communication services as a basis, we have generalized and formalized typical interaction models and communication relationships that are intended to support the development of Mbus applications. In Section 6.3, we have described the idea of providing informal interface descriptions of Mbus entities in *Mbus application profiles* and presented a survey of commonly used interaction models, some of which we have adopted from other distributed system solutions, such as the RPC mechanism.

One motivation for these higher layer protocols is to provide suitable communication mechanisms for applications that provide a distinct controller-controllee-relationship and require reliable means to invoke remote procedures. In addition, we have designed mechanisms that combine the flexibility of the Mbus group communication mechanisms with the usefulness of request/response-oriented communication. The *Mbus anycast* and the *coordinated Mbus RPC* protocols are examples for such mechanisms.

In this context, we have introduced the notion of different *control relationships* that are required by different types of applications. For example, Mbus entities can require a tight control relationship for being coordinated in a request/response-oriented fashion but can work independently, i.e., without a controller for generating soft-state events. The concept of *registering* with Mbus entities and redirecting event notifications is again designed to combine the flexibility of Mbus group communication with explicit control mechanisms. For example, the concept of a *default destination address* for event notifications and other Mbus commands allows for components that can provide useful functions without requiring an external controller, and the redirection concept supports the third-party deployment of modules by adapting their addressing scheme to the needs of a specific application.

In more dynamic scenarios, where Mbus sessions have to be established in an ad-hoc fashion and cannot necessarily be related to the environment of a single user, the distribution of the initial configuration of Mbus entities has to be done explicitly, i.e., by deploying an appropriate protocol. In Section 6.4, we have described an *Mbus bootstrapping* mechanism that addresses the issues of *locating Mbus-based services* in a network and *securely associating* with an Mbus service. The *Dynamic Device Association* (DDA) mechanism is targeted at initiating Mbus sessions with entities that do *not* belong to the presence of a single user. It therefore provides an

authentication and authorization concept thus allowing services to be configured with respect to access permissions and user identities. The DDA protocols itself are designed to be usable in ad-hoc scenarios and are very light-weight in order to allow for simple implementations. The announcement-based discovery mechanism also relies on soft-state communication and is addressing scalability and efficiency.

In Section 6.5, we have extended the DDA bootstrapping process for the initiation of multi-party Mbus sessions and have addressed the issue of providing Mbus connectivity over a set of possibly heterogeneous links that may not be completely inter-connected and that may be subject to changing link and topology characteristics. These extensions rely on the use case of a *user device* that can be attached to multiple network links and can detect the Mbus connectivity between entities on different links. Based on the observed connectivity the user device can activate message forwarding where required, thus providing a transparent Mbus group communication transport over a set of independent network links. This approach is intended to provide ad-hoc communication by a set of simple mechanisms for application layer message forwarding, based on a centralized setting, where the user device acts as the central message hub. The presented mechanisms are not intended as general *ad-hoc-networking* solutions or as substitutes for ad-hoc-network routing protocols but are designed for environments where IP link-local auto-configuration (or manual configuration) is in place and entities from different links should be enabled to participate in an Mbus session despite the lack of full multicast connectivity.

In summary, we believe that the Mbus framework provides an interesting combination of light-weight distributed systems characteristics, coordination facilities and group communication mechanisms. It has been advanced with respect to operational features such as bootstrapping and deployment in limited network environments, which allows its deployment for "real-world" applications, e.g., the development of end-user applications and reasonable complex systems such as multimedia conferencing gateways. The design of the Mbus deliberately deviates from traditional approaches for distributed systems such as the RPC mechanism or toolkits such as ISIS and its successors and foregoes static binding and virtual synchrony for flexibility and efficiency.

In the following, will validate this thesis by first discussing Mbus implementation aspects in Chapter 7, *Mbus Implementations* and subsequently presenting the results of some simulations and measurements in Chapter 8, *Evaluation*. In Chapter 9, *Mbus in Conferencing Systems* we describe the design of our conferencing endpoint architecture and the development of Mbus application profiles. The results of this work and its deployment for complete applications are presented in Chapter 10, *Mbus in Projects*.

Chapter 7

Mbus Implementations

In this chapter, we describe some existing Mbus implementations in order to highlight implementation requirements, typical strategies for implementing the Mbus transport layer and the Mbus Guidelines interactions and to discuss implications for programming models and application designs.

In Section 7.1, we describe generic Mbus implementation considerations and in Section 7.2, we describe requirements for underlying IP stacks and operating systems. This is followed by an overview of an Mbus C++ implementation in Section 7.3, an overview of an Mbus Java implementation in Section 7.4 and a description of some Mbus implementations for scripting languages in Section 7.5.

7.1 Implementation Considerations

In general, the Mbus message service relies on an *asynchronous* model, i.e., entities send messages and continue with their operation without waiting for a response. The *reliable transport* mechanism (Section 6.2.1.5) and the Mbus RPC service (Section 6.3.2.2) could be used to support synchronous communication, i.e., applications could employ a synchronous API to send corresponding messages.

However, this is not implied by the protocol definition. In fact, in order to allow for a maximum degree of concurrency and efficiency, Mbus implementations will typically provide an asynchronous interface. Applications that need a synchronous communication model can always implement the necessary synchronization mechanisms themselves.

The discussion of specific implementation approaches below will consider *single-threaded* and *multi-threaded* implementations. Single-threaded Mbus user space implementations (that share an execution thread with the application) require some sort of asynchronous event multiplexing in order to process the different events that can occur:

- Input/output events can occur for the Mbus UDP/IP communication, e.g., when a UDP datagram has been received, the corresponding Mbus protocol handler function must be called in order to process the message.

- Scheduled events must be processed, e.g., to send out an `mbus.hello` message at a given time or to trigger a retransmission of a reliable message.

In single-threaded UNIX environments with BSD-socket APIs, this event multiplexing will usually be implemented by an event management and notification service that relies on the UNIX `select` (or `poll`) system call.

In multi-threaded environments, each event source can be handled by an independent thread, i.e., every socket could be read from synchronously. Threads that manage event source and threads that handle events run in parallel, thus the same asynchronous model can be established.

In both cases, event handlers, e.g., functions in applications, are usually being invoked by some form of *callback* mechanism. For example, an application would register a dedicated function for processing incoming Mbus messages, and the Mbus implementation would invoke the corresponding function upon receiving and successfully processing an incoming Mbus message. Based on these assumptions, an Mbus implementation has to provide *at least* the following service interface:

`send_message`: An Mbus message is sent to the specified Mbus address. Besides the message itself and the address parameters, the type of the message needs to be specified (`unreliable`, `reliable`).

`register_receive_callback`: An application function is registered as a callback function that should be invoked upon receiving incoming Mbus messages.

It should be noted that the implementation of such a callback mechanism is highly programming language dependent. E.g., in the C programming language, the application would usually pass a function pointer to a corresponding callback registration function, whereas in some object-oriented programming languages other idioms are popular, e.g., defining a pure virtual function in a class that is derived from an abstract base class providing the required interface for Mbus applications. In our discussion of available Mbus implementations, we will find both approaches.

With this basic service interface in mind, we can list the following *functions* that a corresponding Mbus implementation needs to perform:

Management of UDP ports: One of the basic functions is the management of UDP ports. At least one socket (or a similar input/output abstraction) must be bound to the common Mbus port and multicast group membership must be managed. In order to be able to receive Mbus messages over IP unicast (through the Mbus unicast optimization mechanisms that we have described in Section 6.2.1.4) an implementation must manage two ports: one for receiving Mbus messages that are sent via IP multicast to the common port number and another port for sending messages to the group or to other entities and for receiving messages that are sent via IP unicast directly to the entity.

Mbus aliveness indications: The Mbus implementation must send the periodic `mbus.hello` messages via IP multicast to the Mbus broadcast address. For this purpose, the implementation must calculate the proper interval as described in Section 6.2.2.1. This calculation depends on the maintenance of an Mbus session membership list as described below.

Membership tracking: The implementation must maintain a list of currently visible Mbus entities in the session. For this purpose it must monitor the `mbus.hello` messages that are received from other entities taking the current expected retransmission interval into account.

Message parsing: An implementation has to process incoming messages by verifying the authenticity, decrypting the message. In addition, an implementation will typically transform the message from its textual representation into an appropriate data structure to facilitate the access to message fragments such as commands and command parameters for applications.

Message sending: For sending Mbus messages, an implementation must perform the following operations: It must encode the message to textual representation, encrypt the message (if configured for the current session) and calculate and prepend the message digest. Depending on the destination address, the implementation must make a decision which underlying transport mechanism to use.

A message with a destination address that does not uniquely specify a single recipient (based on the current set of known entities) must be sent via IP multicast (or broadcast if configured). If unicast optimization is supported, the implementation sends messages with unique destination addresses via unicast to the unique endpoint address of the recipient.

Handling retransmissions: Messages that are sent reliably must be stored for retransmission until a corresponding acknowledgment has been received or until the maximum time frame for retransmissions and acknowledgments has past. If no acknowledgment for the last message has been received after the specified time, the sending operation must be canceled and an failure condition should be signaled.

Handling acknowledgments: The reception of reliable messages must be acknowledged as described in Section 6.2.1.5, i.e., the message must either be acknowledged immediately by sending a dedicated acknowledgment message or the acknowledgment must be postponed in order to piggy-back the acknowledgment onto another message (taking the timeouts for acknowledging reliable messages into account).

7.2 Basic Implementation Requirements

Having described the general services and functions in Section 7.1, we will now list the main requirements for host platforms that must be fulfilled to provide a basis for Mbus implementations. Essentially, the requirements can be divided into two areas: *IP communication services* and *other operating system services*.

IP communication services: The Mbus protocol is layered on top of IP/UDP. Therefore, a software platform must provide the possibility of sending and receiving UDP datagrams. The Mbus protocol can be used with either IPv4 or IPv6.

Since the Mbus is using IP multicast, software platforms should also support IP multicast. Support for local IP multicasting includes sending multicast datagrams, joining multicast groups, receiving multicast datagrams, and leaving multicast groups. Implementing the Mbus protocol requires a host IP implementation that reaches conformance level 2 for IP

multicast as defined by RFC 1112 [RFC1112]. This includes an implementation of IGMP (*Internet Group Management Protocol*)[1].

In case IP multicast is not available, the Mbus protocol can alternatively be layered on top of IP broadcast by sending messages to the generic multicast address of a network (IPv4). For IPv6, either the node-local or the link-local multicast should be used.

Other operating system services: For implementing the Mbus protocol in single-threaded environments, services such as asynchronous event multiplexing and timer event notification are required. In multi-threaded environments, the corresponding functionality can (potentially) be provided by the Mbus implementation itself, however, this is not considered to be the most straightforward and most efficient solution. Therefore, event multiplexing and timer management facilities will be useful in these environments nevertheless.

These services are needed for the following functions:

Receiving on multiple UDP ports: An Mbus implementation will typically receive UDP datagrams on two UDP ports: Datagrams that are sent to the IP multicast group are received on the standard Mbus port, and datagrams that are targeted to a specific entity are received on a unique port per entity. This allows for multiple Mbus implementation instances on the same node that can all be addressed independently. These two ports have to be monitored and (e.g., if a socket API is present), messages have to be read from the sockets as soon as new datagrams have been received.

One popular model for implementing the monitoring of multiple input sources in single-threaded environments is called *asynchronous event multiplexing* — a concept that avoids the blocking reading of a single input source by defining services that provide the monitoring of a set of input sources and can notify the application as soon as input data is available. For UNIX systems, this service is usually provided by a program's *main loop* that relies on the use of the UNIX system call `select` or `poll`.

For UNIX-like environments, it must be possible to either have the Mbus implementation deploy this mechanism itself, or — in case the application that the Mbus module is to be embedded in already uses it's own `select`-based main loop — to integrate the Mbus module into the application's main loop.

Timer for Mbus protocol events: An Mbus implementation has to perform certain actions in well-defined time intervals. For example, periodic `mbus.hello` messages have to be sent and the liveness of other entities has to be checked after certain time periods.

For UNIX-like environments, the notification of timer events is often achieved by using the `select` system call. Other platforms may provide other services for timer

[1] IGMP support is required to accommodate switched networks that are connected by switches that support *IGMP-snooping* in order to limit distribution of multicast traffic. If such switches were in place and hosts did not indicate their interest in receiving packets sent to the Mbus multicast group, they would not receive messages from entities from other links.

event notification.

7.3 C++ Implementation

We have developed a C++ implementation that provides all specified Mbus features and allows for the development of applications in a very simple and comfortable way. It is available as a library that can be linked with application specific code in order to build a complete application. The C++ implementation itself covers the Mbus transport specification. Additional implementations for the higher layer services (*Mbus Guidelines*) are available that provide additional features such as support for RPCs and other interactions.

7.3.1 Design

The basic Mbus transport implementation itself is single-threaded and provides a strict *asynchronous* interface. I.e., application code will never be blocked when calling an API function to send a message and there is no mechanism to wait for the reception of a message by calling a corresponding API function. Instead, the Mbus implementation provides an API to register callbacks that should be called when incoming messages have arrived.

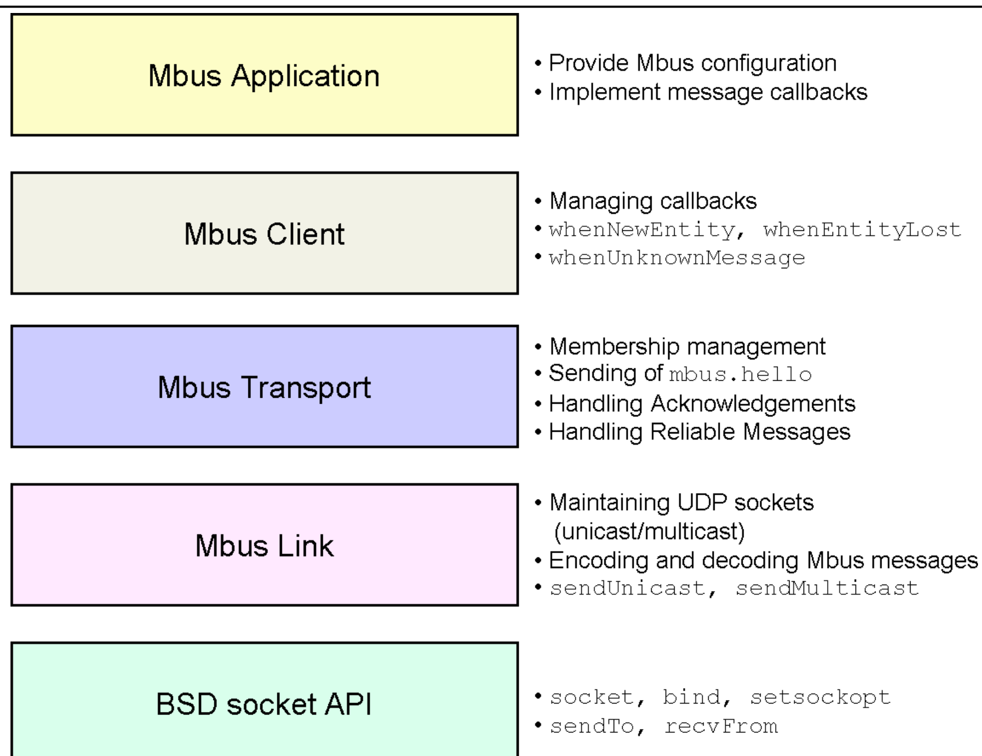

Figure 7.1: Layers in the C++ implementation

Figure 7.1 provides on overview of the implementation approach. We have split the implementation into several layers:

- The socket API is part of the operating system and provides the usual system calls to create sockets, to bind sockets to endpoint addresses, to manage multicast membership and to send to and receive from other endpoints.

- The Mbus link layer manages the UDP sockets for one Mbus application. It creates a socket for receiving multicast messages and a socket for receiving unicast messages and for sending. These sockets are registered with an event notifier that calls a certain callback function as soon as data has been received.

  For messages that are to be sent, the Mbus link layer transforms Mbus messages from their internal C++ object representation to text representation and performs encryption and message digest calculation. A sender has to specify whether the message should be sent to the multicast group or to a specific endpoint.

  For receiving messages, the Mbus link layer can be parameterized with a list of endpoint addresses (IP address and port tuples) that represents an *address filter*. Messages with an UDP/IP source address that match one address of the filter list are discarded. This is mainly used to configure the Mbus link layer to discard its own messages that have been sent to the multicast group[2].

  Messages that are not filtered are checked for authenticity, decrypted and decoded before they are passed to the next higher layer. Messages that fail the authenticity check or cannot be decoded are signaled as an exceptional condition.

  The Mbus link layer does not know anything about Mbus addresses nor the structure of messages. It also does not perform any management functions such as sending `mbus.hello` messages. Its sole purpose is to manage the UDP ports and to encode/decode messages.

- The Mbus transport layer provides the main functionality and implements the functions that we have described in Section 7.1.

  It maintains a list of currently known Mbus entities (and a mapping of their Mbus addresses to endpoint addresses). Based on this information, it is decided whether a message is sent via unicast or via multicast using the interface of the lower link layer.

  The Mbus transport layer provides the periodic sending of `mbus.hello` messages and the adaptation of the interval depending on the observed group size. It also manages reliable messages by keeping a copy of sent messages and retransmitting them if necessary.

  For upper layers, it provides a single method to send a message. This method is responsible for completing the Mbus message header (setting the sequence number and other fields) and for selecting the appropriate link layer transport based on the destination address.

  Received messages are forwarded through a callback to an upper layer implementation. Mbus management commands such as `mbus.hello` and `mbus.bye` are not forwarded but translated into event notifications such as `new entity` and `entity lost`. I.e.,

[2] The socket option `MULTICAST_LOOP` must be enabled on the sending socket in order to allow messages to be received by other entities (represented by other processes) on the same system.

the Mbus transport layer monitors the `mbus.hello` messages from other entities and when they have not been received for a certain period of time, the corresponding entity is removed from the set of currently known entities and an `entity lost` event is generated.

- The Mbus client layer is the interface to the application. It provides the possibility to register Mbus commands with certain callback functions. When a message is received from the transport layer that provides a matching Mbus command, the command (and the message header) is passed as an argument to the specified callback function. Commands that have not been registered are passed to a default message handling function.

  In addition, the Mbus client layer provides callbacks for events from the lower transport layer such as `whenNewEntity` for signaling the appearance of a new Mbus entity.

These layers are implemented as C++ classes. The Mbus client (`MClient`) layer is implemented as an abstract base class that application programmers can derive application specific classes from in order to redefine the functions `whenNewEntity` etc.

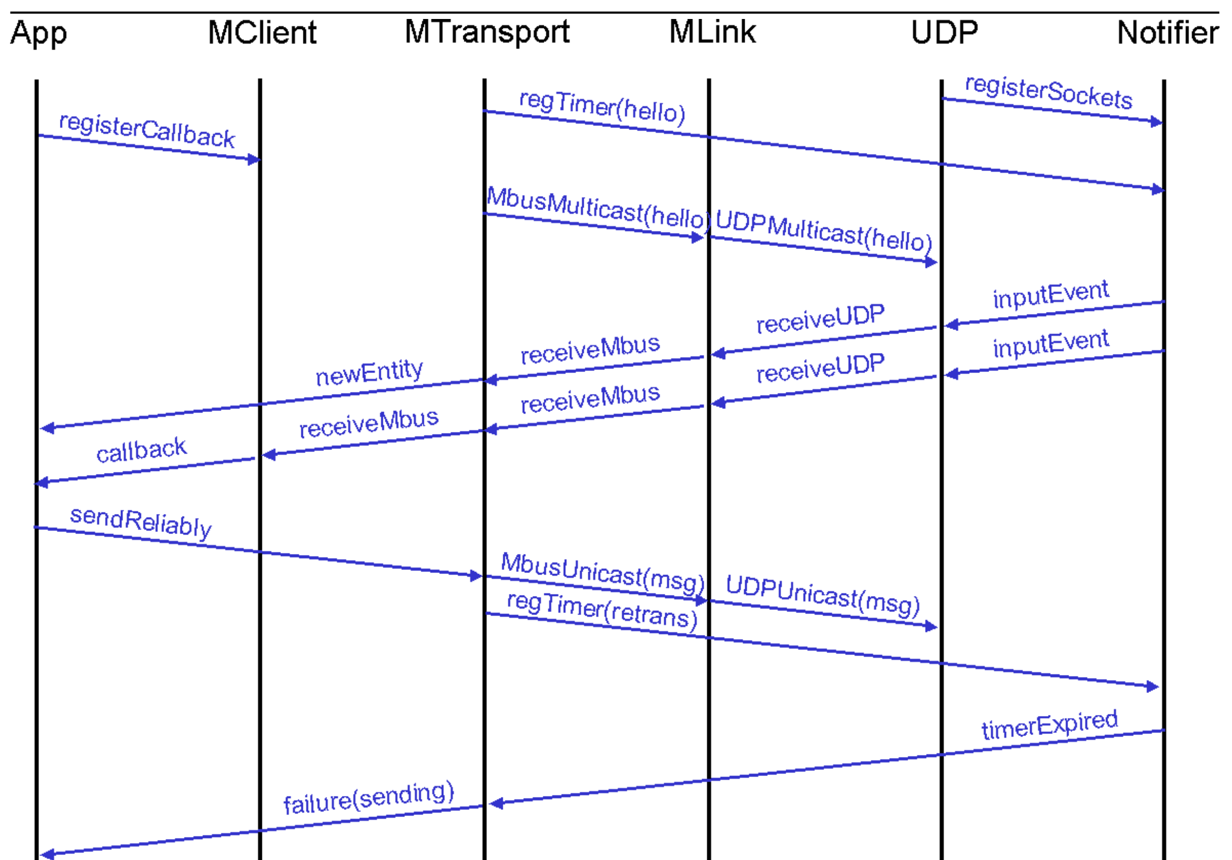

Figure 7.2: Layer interaction in the C++ implementation

Figure 7.2 illustrates the asynchronous, event-based invocation of a protocol handler and an application code callback function. The figure provides the stack of interacting classes, and the top of the stack (the application code) is at the left hand side. The `Notifier` is actually not

part of the stack — we have included it here to show the interaction of the protocol layers with the event management system.

In the initialization phase, both the UDP socket layer and the Mbus transport layer register their event sources (sockets and timers) with the notifier. The application object registers a callback for a specific Mbus command name with its `MbusClient` base class. The transport layer emits a first `mbus.hello` message (represented as a C++ object) that is encoded by the Mbus link layer and send as a UDP message via IP multicast to the Mbus multicast group.

The notifier then signals an input event to the UDP layer. The UDP layer reads the received message by calling the `recvfrom` system call and passes the message to the Mbus link layer by calling a corresponding member function. In this case, we assume that the incoming message contains a single `mbus.hello` command from another Mbus entity. The link layer object authenticates the message, decrypts and decodes it and passes it as a C++ object to the Mbus transport layer. The Mbus transport layer notices that the message is the first message received from the sender and notifies the application by calling the `newEntity` method. In addition, the transport layer starts timers for monitoring the aliveness of the new entity, which is not shown here for the sake of clarity.

As a next step, the application initiates the reliable sending of a new message by calling the method `sendReliably` of the Mbus transport layer. For this call, the application must also specify a unique Mbus destination address. The transport layer retrieves the corresponding endpoint (UDP/IP) address and calls the method `sendMbusUnicast` of the Mbus link layer. In addition, the transport layer registers a timer for triggering a possible retransmission with the notifier. The link layer encodes and encrypts the messages, adds the authentication header and sends the message to the specified IP address and port number.

For this example, we assume that the sending of the message (or the sending of the corresponding acknowledgment) has failed and that the maximum number of retransmissions (not shown here) has been reached. Eventually, the final retransmission timer expires, which is signaled to the transport layer. The transport layer removes the message from the list of messages that have to be acknowledged and indicates the failure condition to the application.

We can observe from this example that the failure indication is generated asynchronously to the original sending of the message. The calling application code does receive the result of the sending operation as a function call result when sending the message but is notified through an asynchronous function call from the lower layer. The same would hold for indicating success on a reliable message transmission (which we do not do in the C++ implementation).

This fundamental asynchronous model is a main characteristic of the C++ Mbus implementation and is also applied to the implementation of the Mbus Guidelines interactions that we describe in Section 7.3.3.

Figure 7.3 is a sample C++ code fragment that demonstrates a common usage scenario: An application-specific class (`Simple`) is derived from a base class `MbusClient` (part of the C++ Mbus library). This base class provides some pure virtual member functions that have to be defined by derived application classes. E.g., the member function `newEntity` is called by the Mbus code on detection of a new entity in an Mbus session.

For receiving messages, the class `MbusClient` provides a dispatching mechanism (not shown here). Applications register commands with handler functions (callbacks) that will be called by the Mbus code on receiving a message that contains the registered command.

The class `MbusClient` provides additional member functions for sending messages that

are also not shown here.

```
class Simple: public MbusClient {
public:
  Simple(const MsgCoder& c, const MAddress &a)
    :MbusClient(a,c) {
    dispatchOn("mbus.quit",callback(*this,&Simple::whenQuit));
  };
  bool unknownMessage(const MMessage& m)
    {cout << m << endl; return true;};

  void newEntity(const MAddress& m)
    {cout << "new entity: " << m << endl;};

  void lostEntity(const MAddress& m)
    {cout << "lost entity: " << m << endl;};

  void whenError(const MbusErrorContext& c)
    {cout << "Mbus error: " << merror[c.theError] << endl;};

  bool whenQuit(const MMessage& m)
  {
    loopInstance()->stop();
    return true;
  }
};

int main(int argc, char* argv[])
{

  Authority* auth=initMbusAuthority();
  MsgCoder c;

  c=auth->msgCoder();

  INIT_NOTIFIER;

  Simple simple(c,MAddress("control","engine","simple"));

  // enter main loop

  NOTIFIER_LOOP;

  return 0;
}
```

Figure 7.3: An Mbus C++ application

7.3.2 Integration into Applications

User-space protocol implementations usually have to address two issues to be usable for a wide range of applications on different platforms:

- The protocol implementation specific event management system must be integrated into the event management system of applications. For integrating a protocol implementation into an existing application, the protocol library has to interface with an existing event notifier.

  This is especially an issue for single-threaded applications where all events must be distributed from a central notifier entity in order to allow for a fair sharing of the execution thread. For multi-threaded applications, each module that runs in its own thread can theoretically rely on its own, independent event notification mechanism. However, this is

not always practical, e.g., in scenarios where a single event must be delivered to multiple subscribers.

- Another issue that must be considered is that event handling itself is quite platform specific, similar to operations that rely on system calls such as socket communication. For example, UNIX-based systems often rely on the `select` system call for querying for events on file descriptors. In the Windows family of operating systems event queues are part of the operating system and provide a different paradigm for monitoring events from multiple sources[3].

  Therefore it not advisable to tie a protocol application to a fixed event notification system, if the protocol implementation should be portable to different platforms. Instead it must be provisioned that the complete event notification mechanism can be replaced by another mechanism that fits the requirements of a specific platform.

In addition to the Mbus communication, Mbus applications typically perform other communication tasks or provide a user interface that can generate events asynchronously. For example, Mbus-based applications in conferencing systems will probably communicate with other applications over the application-specific horizontal protocol, e.g., RTP, and will provide a GUI for rendering application data and for controlling the application. This means that the Mbus event notification system must accommodate that the Mbus sub-system be integrated into a larger application context. We have addressed this issue by a two-pronged approach:

- The Mbus implementation itself relies on a notifier abstraction providing a simple interface that is designed to meet the specific needs for Mbus implementations: it provides the possibility to register timer events and file descriptors.

  This interface can be used to provide notifier implementations for different specific environments, e.g., UNIX and Windows platforms.

- For UNIX based systems, we have developed an event notification mechanism that provides the usual interface of registering callback functions for events on file descriptors and for timer expiration events. Once an application has been initialized, e.g., the `MbusClient` object has been created, control is passed to the notifier that subsequently runs in a loop for dispatching occurring events. This notifier has been made available as an object code library under the name `libnotifer`, hence we refer to it using that name in the following.

  In addition to this normal event notification service, `libnotifier` provides an interface that allows *external* notifiers to take over the control of the notifier's main loop. In this case, all registered event sources will be passed on to the external notifier that can for example be represented by an X11 main loop. This means, the Mbus implementation (and other `libnotifier` clients) use `libnotifier` as an interface to register event sources and callback functions, where the actual main loop is run by another notifier, e.g., the X11 main loop.

[3]For compatibility reasons, Windows operating systems also provide a `select` function, which is often used for porting TCP/IP based UNIX applications to a Windows platform.

The combination of the C++ Mbus implementation and `libnotifier` has evolved into a core framework for many applications that have been developed at our group during the last five years. Some of the results are described in Chapter 10, *Mbus in Projects*. The `libnotifier` approach has led to the development of many Mbus based GUI applications using different toolkits and notifier mechanisms without requiring changes to the Mbus protocol implementation.

The C++ implementation itself is available for UNIX and MS Windows based systems. For Windows based systems, we have implemented a direct integration with the Windows event management, i.e., without employing `libnotifier`.

7.3.3 Mbus Guidelines Implementation

Based on the C++ implementation of the basic Mbus messaging service (as defined by RFC 3259), we have developed an implementation of most of the interaction schemes of the *Mbus Guidelines* that we have described in Section 6.3.[4]

The design of the Guidelines implementation (that has been made available as an object code library called `libmbusapp`) follows the design principles of the basic transport implementation: the overall operation is completely asynchronous, i.e., applications register services such as an RPC implementation together with a callback function with the protocol management layer and the application function will be invoked upon the reception of a corresponding message. As another example, when calling an Mbus RPC at a remote entity, the application also registers a callback function that should be called for processing the RPC result message. In this case, the protocol layer performs functions such as correlating RPC answers to requests (and generating proper RPC messages and return commands).

Figure 7.4 depicts an application class called `Simple` that is derived from the `libmbusapp` class `RPCServer` and from the class `MbusClient` of the transport implementation. The constructor of the `Simple` class registers an RPC called `doIt` by calling the method `dispatchOnRPC`, a member function of `RPCServer`. The `RPCServer` class registers the corresponding Mbus command with the command dispatcher of `MbusClient`.

Upon receiving an RPC, the `RPCServer` object verifies the formal RPC command syntax and delivers the RPC by invoking the registered callback function, in this case, this is the member function `whenRPC` of the `RDisp` class. In this example, the RPC handler function does nothing excepting for generating an RPC result.

Figure 7.5 depicts an example of an application that *invokes* a remote procedure. The invocation is performed in the function `Simple::whenFoo` by constructing a `FooStatus` object and passing it to `RPCClient::send` (the application class `Simple` is derived from `RPCClient`. The application class `FooStatus` is derived from the `libmbusapp` class `RPCStatus` and defines a few functions for processing RPC command results. For example, the method `whenOK` will be called when an RPC return command with status `OK` has been received.

The constructor of `FooStatus` specifies a maximum waiting time of 10000 milliseconds. If no corresponding RPC return command has been received in this time frame, the function `whenTimeout` will be called for notifying the application.

[4]The C++ Mbus Guidelines implementation has been written by Olaf Bergmann.

```
class RDisp : public RPCDispatcher {
public:
  bool whenRPC(const RPCCmd &, const MList *args);
};

bool RDisp::whenRPC(const RPCCmd &cmd, const MList *args) {
  std::cerr << __FUNCTION__
            << ": got rpc (" << cmd.cmd << "," << cmd.id() << ")"
            << std::endl;

  return rpc_return(cmd,new RPCResult(RPCResult::RPC_OK)) == RPC_OK;
}

class Simple: public Mbus_Application::RPCServer {
public:
  Simple(const MsgCoder& c, const MAddress &a)
    : RPCServer(a,c), MbusClient(a,c)
  {
    dispatchOnRPC("doIt",&rdisp);
  }

  bool unknownMessage(const MMessage& m)
  {std::cout << m << std::endl; return true;};

  void newEntity(const MAddress& m)
  {std::cout << "new entity: " << m << std::endl;};

  void lostEntity(const MAddress& m)
  {std::cout << "lost entity: " << m << std::endl;};

  void whenError(const MbusErrorContext& c)
  {std::cout << "Mbus error: " << merror[c.theError] << std::endl;};
protected:
  RDisp rdisp;
};
```

Figure 7.4: Mbus RPC server application class

This reveals the completely asynchronous procedure for RPC interaction that is provided by `libmbusapp`: A calling entity issues the RPC and then returns the control to the calling function and thus eventually to the notifier's main loop. The RPC return command is hence treated as any other Mbus command that is received and processed asynchronously.

This asynchronous model and the basic principle of having the `libmbusapp` deal with message construction, message decoding, correlating answers to requests and managing timers is applied for the implementation of other Mbus Guideline interactions as well. We cannot provide a complete description of the Guidelines layer here, but have to refer the reader to the Mbus web page (`http://www.mbus.org/`), where all Mbus implementations are available as source code packages.

7.3.4 Summary

Summarizing, we can state that the C++ Mbus implementation has evolved into a framework for the development of locally distributed systems that is offering a strictly asynchronous programming model that fits well into the single-threaded event multiplexing model for UNIX based systems.

The provision of a callback function based message-dispatching service for both basic Mbus commands and higher layer commands has proven to be an efficient tool for the fast development of robust applications. Application programmers can specify the interface of their Mbus applications by registering the corresponding command names along with the respective callback functions and the protocol implementation provides the message delivery, the correlation

```
class Simple: public Mbus_Application::RPCClient {
public:
  bool whenFoo(const MMessage &);

  Simple(libnotifier::Notifier &n,
         const MsgCoder& c,
         const MAddress &a)
    : RPCClient(n,a,c), MbusClient(a,c) {
    RPCClient::dispatchOn("foo",callback(*this,&Simple::whenFoo));
  };

  bool unknownMessage(const MMessage& m)
  {std::cout << m << std::endl; return true;};

  void newEntity(const MAddress& m)
  {std::cout << "new entity: " << m << std::endl;};

  void lostEntity(const MAddress& m)
  {std::cout << "lost entity: " << m << std::endl;};

  void whenError(const MbusErrorContext& c)
  {std::cout << "Mbus error: " << merror[c.theError] << std::endl;};
};
// ------------------------------------------------------------
class FooStatus : public Mbus_Application::RPCStatus {
public:
  FooStatus(MCommand *c):Mbus_Application::RPCStatus(c,10000) {}

  void whenOK(const RetValue &m)
  {std::cerr << __FUNCTION__ << ": got msg: " << *m.msg << std::endl;};

  void whenERROR(const RetValue &m)
  {std::cerr << __FUNCTION__ << ": got msg: " << *m.msg << std::endl;};

  void whenUNKNOWN(const RetValue &m)
  {std::cerr << __FUNCTION__ << ": got msg: " << *m.msg << std::endl;};

  void whenTimeout()
  {std::cerr << __FUNCTION__ << ": expired!" << std::endl;};
};
// ------------------------------------------------------------
bool Simple::whenFoo(const MMessage &m) {
  MCommand  *cmd2 = new MCommand("baz",new MSymbol("abcd"));
  FooStatus *fs   = new FooStatus(cmd2);
  MbusError e     = send(m.header.source,fs);

  if (e != MBUS_OK)
    std::cerr << __FUNCTION__
              << ": send:" << strerror(e) << std::endl;
  return true;
}
```

Figure 7.5: Mbus RPC client application class

of answers to requests and other management tasks.

For large-scale Mbus applications with many communicating entities and complex interfaces, we have experienced that is helpful to be able to automatically verify the structure of received messages and have therefore defined a validation function. This function can check that a given Mbus command conforms to a specified parameter list structure. This mechanism has helped us to reduce the necessity for manual verifications and has thus simplified the development of application programs.

With these mechanisms we have been able to develop quite complex applications as federations of multiple Mbus entities such as multimedia conferencing gateways as we have described in Chapter 10, *Mbus in Projects*. For these applications we have not really identified a need for formal interface descriptions and automatic code generation. Given the standardized interaction schemes that are defined by the Mbus Guidelines and that we have implemented, an automatic generation of *client and server stubs* would have been possible quite easily, however the C++

abstractions that we have provided by the protocol implementations have proven to be powerful enough to directly code the necessary functionality in a concise and expressive way.

7.4 Java Implementation

In addition to the C++ implementation, a Java implementation of the Mbus base protocol and the higher layer protocols have been developed at the University of Bremen.[5] The supported features are similar to those provided by the C++ implementation, however the implementation provides some interesting characteristics in comparison to the C++ implementation that we want to describe in the following.

One fundamental difference to the C++ implementation is that the Java implementation itself is multi-threaded, relying on standard Java virtual machine multi-threading capabilities. For example, timers for periodic events such as timers for sending `mbus.hello` messages are implemented as pseudo-parallel threads. Hence the Java Mbus implementation does not rely on a central event management system such as the C++ implementation, but relies on a multi-threading approach that is common in the Java environment.

The Java implementation also provides a layered design: the *Transport Layer*, the *ConnectionControl Layer* and the *Service Layer*. The *Transport Layer* provides transport of Mbus messages (with optional encryption) and integrity by the use of message digests. The *ConnectionControl Layer* adds support for reliable messages through retransmitting and takes care of presence information (regular announcements of the `mbus.hello` messages and processing received `mbus.hello` messages from other entities). Together the *Transport Layer* and the *ConnectionControl Layer* implement the functionality of the Mbus transport specification.

The third layer provides support for the *Mbus Guidelines* interactions such as RPCs, properties and events and some additional message processing utilities for verifying the message syntax and facilitating access to message fragments. It is recommended that Java Mbus applications be built onto this layer.

Despite of the multi-threaded architecture, the general programming model is still asynchronous: an application class has to implement interfaces that are defined in the Java Mbus package. This interface comprises functions that will be called asynchronously, e.g., on the reception of messages. For example, in order to receive RPCs, an application class has to be derived from a class called `RPCListener` and has to define the virtual function `handleRequest` that is called when the Guidelines layer has received and validated an incoming RPC message. In order to trigger the sending of a return command, the application code explicitly calls the method `rpcReply` of the Guidelines layer class `MbusInterface`. Figure 7.6 depicts (a fragment of) a sample Java Mbus application. Mbus messages are sent in the method `readInput`, and incoming messages are processed in the method `newMbusMessage`.

One interesting property of the *ConnectionControl* layer is the concept of so-called *virtual Mbus interfaces*, an optimization that allows multiple Mbus entities in a single application to use the same *ConnectionControl* layer. The *ConncetionControl* layer can perform the Mbus management tasks such as sending `mbus.hello` messages for multiple application entities. In addition, the layer is able to optimize the message delivery for messages that are exchanged

[5] The Java implementation has been developed by Stefan Prelle.

```
public class MShell extends Thread
    implements AddressInterfaceListener, ConnectionControlListener {

    private  static String[][] ADDRESS = {{"app","mshell"},
                                          {"id", Address.createID()}};
    private ConnectionControlLayer layer;
    private AddressInterface iface1;
    private Address        myAddress, allTargets;
    private boolean        showHellos;
    private int            seq = 1;
    //---------------------------------------------------------------
    public MShell(boolean showHellos) throws MBusException {

        myAddress = new Address(ADDRESS);
        allTargets = new Address("()");

        layer = new DefaultConnectionControlLayer(this);
        iface1= new ActiveAddressInterface(myAddress, this);
        layer.addInterface(iface1);
        readInput();
    }
    //---------------------------------------------------------------
    private void readInput() {
        BufferedReader in =
            new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            String line = in.readLine()+"()";
            Command command = Parser.parse(line);
            iface1.broadcast(command);
        }
    }
    //---------------------------------------------------------------
    public void newMBusMessage(Message mess) {
        System.out.println("\n"+mess);
    }
    //----------------------------------------------------------------
    public void deliverySuccessful(int seqNum, Message mess){
        System.out.println("Message "
                           + mess.getHeader().getSequenceNumber()
                           +" has been delivered");
    }
    //----------------------------------------------------------------
    public void deliveryFailed(Message mess){
        System.out.println("Message "
                           + mess.getHeader().getSequenceNumber()
                           +  " has not been delivered. ");
    }
    //----------------------------------------------------------------
    public void newEntity(Address addr) {
        System.out.println("NEW ENTITY: " + addr);
    }
} // MShell
```

Figure 7.6: Sample Java Mbus application

between the locally attached Mbus entities. These messages can be delivered directly in their Java object representation without the need for encoding (and subsequently decoding) them first, thus allowing for an efficient implementation of intra-application Mbus communication. Java-based application components can use the same Mbus implementation interface, regardless whether they are distributed in separate processes or composed into a single application.

7.5 Mbus Implementations for Scripting Environments

In addition to the C++ and Java implementations, a set of different Mbus implementations for scripting languages has been developed, e.g., Tcl and Python implementations. While the C++ and Java implementations are intended for non-trivial applications that fall under the *programming in the large* category, scripting environments provide benefits for rapid prototyping and are sometimes more suited to the development of specific application modules such as graphical

user interfaces.

In particular, the decomposition of applications into Mbus components is a strong argument for Mbus scripting environments because it enables application programmers to develop the core functionality in a ”real” programming language such as C++ and Java and implement the user interface in a scripting language, which is often more convenient because some scripting languages are explicitly targeted at the simple and efficient construction of user interfaces. For example, the success of the scripting language Tcl [Ousterhout94] (see also Section 4.4.3) is not based on the beauty of the language design[6] but on the fact that it has provided a binding to the Tk toolkit for the X11 Window System and thus allowed for the easy creation of graphical user interfaces on UNIX/X11 platforms. Separating the functionality from the user interface has the additional advantage that it increases the re-usability of the functional components, as user interface development is still a platform specific aspect of application development. If an existing user interface module cannot be ported to a new platform, it is still possible to rewrite the user interface components as it is decoupled from the core functionality anyway.

The Mbus scripting language implementations differ significantly with respect to their fundamental design. The Tcl implementation is actually implemented by providing a Tcl interface to the C++ implementation, i.e., the Mbus C++ implementation is linked to a Tcl interpreter that has been extended by some Tcl constructs that allow for accessing the C++ code. The rationale for this approach is to implement the time-consuming tasks such as message encoding and encryption efficiently in C++ while offering the programmer a simple interface to these functions.

Programming Mbus-based Tcl applications has been made extremely simple. In essence, an asynchronous model is employed, similar to the C++ implementation itself. The programmer can specify the Mbus address of her application and subsequently register Tcl procedures as callback functions for specific Mbus command names. The program runs in the Tcl main loop that is used instead of the regular main loop of the C++ implementation. Although providing a Tcl interface to the C++ Mbus library has in general proven to be a viable solution for rapid application development, this approach has the principal disadvantage that depends on a working Mbus C++ implementation for a specific platform. This has been a problem in the first years after the C++ standardization was completed (1998). Different compilers provided different degrees of standard compliance, which has sometimes required adaptations to the code.

The Mbus Python Implementation

The Mbus Python implementation[7] follows a completely different approach: it is a native Python implementation that can thus be used with any Python interpreter and is not depending on additional object code modules.

Python is an easy to learn scripting language that provides efficient high-level data structures and supports a simplified object-oriented programming style. Similar to Tcl, Python bindings for different GUI toolkit exist, such as an Xt and Qt binding. Typical for a scripting language, Python provides a dynamic type system. Compared to Tcl, it has more features of a ”real” programming language such as different data types and control mechanisms such as exceptions.

[6]Some would even argue that Tcl has been successful *despite* its language design.

[7]The Mbus Python implementation has been developed by Kevin Loos for the Hausgeist project.

The Mbus Python implementation is also following an asynchronous model and provides mechanisms to register user defined functions as callback functions for handling incoming Mbus commands. The implementation provides support for some Mbus Guidelines interactions such as RPCs.

The Python implementation has been used successfully with different GUI toolkits, for example with GTK (the GIMP toolkit[8]), with the cross-platform GUI toolkit wxWindows[9] and with Trolltech's Qt cross-platform application framework[10] and is currently the most often used Mbus scripting implementation. Our deployment experience has shown that it provides a good compromise between ease of use, feature-richness and efficiency. Hence we have used this implementation for GUI-based Mbus applications on less powerful devices that are restricted in both CPU performance and storage space such as PDA platforms, which we describe in Section 10.2.

In addition to the Tcl and the Python implementation, there is also an Mbus implementation in Perl[11] that is also a native implementation, i.e., completely written in Perl. The Perl implementation follows a comparatively simple approach and does not provide new characteristics compared to the Tcl and Python implementation, so we will not discuss it here in more detail.

7.6 Summary

The Mbus protocol has been implemented many times, and in this chapter we have presented a few representative approaches. There are in fact even more implementations than the ones we have mentioned so far in this chapter:

- An Mbus C implementation is part of the UCL (University College London) *common multimedia library* (commonlib) that is used as an essential building block for Mbone tools such as RAT and VIC (see also Section 10.1.2).[12]

  In fact, we have performed a number of (successful) interoperability tests with the UCL implementation and our different implementations and have built applications that included UCL-Mbus based entities.

- A C# implementation for Microsoft's .NET framework has been developed within the Hausgeist project.[13]

- The author has also developed another C implementation that is intended to be extremely portable and is designed to run in very limited environments. For example, we have

[8] `http://www.gtk.org/`

[9] `http://www.wxwindows.org`

[10] `http://www.trolltech.com/`

[11] The Mbus Perl implementation has been developed by Markus Germeier

[12] The UCL Mbus implementation has been developed by Colin Perkins.

[13] The Mbus C# implementation has been developed by Daniel Losch.

successfully deployed this implementation on a small one-chip-computer that has been used for device control, e.g., in home networking scenarios.

During the development and the deployment of these implementations we have been able to advance useful implementation strategies and convenient service models that Mbus implementations should offer to application programmers. We have also learned how to design applications in a way that allows for flexible re-use of their components in other application contexts.

One of the key insights that we have gained is that it is possible to develop complex distributed applications that are based on a completely asynchronous communication model that is also reflected by the application design itself. All of our implementations provide an asynchronous interface for applications, even the multi-threaded Java implementation.

The precondition for making this work is the existence of standardized interaction schemes, such as Mbus RPCs, properties and event notifications. Another important aspect is a sufficient support by Mbus implementations for a structured, callback-based programming model. For example, asynchronous delivery of RPCs (or RPC results) is only accomplishable in larger scale if the application programmer does not have to analyze RPC command names herself and test for the validity of argument lists manually. The description of our gateway applications in Section 10.3 shows how we have connected the Mbus C++ implementation to an application's state machine that takes Mbus events such as incoming RPCs and event notifications as input for state changes. I.e., we show how to build a complex application with a very formalized approach, relying on asynchronous Mbus communication.

Concerning strategies for designing decomposed Mbus-based applications with respect to re-usability and efficiency, we have learned that the approach to decouple an application's core functionality from it user interface by implementing them as separate Mbus entities is advantageous in several ways:

- Separating the user interface from the actual functionality is a useful design principle for developing robust applications because it requires the design of well-defined interfaces and thus leads to a modular overall design.

- It enhances the portability of applications because the often platform specific user interface functions can be adapted or newly implemented without the need to change the rest of the application. Thus, the separation allows for choosing the right tool for the right job, e.g., C++ for computationally complex calculations and a scripting language with a GUI toolkit binding for developing the user interface.

- Another advantage that should not be under-estimated is the possibility to increase the application components' concurrency by increasing the number of processes, especially given the asynchronous Mbus communication model. It is not only possible to increase the concurrency by distributing an application onto several execution threads, i.e., multiple Mbus processes, on the same CPU. Instead the application components, i.e., a user interface and the core module, could as well reside on different hosts.

We will substantiate these observations in the following sections by discussing Mbus protocol tests and simulations in Chapter 8, *Evaluation* and by providing details on the design of actual Mbus-based applications in Chapter 9, *Mbus in Conferencing Systems* and in Chapter 10, *Mbus in Projects*.

Chapter 8

Evaluation

Having described the Mbus transport and higher layer communication mechanisms in Chapter 6, *The Mbus Framework* and having provided an overview of implementation design considerations and specific implementations in Chapter 7, *Mbus Implementations*, we try to validate our design and the fundamental thesis of this document in the following sections: the objective is to demonstrate that the Mbus protocol is actually usable for its intended application domain, i.e., for the coordination of locally distributed application components. We demonstrate this by quantifying the performance of Mbus transport mechanisms in some selected test scenarios. Besides showing the general feasibility of the Mbus concept, we also try to gain some insights of the effects that can be achieved by different optimizations and configuration options.

The objective of these tests is not to compare the performance of Mbus protocol implementations to mechanisms such as ONC RPC. It is obvious that the Mbus protocol, due to its text representation, cannot compete with these specialized protocols that are aimed at efficient point-to-point communication, relying on static interfaces definitions, binary representations and other mechanisms. As motivated earlier, we have deliberately followed a more flexible approach that supports both point-to-point and group communication and does not necessarily require static interface descriptions. Furthermore, the objective is also not to compare the performance of different Mbus protocol implementations. Instead we will compare different techniques for realizing certain interaction schemes in order to learn the performance implications and possibilities for further optimization.

In Section 8.1, we present the result of some roundtrip-time measurements in different configurations, e.g., results of investigating the costs of security mechanisms such as message authentication. The objective of these tests is to quantify the performance of *fundamental* Mbus protocol mechanisms and to validate the concept of a secure, message-oriented coordination protocol. In Section 8.2, we investigate the performance of Mbus RPC communication in different configurations. This investigation has two objectives: we want to compare the performance of RPC communication with the performance of basic Mbus communication (as measured in Section 8.1), and we want to demonstrate the effect of different optimization techniques such as the unicast optimization and acknowledgment piggybacking. In Section 8.3, we investigate the performance of the Mbus fault tolerance mechanisms by simulating packet loss and measuring roundtrip-delays and the number of undelivered messages at different loss rates. The objective of this simulation is to demonstrate the effect of the Mbus fault tolerance mechanisms and to gain some insights about maximum loss rates that can be tolerated. We summarize our observations in Section 8.4.

8.1 Basic Message Transport and Security Mechanisms

Table 8.1 and Table 8.2 present the results of some Mbus message roundtrip-time measurements we have conducted between two entities. We have written two short test Mbus applications that locate each other on an Mbus and start to exchange a specified number of messages. The message exchange is *synchronous*, i.e., the sending application sends a message, waits for a reply and sends the next message.

The Mbus applications have been developed with the C++ implementation, since this implementation provides the highest degree of configurability of transport modes. We have run the tests multiple times, with different security configurations. We have also run tests where both entities resided on a single host (exchanging messages through host-local communication) and tests where both entities resides on two different hosts in a Fast Ethernet link (connected by a switch). In addition, we have once sent all messages via IP multicast (the basic Mbus transport mode) and once via IP unicast (relying on the unicast optimization that we have described in Section 6.2.1.4). For all tests, we have sent 50000 messages and then calculated the average roundtrip-time per message.[1] Both hosts provided identical hardware and software configurations: 1 Ghz AMD Athlon CPU, 0.5 GB RAM, running Linux-2.4.18.

Table 8.1: Roundtrip-times (1)

	HOSTLOCAL Unicast	**LINKLOCAL Unicast**
no digest, no encryption	1.91 ms	1.61 ms
SHA1 digests, no encryption	2.11 ms	1.60 ms
MD5 digests, no encryption	2.13 ms	1.63 ms
SHA1 digests, DES encryption	2.41 ms	1.85 ms

Table 8.2: Roundtrip-times (2)

	HOSTLOCAL Multicast	**LINKLOCAL Multicast**
no digest, no encryption	10.12 ms	6.65 ms
SHA1 digests, no encryption	10.72 ms	6.64 ms
MD5 digests, no encryption	10.80 ms	6.66 ms
SHA1 digests, DES encryption	10.97 ms	6.93 ms

The results are quite interesting:

- The first observation is that, in general, the roundtrip-times are quite reasonable for the Mbus application area: the development of locally distributed applications, e.g., separating an application's core functionality from the user interface. Even the worst case roundtrip-time of about 11 milliseconds will probably not play a significant role and will

[1] Because we have taken the total time for 50000 messages, we cannot determine the variance, our maximum and minimum values.

not cause a noticeable degradation of the application's overall performance. For certain (realistic) configurations we have achieved roundtrip-times of about 2 milliseconds which means that we can assume a single message to be transmitted and delivered in 1 millisecond.

It should be noted that the used implementation is by no means optimized, e.g., the decryption and decoding step involves unnecessary copying operations, which should be optimized for a production release. In addition, the generated object code contains symbol information for debugging, i.e., is not *stripped*, which typically impedes some of the compiler optimization to be effective.

- When we compare the different roundtrip-times with respect to the configuration of the security parameters, we can observe the obvious fact that no security is more efficient than optimal security — just as one would expect.

  However, a closer look reveals that the usage of message authentication only increases the roundtrip-time by *at most* 11% (MD5 for host-local unicast), whereas for other configuration the increase is not measurable at all. As a result, we can state that message authentication algorithms are that efficient (on modern CPUs) that there is no point to abandon message authentication (for performance reasons). Our measurements also assert the slightly higher efficiency of the SHA1 hashing algorithm compared to MD5.

  Enabling DES encryption results in an additional performance decrease of approximately 14%, which we consider tolerable for most applications. In any case, we can state that enabling security does not increase the roundtrip-delay by an order of magnitude or by a factor of 2 but merely by 26% (comparing host-local unicast without authentication and encryption to host-local unicast with SHA1 and DES).

- Comparing the results with respect to the *distribution* of entities reveals the interesting insight that the roundtrip-time for messages exchanged between *different* hosts is actually *lower* than the roundtrip-time for a host-local message exchange. For the host-local measurements, we have run two independent Mbus processes, i.e., with independent sockets, communicating over the operating system's loopback interface.

  For all test runs, the roundtrip-time for host-local communication was always at least 15% percent higher as observed for the distributed setup. We ascribe this to the overhead caused by frequent context switching that is required in the local scenario. Both processes compete for the CPU and have continuously to be moved from the `wait` state to the `run` state, which causes cache invalidation and register swapping.

  For two entities that reside on two different hosts (with largely un-loaded CPUs), the communicating entities can more or less stay on the CPU which results in a better performance.

  The main conclusion that we can draw here is that (given the switched Ethernet configuration that we have used for this test) it is definitely reasonable to decompose an application into locally distributed components. It should be noted that for these test runs, we have not even benefited from parallelism because the two entities communicated in a completely synchronous fashion.

- Finally, we notice a significant performance decrease when we employ IP multicast instead of IP unicast for Mbus communication. The Mbus C++ implementation allows for activating and de-activating the use of the Mbus unicast optimization. When de-activated, the roundtrip-time increases by an approximate average of 400%.

  We ascribe this to a significantly less efficient processing of multicast packets by the Linux kernel, however cannot provide a detailed discussion in this context.

  We can conclude that the *unicast optimization* (that has to be implemented in order to realize the more efficient unicast transport) is an important functionality of an Mbus implementation and should be enabled when applicable. For the C++ implementation, unicast transport is selected automatically for all messages that provide a sufficiently unique Mbus addresses.

Summarizing, we can state that these measurements have revealed some very interesting and partly unexpected insights: It is important to note that the Mbus security mechanisms do not impact the overall performance significantly. Hence, there is no strong argument to not rely on message authentication, which is also the reason why RFC 3259 requires all conforming implementations to provide at least the baseline (SHA1) authentication. We have also seen that the distribution of Mbus entities onto different hosts does not have to imply a performance decrease — quite the contrary: for some communication patterns it can actually result in a performance increase by avoiding overheads caused by context switches.

Finally, we can conclude from the significant performance differences for IP multicast and unicast communication that Mbus interactions that are inherently point-to-point such as RPC communication should at best be mapped to direct IP unicast communication. This requires an enhanced Mbus implementation that provides the Mbus unicast optimization as described in Section 6.2.1.4.

8.2 RPC Communication

In this section, we want to quantify the RPC communication performance with respect to different optimization configurations. We measure the RPC throughput between two entities, where one entity continuously sends RPCs to another entity. Both entities are synchronized, i.e., the sender waits for an RPC return command before sending a new RPC.

We measure the throughput by taking the time for a given number (50000) of RPC interactions and calculate the time per RPC for each test run. For all test runs, message authentication and encryption has been disabled. We have run the tests between two entities running as separate processes on the same host, using host-local multicast for multicast communication, and we have performed the same tests with two entities on different hosts, using link-local multicast for multicast communication.[2] The hardware and software configurations of the hosts are identical to the test setup in Section 8.1.

[2]Note that although RPC communication is by definition a point-to-point interaction (i.e., relying on Mbus unicast communication), the corresponding messages can nevertheless be transmitted by either IP unicast or IP multicast. Only implementations that provide the optional unicast optimization will be able to map an Mbus unicast message to an IP unicast transmission.

The first objective for this test is to quantify the effect of acknowledgment piggybacking. In theory, RPC communication should be an eligible interaction class for optimizing the sending of acknowledgments, as an RPC interaction comprises two messages, each of which is sent reliably and must thus be acknowledged. Without acknowledgment piggybacking, a receiver of an RPC command would first acknowledge the reception of the RPC command by sending a dedicated Mbus acknowledgment message, then perform the requested operation and finally send the RPC return command, again using reliable transport. This message has to be acknowledged by the RPC sender, before it will send the next RPC. Figure 8.1 depicts the message exchange for a non-optimized configuration, and Figure 8.2 depicts the message exchange with acknowledgment piggybacking. The reduction of the amount of messages is obvious.

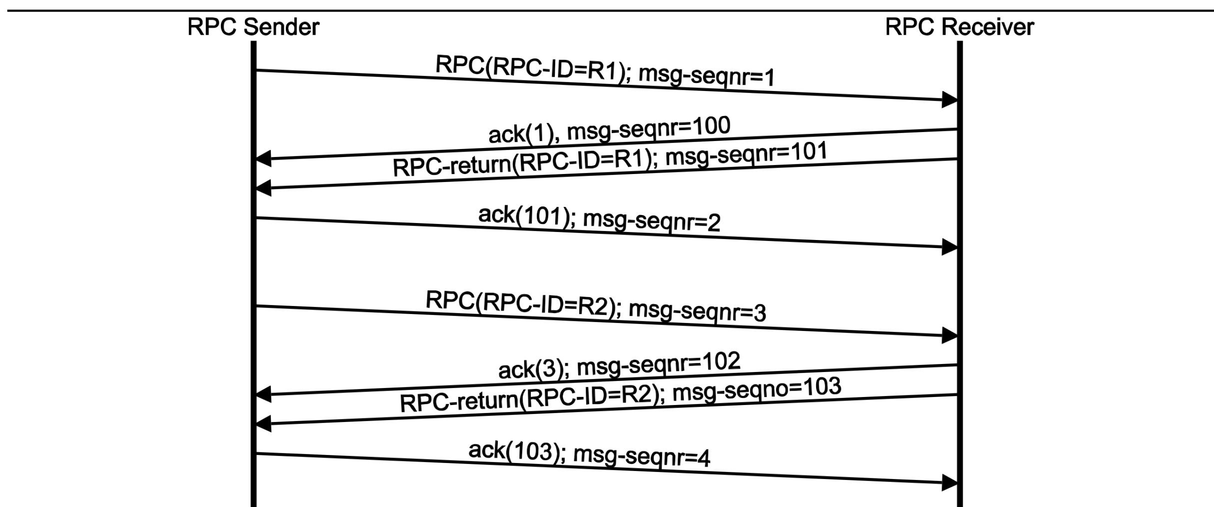

Figure 8.1: RPC interaction without acknowledgment piggybacking

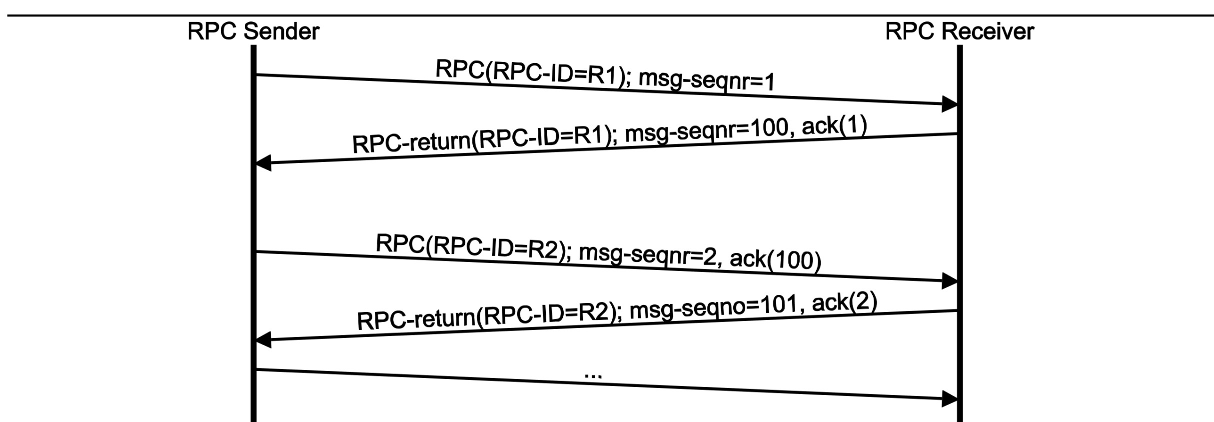

Figure 8.2: RPC Interaction with acknowledgment piggybacking

In a series of RPC interactions as performed in these test runs, enabling acknowledgment piggybacking reduces the number of messages that need to be exchanged by 50%, as each

message that has been sent via reliable transport will be acknowledged by the next regular message that is sent in the opposite direction.

The second objective for this test is to investigate the effect of the unicast optimization in combination with acknowledgment piggybacking. As RPC communication is per se a point-to-point interaction, direct unicast communication is applicable to all message exchanges within an RPC interaction. Therefore we have performed all test runs twice, once in multicast-only mode and once with enabled unicast optimization. The results are depicted in Table 8.3. From the results of Section 8.1 and the RPC discussion above, we would expect that unicast-optimization results in performance increases and that acknowledgment piggybacking can also enhance the efficiency of RPC interactions.

Table 8.3: RPC interaction times

	HOSTLOCAL	**LINKLOCAL**
multicast, no ack-piggybacking	12.97 ms	14.21 ms
multicast, with ack-piggybacking	12.30 ms	7.09 ms
unicast, no ack-piggybacking	9.78 ms	5.26 ms
unicast, with ack-piggybacking	6.68 ms	4.46 ms

In general, our initial expectations have been confirmed by this test: both acknowledgment piggybacking and unicast optimization incur performance increases in almost all configurations. However, a closer look at the results reveals some interesting insights:

- The positive effect of acknowledgment piggybacking is especially noticeable for the distributed setup, where both entities reside on two different hosts, in particular when all messages are transmitted via IP multicast. In these cases, acknowledgment piggybacking resulted in a performance increase of 100%.

  We explain this by the latency for processing and delivering multicast messages that is introduced by the hosts' operating systems, which is especially noticeable when the number of messages increases. With acknowledgment piggybacking enabled, the number of messages is reduced by 100%, which is reflected by the decrease in roundtrip-times.

- For the test runs between entities on one host, acknowledgment piggybacking is only effective when unicast communication is enabled as well. For the multicast-only case, the effect is hardly noticeable.

  This corresponds largely to our observations in Section 8.1, where we have noted that performance decreases significantly when multicast is used exclusively, almost regardless of other optimizations that are applied. For this test however, it is interesting to note that even a reduction of the number of messages by 100% does still not help to increase the performance significantly, whereas in the distributed case, we have noticed a significant performance increase.

- The combination of unicast optimization and acknowledgment piggybacking is always the most efficient configuration. For the host-local case, it incurs a performance benefit of almost 100%, and for the distributed cases it result in an performance increase of more than 200%.

- When we compare the figures for the unicast-enabled test runs, we can state that enabling acknowledgment piggybacking results in a rather small performance increase, compared to the distributed multicast-only setup. Although the number of exchanged messages is reduced by 100% we can only observe performance increases of 46% (host-local setup) and 18% (distributed setup).

  We ascribe this modest improvement to the additional processing and management overhead that is required to implement acknowledgment piggybacking. An entity that receives a reliable message must add this message to a per-participant list of yet to be acknowledged messages and must start a timer to eventually send this acknowledgment in a dedicated message.

  In summary, we can state that the gained performance increases are worth this effort. We expect the effect to be especially noticeable for interactive applications, e.g., applications with a GUI, because the elimination of dedicated acknowledgment messages results in less frequent process activations and context switches.

- When we compare the optimal results of this RPC test with the optimal results of the basic message roundtrip-time test in Section 8.1, we can observe a significant higher roundtrip-delay for RPC messages. We ascribe this to two factors:

  1. The exchanged messages for the RPC tests are significantly more complex than the basic messages of Section 8.1 that provide no parameters at all. As described in Section 6.3.2.2, RPC commands always provide a nested list of management information, e.g., for type specification and RPC identification purposes.

     This results in a slight overhead for all basic operations on messages such as copying and decoding. The C++ implementation decodes every Mbus parameter list element to an independent C++ object, and every list parameter is decoded to an STL container object.[3] While this facilitates the processing of messages for higher-layer operations and for application programmers, it incurs a certain processing cost that we have to take into account.
  2. The RPC protocol implementation represents an additional processing layer in the Mbus protocol stack that each RPC message must pass through. In particular, the RPC layer provides the registration of callback functions for incoming RPCs, which results in a lookup operation for every incoming message. For sending RPCs, the RPC layer starts a timer for verifying that the corresponding return command is received in a certain time frame. In addition, incoming return commands must be assigned to the corresponding RPC commands. Obviously these operations also contribute to the increased effort for handling RPC messages.

  It should be noted that these operations cannot be obsoleted by a different design; they are necessary for implementing the RPC paradigm of request/response communication. In other words, if these operations would be left to applications to implement the required service, this would not result in a performance increase.

[3] The *Standard Template Library* (STL) is part of the C++ standard library and provides generic container classes and algorithms that are based on the C++ template mechanism.

Summarizing, we can state that acknowledgment piggybacking and unicast transmission are useful optimizations of the basic Mbus transport mechanisms. For explicit point-to-point style communication such as RPC communication, the benefit is clearly visible, despite the increase of complexity for the implementation and the message processing.

High-volume multicast message transmission is dramatically less efficient than unicast communication, even for host-local communication. We cannot provide a detailed analysis of the delays in this document, but would recommend this as future work. However it should be noted that the way we have used the multicast transport in these tests is by no means typical for applications that employ Mbus for coordinating their application components. For measuring the roundtrip-delay we have continuously sent messages as fast as possible, i.e., generating significant load both for the user processes and for the operating system.

Finally, we have gained an approximate figure for the performance decrease that is caused by higher layer communication, i.e., RPC communication in this case. We can expect the roundtrip-time for a single RPC interaction to be about twice as long as the roundtrip for a simple message exchange, depending on the complexity of the argument list. This seems reasonable, given the required extra processing that has to be performed for the management of RPC communication.

8.3 Fault Tolerance

Fault tolerance in Mbus applications is achieved by two protocol mechanisms: the retransmission mechanism for reliable messages and the soft-state mechanism for periodic transmission of ephemeral information, where the latter is rather a concept for the fault tolerant dissemination of information than a protocol mechanism in itself.

In this section, we will investigate the performance of the retransmission based reliable transport mode of the Mbus protocol. The fundamental idea is to simulate the behavior of two Mbus protocol instances under different packet loss conditions. The scenario is similar to that of Section 8.2: two entities exchange RPCs, and one entity is continuously sending RPC messages that are answered by the other entity. We used RPCs for this simulation because an RPC interaction comprises an exchange of two reliable messages, and because the RPC protocol implementation already provides error reporting mechanisms for failed RPC interactions — a mechanism we need to measure the performance and the error rate.

This simulation has been performed with similar test programs as for the test in Section 8.2, however we have augmented the *Mbus Link Layer* (see Section 7.3) by a mechanism to drop packets with a pre-configured probability for each packet, which is transparent to the upper layers of the Mbus protocol stack. I.e., for this simulation, we use regular Mbus programs that are run as separate processes, but we control the number of packets that are delivered to the Mbus transport layer in order to simulate packet loss.

The simulation has been set up as follows: We have used two entities running as separate processes on the same host; one entity acts as a sender and the other entity acts as a receiver of RPC requests. We have enabled acknowledgment piggybacking and unicast optimization, and we have disabled message authentication and message encryption. I.e., an RPC interaction comprises two Mbus messages, each of which piggybacks the acknowledgment of the previous message from the communication peer. We run the two programs and measure the average RPC roundtrip-time and the fraction of failed RPC interactions for a large number of RPC exchanges

(500). For each of these iterations we increase the packet loss probability by 1%, starting from 0% and stopping at 45%.

In general, the sending of an RPC works as follows: upon sending the initial RPC request message, the RPC protocol implementation starts a timer that is set to the maximum retransmission timeout for reliable messages as specified in Section 6.2.1.5.[4] When the RPC return message is received in this time frame, the timer is canceled and the return message is delivered to the calling application. If the return command is not received before the timeout, the RPC interaction is canceled, i.e., the management state in the protocol implementation is destroyed, and a failure condition is signaled to the application.

We expect that, when increasing the loss probability, we will eventually see an increasing fraction of failed RPCs. However, we expect that for lower loss rates, the Mbus retransmission mechanism will repair the losses and enable a successful RPC interaction despite the loss of a small amount of messages. In addition, we expect that the RPC roundtrip-time, i.e., the time between the initial sending of a request and the reception of the return message, will increase with the loss probability, as message retransmissions have to be taken into account.

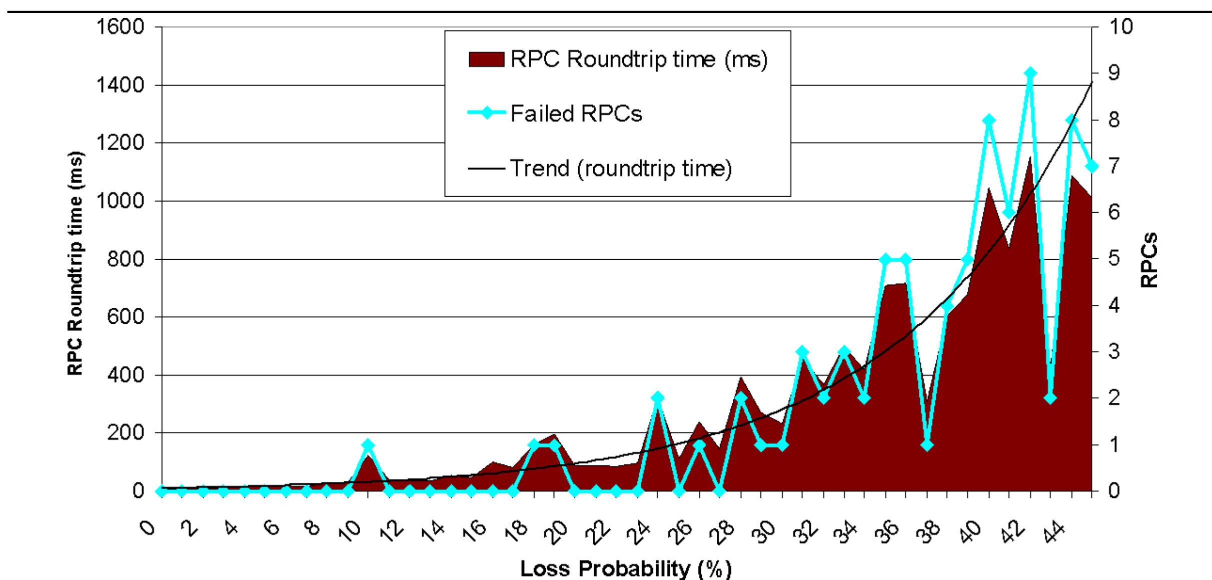

Figure 8.3: Effect of packet loss on RPC delivery performance

Figure 8.3 depicts the results of a test run. At large, the RPC failure rates and the roundtrip-times are both similarly correlated to the increase of the loss probability: as expected, both increase significantly as the loss probability goes up. Significant spikes in the graph of failed RPCs go together with spikes in the roundtrip-times graph. These spikes can be explained by a random concentration of message loss at a given point of time, which can for example lead to the failure of a complete RPC interaction. These spikes are correlated because a failed RPC interaction always causes the maximum delay, as the sender waits for retransmissions to

[4] The RPC layer does not deal with retransmissions on the Mbus transport layer, however, the RPC timeout must be large enough to accommodate the largest possible delay that may be caused by multiple retransmissions of the RPC message by the transport layer.

succeed.

Except for the spikes, the graphs are both roughly exponential. We see small increases for lower loss probabilities such as 10% and larger increases for ranges with higher loss probabilities such as 40%. Concerning RPC failures we can state that there are almost no failed RPCs for loss probabilities up to approximately 20%. Even for high loss probabilities of 45%, the average RPC failure rate stays below 10%.

The roundtrip-time reacts more directly to increased loss probabilities, which is not surprising, as every lost message increases the sum of the roundtrip-times. The increase in the loss probability range from 0% to 10% is still moderate. The roundtrip-times for loss probabilities up to 20% can already reach 100 milliseconds, which could be noticeable, especially for high-volume exchanges of RPC messages (which however are not the typical case). With loss probabilities higher than 20% the corresponding roundtrip-times can partly affect the performance of a distributed application significantly.

The reason why the roundtrip-times do not increase linearly with the loss probability is that the retransmission back-off increases linearly with each retransmission of a message. Each single Mbus message has a given loss probability. Let the loss probability be 20%. If an RPC request message (or the corresponding acknowledgment) is lost, this will cause one retransmission of the RPC message, after a time `T_r`. The new retransmission timeout is set to `2*T_r`. There is again a 20% probability that this message will be lost. If this is the case, a new retransmission is sent with a timeout of `3*T_r`. It is obvious that increasing loss probabilities result in increased probabilities for multiple retransmissions of a single message, which takes more time as the retransmission timeouts increase linearly and sum up. For example, a message that has to be retransmitted once, takes a time of `T_r` (for waiting for the acknowledgment timeout), whereas a message that is retransmitted twice takes a time of `3*T_r`.

In summary we can state the Mbus retransmission mechanism is able to cope quite well with packet loss probabilities of up to 20%. Although the roundtrip-time increases significantly with higher loss rates, the probability of RPC interactions that fail completely is fairly low.

It should be noted that a loss probability of 20% or even 10% is actually comparatively high and would be an indication for severe problems. On the other hand, we have simulated the message loss in a strictly synchronous scenario, where buffer overflows are essentially not possible due to the implicit flow control. In less synchronous Mbus communication scenarios, message loss can be caused by buffer overflows due to different sending and message processing rates, hence, it is important to see that the Mbus mechanisms for reliable transport is able to cope with these higher loss rates.

During our simulations we have noticed another interesting event that occurred in higher loss probability scenarios. Obviously the message loss does not only affect RPC messages but also the periodic `mbus.hello` messages that are exchanged to indicate entity aliveness. When a series of `mbus.hello` messages from entity `A` are lost at entity `B`, entity `B` will eventually remove `A` from its list of known entities. Now, this has consequences for messages that `B` wanted to send to `A`. Reliable message transport is only defined for point-to-point messages, hence an implementation will typically check the existence of a corresponding entity for any reliable message that is destined to a specific Mbus address. If the protocol implementation has removed an entity `A` from is list of currently known entities it will therefore refuse to send a reliable message to `A` and will notify the application program of the error condition.

Typically, one of `A's` next `mbus.hello` messages will be received by `B` and `B` will adopt

A for the list of known entities again. In rare scenarios with *very* high loss rates, the detecting and loosing of Mbus entities can even oscillate, however we did not observe this with loss probabilities less than 40%.

8.4 Summary

In this chapter, we have analyzed the behavior of the Mbus protocol in selected scenarios in order to quantify the effects of the corresponding protocol mechanisms. This analysis has provided some interesting insights. The main result of Section 8.1 is a quantification of the basic message transport performance. For example, with our given hardware and software configuration, we have been able to achieve roundtrip-times of 1.60 milliseconds, which is quite acceptable (for a non-optimized implementation).

A second very important observation is that security is not very expensive with respect to the overall roundtrip-times. Of course, the effective costs depend on the specific hardware and software platform that is deployed, but for the C++ implementation (that employs C implementations of the MD5, SHA1 and DES algorithms) we have seen that the activation of message authentication does not cause a significant overhead.

Both Section 8.1 and Section 8.2 have revealed that multicast-only communication is *significantly* slower than unicast communication. In Section 8.2, we have investigated the influence of different optimization mechanisms on the roundtrip-time for Mbus RPC interactions. RPC communication provides an point-to-point exchanges of two reliable messages, therefore we have investigated the effect of the Mbus unicast optimization (that can optionally be provided by implementations) and the effect of the also optional piggybacking of acknowledgments. The first observation was that RPC communication incurs a slightly higher degree of overhead due to increased message length and complexity and due to additional protocol management effort that has to be taken into account. The second observation was that the mentioned optimization are quite helpful to increase the overall performance of RPC interactions.

While acknowledgment piggybacking helps to reduce the absolute number of individual messages that have to be exchanged, the unicast optimization leverages the typically more efficient IP unicast transport mode for messages that are destined to a single entity anyway. It should be noted that unicast optimization has another obvious positive effect that we did not explicitly quantify in this chapter: In group communication scenarios, i.e., the typical case for Mbus communication, it enhances the performance of entities that are *not* part of on point-to-point message exchange, because the unicast transport delivers the message to the destination entity's UDP port only. Entities that are not addressed will not receive the message on their UDP interface and will not have to filter the message based on the Mbus destination address. This is especially useful in single-threaded applications where, without unicast optimization, uninvolved entities would be activated by their event notification mechanism in order to process a message that has be discarded anyway.

The analysis of the Mbus fault tolerance mechanisms, namely the retransmission-based reliable transport mode, has shown that the specified mechanism can cope quite well with loss rate of up to 20%. In a specific simulation scenario, we have seen that while the RPC roundtrip-time increases, the RPC failure rate remains at a very low level (1% in the worst observed case).

Chapter 9

Mbus in Conferencing Systems

The Mbus transport mechanisms and the Mbus Guidelines abstractions that we have described in Chapter 6, *The Mbus Framework*, provide an application independent framework for the development of locally distributed applications. Application specific commands and addressing schemes are specified in so-called *Mbus Application Profiles*.

The main application area that the Mbus framework has been intended for is *local coordination of conferencing endpoint components*. In this chapter, we describe our local Mbus-based coordination architecture for conferencing systems. Naturally, a local coordination framework for conferencing systems must also consider the general architecture of multimedia conferencing in the Internet, e.g., it must be aligned with existing wide-area conference control protocols. Hence, our work does not only include Mbus profiles for coordination of conferencing systems but also involves the definition of a system model and the development of an architecture and a specific description language for the local aggregation and wide-area negotiation of capabilities and conference configurations. Furthermore, we do not limit the scope of our work to local *endpoint* coordination but extend it to the coordination of server systems, such as SIP proxies and SIP-H.323 gateways.

The *application* of this architecture to specific projects in described in Chapter 10, *Mbus in Projects* — in this chapter, we focus on concepts and the description of a selected Mbus application profile. In Section 9.1, we present a fundamental model of *internal management* and discuss the required services for the internal management of conferencing endpoints and gateways. Section 9.2 covers the description of a specific management service: a framework for the description and negotiation of configurations for conferencing systems, and Section 9.3 discusses an Mbus application profile for the coordination of call control engines — particular components in multimedia conferencing systems. We summarize the main aspects of our work in Section 9.4.

9.1 System Model

One conclusion of our discussion of the H.323 conference control architecture and the Internet Multimedia Conferencing architecture in Chapter 2, *Conferencing Architectures*, was that both architectures do not address local coordination, e.g., the enforcement of conference control semantics on a local endpoint. However, in both architectures there is a notion of different application entities that participate in individual application protocol sessions in order to realize

the overall conferencing application service.

In this section, we will present a system model for conferencing endpoints that we have developed for our Mbus-based conferencing systems. This model defines the relationship of local Mbus coordination to wide-area conference control and defines different aspects of local coordination that we have conceived. For example, we distinguish between Mbus-based control of call signaling engines and the coordination of media engines.

9.1.1 The Internal Management Concept

In [Ott97], Ott has categorized the services of so-called *Internal Management Services* of desktop multimedia conferencing systems. In this context, Ott has described a local system model that provides multiple application entities and a coordinating instance that orchestrates these entities.

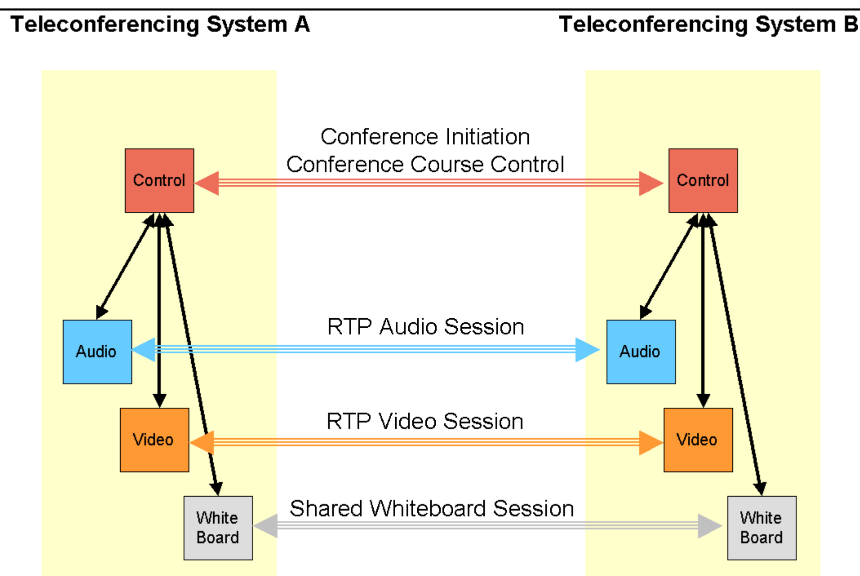

Figure 9.1: Internal management and wide-area communication

In this fundamental model that is depicted in Figure 9.1, a controller entity of a conferencing endpoint is engaged in wide-area conference initiation and conference course protocol sessions and implements corresponding control semantics, e.g., control messages for floor control, by controlling the local application entities through the internal management system. Each of the local application entities is a member of an application protocol session, such as an RTP session for audio transport, and is managed by the controller. For example, the controller could be a SIP implementation that initiates and manages SIP call relationships.[1] When a new conference has been established, the local applications need to be parameterized with parameters for their application sessions that have been negotiated in the SIP call setup phase.

[1]Ott's model, although in principle designed as a general architecture, has originally been applied to H.323 conferencing systems. The fundamental concepts can be transfered to other protocols as well.

Ott has classified the internal management functions as *interoperability services* and *coordination services*. Interoperability services enable a set of conferencing systems to automatically determine possible ways of interoperation by a procedure called *capability exchange*, where coordination services are intended to help maintaining consistency across all application entities on a single endpoint, e.g., in order to implement a conference policy.

9.1.2 Mbus Coordination Model

In Section 1.3.1 and Section 3.1.1, we have described the concept of a decomposed conferencing endpoint and discussed the requirements for a local coordination mechanism. In this Mbus based local coordination model, we generalize the aforementioned internal management architecture by modeling the entities of a local conferencing system as a federation of components that may establish different communication relationships between each other. For example, we do not impose a strict point-to-point communication model, where all application entities are only engaging in a control relationship with the local controller. Instead we generally assume group communication between all entities, whereas some entities may establish dedicated control relationships to other entities, e.g., in order to enforce the implementation of conference control.

Furthermore, we consider a higher degree of granularity of components and extend the set of communicating entities to user interface components, call signaling protocol engines and potentially even to sub-systems of application entities, such as an application's RTP and rendering component.

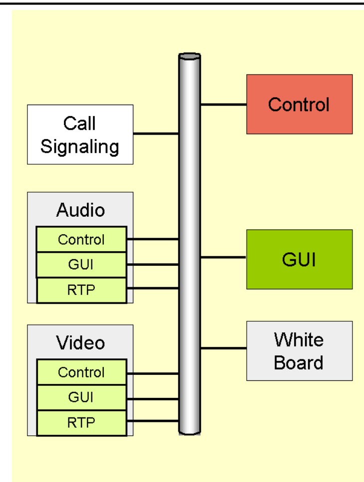

Figure 9.2: Mbus coordination model

Figure 9.2 depicts this model schematically: a set of endpoint components attaches to the Mbus as a local rendezvous and group communication system. The video and audio components exhibit a sub-structure of components, each of which also attaches to the Mbus. The call signaling and call control function is not provided by the controlling entity itself but is implemented as a separate module. The motivation for this is two-fold:

1. In Section 3.1.1, we have analyzed the requirements and typical implementation strategies for multimedia conferencing systems, especially component-based IP telephony endpoints. One observation in this context was that many implementations already follow component-based approaches, e.g., by employing a component supplier's call-signaling engine. By separating the endpoint control from the call-signaling functionality, we enable the re-use of these components in different application contexts.

2. From a service perspective, call signaling and call control protocols such as SIP and H.225.0/H.245 provide many similarities and largely offer the same services: they provide an interface to user location services, they provide the concept of negotiating and communicating conference configurations and they provide mechanisms for terminating conferences (amongst other services).

   Given this similarity of services it becomes possible to define a generic interface to call signaling and call control functions that provides a common set of features and can be used to control specific components regardless of the concrete call control protocol. We present a corresponding approach in Section 9.3.

   It should be noted that, in our model, we combine call signaling and call control into one component. Our discussion of H.225.0, H.245 and SIP in Chapter 2, *Conferencing Architectures*, has indicated that these functions are closely related and, in case of H.323 *Fast Start*, are intertwined anyway. Therefore, an Mbus call control entities does provide both call signaling and call control services.

   As we have described in Chapter 2, *Conferencing Architectures*, H.323 and SIP based conference systems provide additional similarities, e.g., both generally rely on RTP as a real-time transport protocol for multimedia applications. Deploying call control components with a generic service interface enables the development of *multi-protocol* endpoints, i.e., endpoints that can interoperate with both SIP and H.323 endpoints.

Within such an endpoint architecture, there can be different communication relationships between the components, not all of them exhibit strict point-to-point control characteristics:

- media engines can coordinate themselves by exchanging periodic receiver and renderer statistics, e.g., in order to provide lip synchronization as we have described in Section 4.2.1;
- media engines can report events such as the beginning of talk spurts to user interface components; and
- call signaling engines can multicast certain events such as `incoming-call` to a group of entities, e.g., a user interface could notify the user appropriately and an endpoint controller could decide whether to accept the call and take over the control of the signaling engine.

In Section 3.1.1 we have analyzed the different functions of an endpoint's different components. As depicted by Figure 9.2, we distinguish the following classes of components:

- call signaling and call control components;

- application components (audio, video, whiteboard);
- user interfaces (GUI); and
- controllers.

For the Mbus-based coordination, each of these components implements a set of Mbus commands and conforms to one (or more) Mbus application profiles. For example, the application entities support fundamental, application-independent Mbus commands for querying capabilities and setting configurations. The RTP-based application entities (the audio and video components) additionally support commands for configuring RTP sessions and for distributing RTP session events within the Mbus session. The call signaling and call control engine supports a dedicated call control profile, which allows a controller to coordinate the establishment and termination of call control relationships. Similar to an RTP profile for application entities, the call control profile can largely be independent of specific call signaling/control protocols, but can generalize their functions and provide rather abstract functions, such as `setup call` and `terminate call`.

9.1.3 Local Coordination in End Systems

In the following, we describe the different phases of the operation of a conferencing endpoint with respect to local coordination. In general, we distinguish between the following different phases of a conferencing system's operation:

System start and initialization: When an Mbus based conferencing system is started, a *minimal set of Mbus entities* is brought into a common session. For this discussion, we assume that each entity provides the required Mbus configuration parameters. The minimal set of entities comprises a controller, the user interface component, a call signaling engine and essential application components such as audio and video components.

The Mbus awareness mechanism allows entities to locate each other, and each entity will build a list of available components in the system.

After this initial rendezvous phase, the system enters an *initial configuration* phase. In this phase, the user interface is generated (e.g., by creating an initial window in case of a GUI), and application entities are configured with respect to their user interface characteristics. In addition, the controller determines the capabilities of the available application entities by obtaining a *capability description* of each entity. The capability descriptions are gathered, augmented by user preferences and transfered to a comprehensive capability and preference description of the whole conferencing system.

This capability description is one parameter for the initial configuration of the signaling engine. Other parameters are the user's name, her call signaling protocol specific URI, and potentially other call signaling specific parameters such as registrar and proxy addresses.

Conference establishment: There are different variants of how a conference can be established, depending on the type of a conference (e.g., a voice call or a scheduled meeting

that is announced publically). For this discussion, we assume that the user initiates a conferencing by inviting another user, employing a call signaling protocol such as SIP.

In this case, the user will initiate the conference establishment through her user interface component by specifying the desired participants and by providing a conference configuration based on the user's preferences. This configuration defines the applications to be used in the conference and can additionally specify parameters for applications, such as the desired quality of a video codec.

The user interface component will forward these parameters to the controller that is going to orchestrate the different application components. The controller configures the involved application entities and also requests parameters for their respective application sessions, e.g., transport parameters such as IP addresses and UDP port numbers. These parameters augment the capability description that is passed to the signaling protocol entity for establishing a signaling control relationship to the invitee's endpoint. The signaling protocol engine, e.g., a SIP implementation, uses the available conference and capability description to negotiate a set of commonly used parameters with the other endpoint. The goal of this negotiation is to determine a set of commonly supported application types and suitable configurations for these applications that allow both endpoints to interoperate.

If this negotiation is successful, the call setup will also provide both endpoints with a complete description of all required transport and application parameters that are needed to instantiate application sessions. The call signaling engine forwards these parameters to the controller, and the controller configures the application entities accordingly and initiates the application sessions. In addition, the controller also notifies the user. The conference is now established.

It should be noted that we are not considering failure conditions at the moment — obviously call setup processes can also fail, e.g., when the called party refuses to accept the call.

Conference operation: During the conference, different forms of coordination are performed. For example, we have already mentioned the internal synchronization of media engines. In addition, there is user initiated control of the system through the user interface, e.g., the user might press the "mute" button. The action has to be forwarded to the application entities, e.g., in order to suspend the sending of audio data.

In case conference course control is employed, conference control actions have to be implemented by the controller — also by coordinating the application entities. For example, when the controller obtains the *floor token* for the endpoint, it has to coordinate application entities to start sending.

Conference termination: When the conference has been terminated, i.e., the local user has requested the termination or the call signaling engine has received a conference termination request over its call signaling protocol, the controller has to notify the involved application engines and the user interface. The application entities will terminate their application sessions, and the user interface will reflect the new state at its display.

9.1.4 Local Coordination in Call-Signaling Gateways

In addition to coordination services for conferencing endpoints, we have also defined coordination services for decomposed signaling gateways, i.e., gateways that implement two or more signaling protocols and provide the service to relay calls from one call signaling protocol domain to another, e.g., in order to allow an H.323 user to call a SIP user.

Figure 9.3 depicts a schematic overview of a corresponding scenario (in Section 10.3 we present real, more feature-rich implementations). The two conferencing endpoints on the left and right hand side of the figure provide different call signaling protocol implementations and are thus not able to interoperate. However, as we have noted in Chapter 2, *Conferencing Architectures*, the call signaling and call control protocols provide largely similar concepts and can thus be mapped to each other. This mapping can be provided by a dedicated entity that translates the individual call signaling messages and manages calls between different protocol domains: a *call signaling gateway*.

In this simplified figure, the call signaling gateway provides two call signaling protocol implementations, a SIP implementation and an H.323 (H.225/H.245) implementation. As the diagram suggests, the gateway system can itself be designed as a component-based system and be implemented as an Mbus application. In this example, there is one controller that coordinates the two call signaling engines. Note that this gateway is a *signaling gateway* only, i.e., it does not relay media streams. Instead it is intended for scenarios where the application entities, i.e., the media engines, of the different endpoint systems are able to interoperate directly (which is often the case for RTP based audio and video applications).

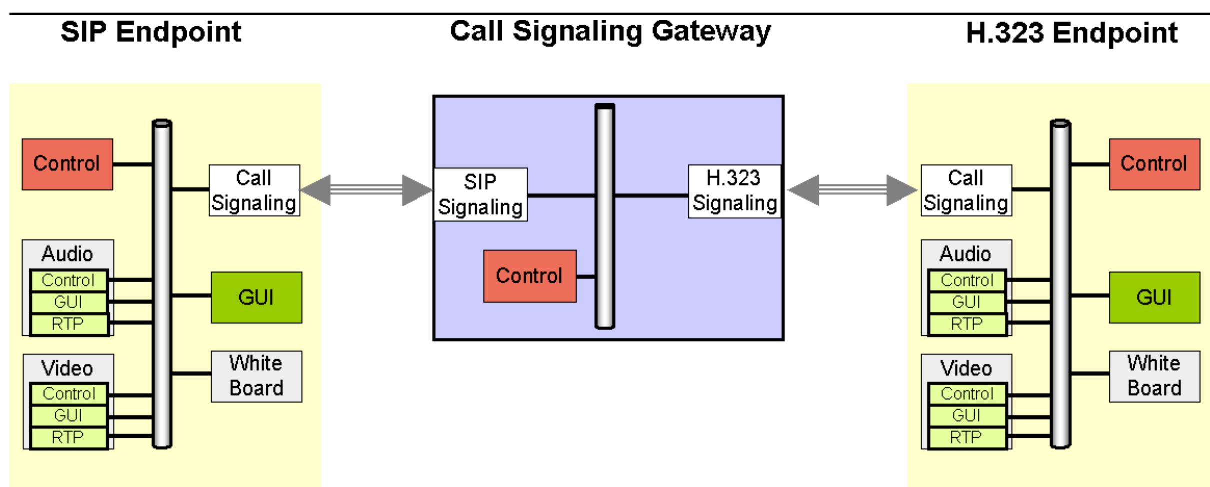

Figure 9.3: Signaling gateway

The following phases of operation are required in order to establish a call through such a gateway system (we assume that the SIP user calls the H.323 user):

System start and initialization: Similar to the endpoint scenario, the gateway system begins its operation by starting its component entities that attach to the Mbus and locate each other. In this case, the controller waits until both signaling protocol engines are available and establishes control relationships to them. The signaling protocol engines do not need to communicate with each other.

The controller initializes the signaling protocol engines, e.g., by conveying configuration parameters. Note that there are no application capabilities as such to be assessed and aggregated since the gateway itself does not provide applications itself. The gatewaying operation will be instantiated on a per call basis, i.e., taking the per-call configuration of the involved endpoints into account.

Conference establishment: When the SIP user wants to call the H.323 user, her SIP engine will not contact the H.323 user's endpoint directly but will contact the signaling gateway through its SIP engine. In this case, the gateway's SIP engine acts as callee's SIP engine that is invited to the call.

The gateway's SIP engine notifies the controller of the incoming call event and (if applicable) provides the controller with the received capability and configuration description that has been sent by the SIP endpoint in its call setup request, i.e., in a SIP `INVITE` request.

If the controller decides that it can relay the call to its H.323 engine, it requests the H.323 engine to setup a call to the H.323 user that is the actual desired participant. For this purpose, the controller forwards the capability and configuration description to the H.323 engine. The H.323 engine tries to initiate a call to the H.323 endpoint and offers the configuration description it has received from the controller. If this setup step is successful, the gateway's H.323 engine will eventually obtain a configuration description from the H.323 endpoint and report this to the controller.

The controller will indicate the progress of the call establishment to its SIP engine. The SIP engine will in turn continue to establish the call with the caller's SIP engine by sending it the configuration description from the H.323 endpoint. The conference is established.

Note that there are often progress indications and call setup acknowledgments that we have not explicitly mentioned here. Section 9.3 considers the requirements for concrete call signaling protocols.

Conference operation: During the conference, the gateway does not have to perform any specific actions, except for maintaining the call state (and possibly forwarding and translating call signaling messages during a call).

Conference termination: When a user decides to terminate the conference and the endpoint's call signaling engine sends a corresponding request to the gateway, the controller forwards the request to its corresponding other signaling engine which sends the termination request via its native call signaling protocol. The corresponding acknowledgment is again forwarded and translated in the opposite direction and eventually the conference has been terminated.

9.1.5 Summary

We have presented a model for the coordination of components in decomposed conferencing end systems and gateways and have analyzed the different phases of operation for these systems.

Based on Ott's *internal management* concept, we have shown how the *interoperability services* and *coordination phases* are employed during the operation of conferencing systems.[2]

The objective in defining this model and the corresponding Mbus based protocol mechanisms is to enable a *component based* approach for the development of conferencing systems. The benefits of such an approach lie in the possibility to deploy third-party components in new applications, i.e., to increase the degree of re-use and to enhance the robustness and manageability of feature-rich applications. In particular, we try to *generalize* the interfaces and the corresponding coordination procedures of conferencing systems. The first step of this generalization is to classify different components by their functions, i.e., distinguishing *media engines*, *controllers* and *signaling protocol engines*.

One goal of this systematization is to provide the possibility for conceiving conferencing systems as *general frameworks* that can be specialized according to specific application requirements by adding certain components. For example, the general framework of a conferencing endpoint could consist of a generic controller and a user interface. With our component based approach and well-defined interfaces, we can compose a concrete application by adding a specific call signaling protocol engine, e.g., a SIP engine, and different application entities, e.g., an audio engine and a video engine. The general framework, i.e., the controller and the user interface, do not necessarily have to be manually adapted in order to accommodate these entities. Instead the idea is for a controller to dynamically learn of the existence of a call signaling engine and available application entities, to learn their capabilities as described above and to integrate them into the overall application context. In the previous sections we have noted some key elements that are required for enabling this approach:

- a general capability and configuration description mechanism that allows application entities to describe their parameters, thus enabling controllers to aggregate them without application specific knowledge; and

- well-defined interfaces for components of a certain type that allow for utilizing the component without necessarily having to know its exact type.

In the following, we describe our approach for a suitable capability and configuration description framework in Section 9.2. We have seen that call control coordination is a function which is required for both endpoint and gateway coordination. Therefore we will have a closer look at the design and the individual Mbus commands of an Mbus Call Control Profile in Section 9.3. These two developments are the basis for implementing our concept of generalized, component based conferencing systems. In Section 10.1.2 we have additionally provided a description of an Mbus-based application entity and its Mbus interface.

[2] It should be noted that the order of capability exchange and coordination is actually not fixed. For example, relying on the Mbus as a coordination mechanism, it is conceivable to add an application entity *during* a conference, learn its capabilities and integrate it into the conference through conference control mechanisms. Although we did not discuss these dynamic changes of conference configurations in order to simplify the presentation we do not want to exclude it.

9.2 Session Description

Our discussion of the Internet Multimedia Conferencing Architecture in Section 2.2 has shown that the protocols for conferencing have evolved around the loosely coupled model, intended to accommodate large-scale conferences on the Internet that can be publically announced by multicasting the conference description through the use of SAP, the Session Announcement Protocol. This is also reflected by the design of corresponding technologies for session description: SDP, the Session Description Protocol, has been designed for the description of multicast conferences that are publically announced and can be joined by interested participants that have received the description, typically simply by joining the conference's application sessions. Because of this simple model (and in order to accommodate bandwidth constraints for global-scope multicast announcements) SDP is a very basic description format that is essentially targeted at providing potential participants with the minimal configuration information that is required to join a conference.

The evolution of conferencing technologies and the development of SIP, the Session Initiation Protocol, into a call signaling protocol that addresses the requirements of *Internet telephony* applications has introduced some elements of *tightly coupled conferencing* into the Internet Multimedia Conferencing Architecture. For example, when initiating an Internet telephony call, users expect endpoints to interoperate just as they would for traditional telephony. In order to interoperate, heterogeneous endpoints have to agree on a set of applications for a conference, negotiate suitable configuration parameters for applications and exchange transport parameters for establishing application sessions. All of this has to be done at the call setup phase, i.e., it is a dynamic function that cannot be performed in advance.

Because these requirements differ fundamentally from the original model of multicasting fixed conference descriptions by the use of SDP, dedicated procedures have been defined that specify how SDP is to be used with SIP in order to achieve a subset of the mentioned interoperability services. RFC 3264 (*An Offer/Answer Model with the Session Description Model (SDP)*, [RFC3264]) specifies a procedure by which two end systems can *exchange* SDP descriptions in order to gain a common understanding of a conference configuration. The procedure relies on one party sending an SDP offer containing a description based on its own capabilities, preferences and transport configuration parameters. The other party returns an SDP answer that contains an updated session description including its own transport parameters. As depicted by Figure 9.4, the fundamental offer/answer model works as follows:

1. The offerer (`A` in this example) sends a description of the intended applications (audio and video), configuration parameters such as codec types for these applications and transport parameters such as port numbers for the application sessions. The general model is that each party describes what it is willing and capable to *receive*. Hence, the transport parameters also specify where `A` wants to receive data for the respective application sessions.

2. The answerer (`B`) returns a subset of `A's` proposed configuration. For example in this case, `B` does not support video (which is indicated by setting the port number of the corresponding transport parameter configuration to zero). `B's` description provides transport parameters that specify where `B` wants to receive data for application sessions, in this case for the audio session.

In this basic example, the two endpoints have negotiated the conference parameters on the

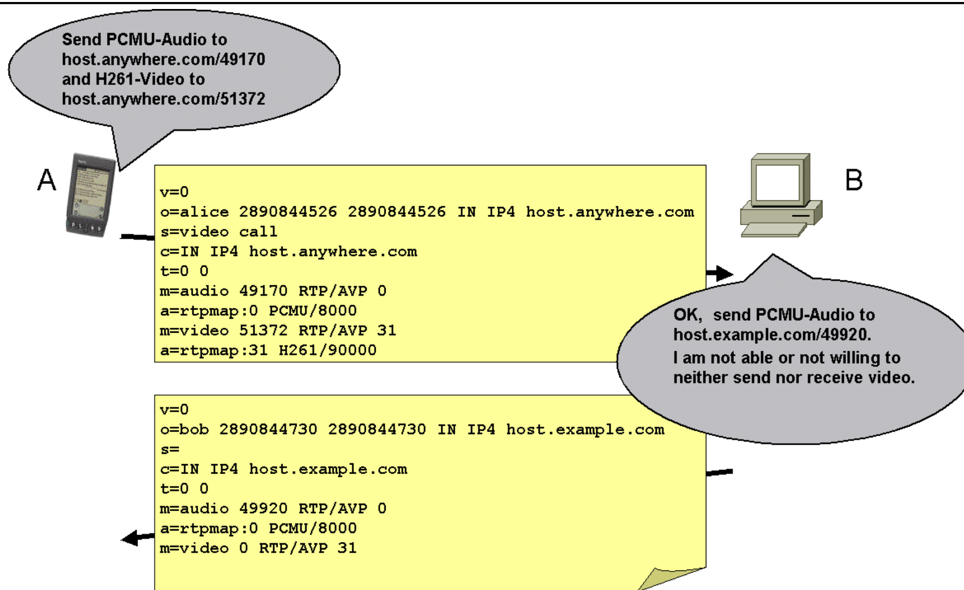

Figure 9.4: The SDP offer/answer model

basis of selecting application sessions, i.e., `A` has offered audio and video but `B` has signaled to support audio only. The SDP offer/answer model also allows to perform some sort of negotiation on the basis of RTP payload formats, where each party describes the payload formats it is willing to receive.

However, in general this model is rather limited. For example it is hardly possible to negotiate individual parameters for generating interoperable configurations for more complex applications such as advanced video codecs. This problem and a number of other issues that we have described in [Kutscher01c] in more detail have led us to develop an approach for describing and negotiating conference configurations: *SDPng*, a successor to SDP, is intended to as mechanism to implement the *interoperability service* in more tightly coupled conferencing scenarios and provide an extensible framework that is not fixed to specific applications. One of the key concepts is to describe conferences and endpoint capabilities in a way that allows conference controllers, gateways and other instances to process the resulting description without having to understand the semantics of parameters — a key concept for developing application independent, component based frameworks as described in Section 9.1.

In Section 9.2.1, we describe the underlying SDPng conference model, in Section 9.2.2 we provide a short overview of the language and its concepts and in Section 9.2.4 we summarize the main ideas. [Kutscher03a] provides the actual specification of SDPng.

9.2.1 SDPng System Model

Any (computer) system has, at a time, a number of rather fixed hardware as well as software resources. These resources ultimately define the limitations on what can be captured, displayed, rendered, replayed, etc. with this particular device. We term features enabled and restricted by these resources *system capabilities*.

Example: System capabilities may include: a limitation of the screen resolution for

true color by the graphics board; available audio hardware or software may offer only certain media encodings (e.g. G.711 and G.723.1 but not GSM); and CPU processing power and quality of implementation may constrain the possible video encoding algorithms.

In multiparty multimedia conferences, participants employ different *components* for multimedia *interaction* in conducting the conference. A *component* describes a particular type of interaction (e.g. audio conversation, slide presentation) that can be realized by means of different applications (possibly using different protocols).

Example: In lecture multicast conferences, one component might be the voice transmission for the lecturer, another the transmission of video pictures showing the lecturer and the third the transmission of presentation material.

Depending on system capabilities, user preferences and other technical and political constraints, different configurations can be chosen to accomplish the use of these components in a conference. Each component can be characterized at least by (a) its intended use (i.e. the function it shall provide) and (b) one or more possible ways to realize this function. Each way of realizing a particular function is referred to as a *configuration*.

Example: A conference component's intended use may be to make transparencies of a presentation visible to the audience on the Mbone. This can be achieved either by a video camera capturing the image and transmitting a video stream via some video tool or by loading a copy of the slides into a distributed electronic whiteboard. For each of these cases, additional parameters may exist, variations of which lead to additional configurations (see below).

Two configurations are considered different if they employ entirely different mechanisms and protocols (as in the previous example) but also if they largely choose the same mechanisms and differ only in a single parameter.

Example: In case of a video transmission, a JPEG-based still image protocol may be used, H.261 encoded CIF images could be sent, as could H.261 encoded QCIF images. All three cases constitute different configurations. Of course there are many more detailed protocol parameters.

Each component's configurations are limited by the participating system's capabilities. In addition, the intended use of a component may constrain the possible configurations further to a subset suitable for the particular component's purpose.

Example: In a system for highly interactive audio communication the component responsible for audio may decide not to use the available G.723.1 audio codec to avoid the additional latency but only use G.711. This would be reflected in this component by only showing configurations based upon G.711. Still, multiple configurations are possible, e.g. depending on the use of A-law or μ-law, packetization and redundancy parameters, etc.

A *configuration* is a set of parameters that are required to implement a certain variation (realization) of a certain component. There are actual and potential configurations.

- *Potential configurations* describe possible configurations that are supported by an end-system. They are a set of any number of configurations per component indicating the functional capabilities of the system as constrained by the intended use of the various components.
- An *actual configuration* is an *instantiation* of one of the potential configurations, i.e., a decision how to realize a certain component. For each instance of a component, there is at least one actual configuration reflecting the mode of operation of this component's particular instantiation.

In less abstract words, potential configurations describe what a system can do (*capabilities*) and actual configurations describe how a system is configured to operate at a certain point in time.

> Example: The potential configuration of the aforementioned video component may indicate the support for JPEG, H.261/CIF, and H.261/QCIF. A particular instantiation for a video conference may use the actual configuration of H.261/CIF for exchanging video streams.

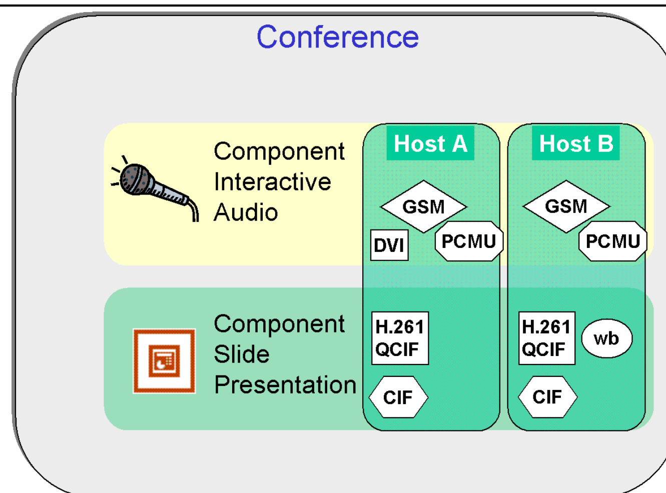

Figure 9.5: Conferencing system model

This model is depicted in Figure 9.5. To decide on a certain actual configuration, a negotiation process needs to take place between the involved peers:

1. to determine which potential configuration(s) they have in common; and
2. to select one of this shared set of common potential configurations to be used for information exchange (e.g. based upon preferences, external constraints, etc.).

Note that the meaning of the term *actual configuration* is highly application-specific. For example, for audio transport using RTP, an actual configuration is equivalent to a payload format (potentially plus format parameters), whereas for other applications it may be a MIME type.

In SAP session announcements on the Mbone, for which SDP was originally developed, the negotiation procedure is non-existent. Instead, the announcement contains the media stream description sent out (i.e. the actual configurations) which implicitly describe what a receiver must understand to participate.

In point-to-point scenarios, the negotiation procedure is typically carried out implicitly: each party informs the other about what it can receive and the respective sender chooses from this set a configuration that it can transmit. Capability negotiation must not only work for 2-party conferences but is also required for multi-party conferences. Especially for the latter case it is required that the process to determine the subset of allowable potential configurations is deterministic to reduce the number of required round trips before a session can be established. For instance, in order to be used with SIP, the capability negotiation is required to work with the offer/answer model that is used for session initiation with SIP — limiting the negotiation to exactly one round trip.

9.2.2 SDPng Design

SDPng is a description language for both potential configurations (i.e. capabilities) of participants in multimedia conferences and for actual configurations (i.e. final specifications of parameters). Capability negotiation is the process of generating a usable set of potential configurations and finally an actual configuration from a set of potential configurations provided by each potential participant in a multimedia conference.

SDPng supports the specification of endpoint capabilities and defines a negotiation process: In a negotiation process, capability descriptions are exchanged between participants. These descriptions are processed in a "collapsing" step, which results in a set of commonly supported potential configurations. In a second step, the final actual configuration is determined that is used for a conference. This section specifies the usage of SDPng for capability negotiation. It defines the collapsing algorithm and the procedures for exchanging SDPng documents in a negotiation phase.

The description language and the rules for the negotiation phase are (in general) independent of the means by which descriptions are conveyed during a negotiation phase (a reliable transport service with causal ordering is assumed). There are however properties and requirements of call signaling protocols that have been considered to allow for a seamless integration of the negotiation into the call setup process. For example, in order to be usable with SIP, it must be possible to negotiate the conference configuration within the two-way-handshake of the call setup phase. In order to use SDPng instead of SDP according to the offer/answer model it must be possible to determine an actual configuration in a single request/response cycle.

9.2.2.1 The SDPng Negotiation Process

Conceptually, the negotiation process comprises the following individual steps (considering two parties, `A` and `B`, where `A` tries to invite `B` to a conference).

1. `A` determines its potential configurations for the components that should be used in the conference (e.g. "interactive audio" and "shared whiteboard") and sends a corresponding

SDPng instance to B. This SDPng instance is denoted CAP(A).

2. B receives A's SDPng instance and analyzes the set of components of the description. For each component that B wishes to support it generates a list of potential configurations corresponding to B's capabilities, denoted CAP(B).

3. B applies the collapsing function and obtains a list of potential configurations that both A and B can support, denoted CAP(A)xCAP(B) = CAP(AB).

4. B sends CAP(B) to A.

5. A also applies the collapsing function and obtains CAP(AB). At this step, both A and B know the capabilities of each other and the potential configurations that both can support.

6. In order to derive an actual configuration from the potential configurations that have been obtained, both participants have to pick a subset of the potential configurations that should actually be used in the conference and generate the actual configuration. It should be noted that it depends on the specific application whether each component must be assigned exactly one actual configuration or whether it is allowed to list multiple actual configurations. In this model, we assume that A selects the actual configuration, denoted CFG(AB).

7. A augments CFG(AB) with the transport parameters it intends to use, e.g., on which endpoint addresses A wishes to receive data, obtaining CFG_T(A). A sends CFG_T(A) to B.

8. B receives CFG_T(A) and adds its own transport parameters, resulting in CFG_T(AB). CFG_T(AB) contains the selected actual configurations and the transport parameters of both A and B (plus any other SDPng data, e.g., meta-information on the conference). CFG_T(AB) is the complete conference description. Both A and B now have the following information:

   CAP(A): A's supported potential configurations.

   CAP(B): B's supported potential configurations.

   CAP(AB): The set of potential configurations supported by both A and B.

   CFG(AB): The set of actual configurations to be used.

   CFG_T(AB): The set of actual configurations (for the different components) to be used augmented with all required parameters.

In this model, the capability negotiation and configuration exchange process leads to a description that represents a global view of the configuration that should be used. This means, it contains the complete configuration for all participants including per-participant information like transport parameters.

Note that the model presented here results in four SDPng messages. As an optimization, this procedure can be abbreviated to two exchanges by including the transport (and other) parameters into the potential configurations. A embeds its desired transport parameters into the list of potential configurations and B also sends all required parameters in the response together with

`B's` potential configurations. Both `A` and `B` can then derive `CFG_T(AB)`. Transport parameters are usually not negotiable; therefore they have to be distinguished from other configuration information.

The collapsing of two capability descriptions is performed by applying the *feature-matching* algorithm of RFC 2533 [RFC2533]. In [Kutscher03a], we have specified the concrete procedures of mapping SDPng documents to RFC 2533 expressions and back and have defined a procedure that involves the following steps:

- translating SDPng potential configurations to RFC 2533 feature set expressions;
- applying the RFC 2533 feature match algorithm; and
- integrating the resulting feature set expressions into the SDPng selection of actual configurations.

For the translation of SDPng potential configuration to RFC 2533 feature set expressions, all attributes of an SDPng capability element and its child elements are transformed to an RFC 2533 expression, and each attribute is translated to a feature predicate. The resulting feature predicates are combined using the `&` (`AND`) operator. The name attributes are not considered. For example, an SDPng element describing video codec capabilities can be transformed as follows:

```
<video:codec name="h263+-enhanced" resolution="QCIF"
        frame-rate="(,24)" h263plus:A="foo" h263plus:B="bar"/>
```

```
(& (resolution=QCIF) (frame-rate<=24) (h263plus:A=foo)
    (h263plus:B=bar))
```

Multiple individual capability elements are independently transformed using the specification above and then combined into a single RFC 2533 feature set by connecting the individual feature sets using the `|` `OR` operator. For example, the following sample SDPng potential configuration would be transformed as follows:

```
<audio:codec name="avp:pcmu" encoding="PCMU" channels="[1,2]"
             sampling="[8000,16000]"/>
<video:codec name="h263+-enhanced" resolution="QCIF"
             frame-rate="(,24)" h263plus:A="foo"
             h263plus:B="bar"/>
```

```
(|
  (& (encoding=PCMU) (channels=[1,2]) (sampling=[8000,16000]))
```

```
  (& (resolution=QCIF) (frame-rate<=24) (h263plus:A=foo)
     (h263plus:B=bar))
)
```

After transforming different SDPng capability descriptions from different participants into their equivalent RFC 2533 form, the following steps are performed to calculate the common subset of capabilities:

1. The individual feature sets are combined into a single expression by creating a conjunction of the feature sets, i.e., the feature sets are connected by the `&` (`AND`) operator.

2. The resulting expressions are reduced to disjunctive normal form, i.e., the canonical form as specified by RFC 2533.

A feature set that has been created by combining multiple independent feature sets and by reducing the result for canonical form does not indicate directly which of the capability elements belong the common subset of capabilities. The following steps are performed to determine whether an individual capability element (e.g., an element pertaining to one of the contributing SDPng capability descriptions) belongs to the result feature set.

Let `R` be the result feature set obtained from the canonicalization:

1. For each capability element, generate the equivalent RFC 2533 feature set by applying the transformation described above. Let `C` be the resulting feature set.

2. Combine `R` and `C` into a single feature set by building a conjunction of the two feature sets (`& R C`). Let the result be the feature set `T`.

3. Reduce `T` to disjunctive normal form by applying the canonicalization as defined in RFC 2533.

4. If the remaining disjunction is non-empty, the constraints specified by capability element (the origin of `C`) can be satisfied by `R`, i.e., `C` represents a commonly supported capability.

9.2.3 SDPng in the Local Coordination Architecture

In Section 9.1.3, we have described how a controller interacts with its application entities and call control components during the initialization of an endpoint and during the conference establishment phase. With respect to session and capability description, the key idea is to provide a general, application-independent framework for processing capabilities of conferencing systems — processors know nothing about application semantics but are nevertheless able to process, i.e., to aggregate and to negotiate, configurations. In our local endpoint architecture, a controller can thus query the capabilities of the present application entities, aggregate this information into a combined capability description of the whole endpoint and can then coordinate a call control engine through Mbus Call Control commands to consider this capability description for the negotiation process during a call setup. Later, when the call setup has been performed successfully, the controller has to analyze the resulting conference configuration and assign each application entity a proper configuration for its own application session. The individual steps in this process can be described as follows:

1. The controller queries the available application entities for their capabilities. In general, each application entity provide capabilities for processing media data for a specific media type; however, there can be multiple application entities associated to a specific media type.

2. The controller aggregates these descriptions (in an application-independent way). It matches the resulting description against configured user preferences and obtains a capability/preferences description for the local endpoint.

3. For establishing a call, the user can select specific media types, applications and configurations, i.e., she can refine her preference settings. Based on this updated capability/preference description, the controller queries the application entities for initial transport parameters, i.e., the IP address and UDP port number for receiving RTP video streams, i.e., the capability description is augmented with transport parameters. Note that for simple application, the initial query for capabilities and the subsequent query for transport parameters can also be performed in one step.

4. The resulting capability description (including the initial transport parameter specification) is then taken as input for the call establishment and therefore passed to the call control engine. The call setup and the capability negotiation result in a collapsed capability description as described in Section 9.2.2.

5. The resulting conference description is processed by the controller. For each component, the controller passes the corresponding actual configuration to the responsible application entity. Each application entity adopts the configuration and the conference can commence.

6. Later changes in the configuration are in principle handled in the same way as the configuration for the conference setup: The controller negotiates an interoperable configuration and provides the corresponding application entities with an updated configuration.

We therefore do not only rely of SDPng's application independent processing concept for inter-system capability negotiation but also for the local processing. Thus, the session description is one important element in our generic endpoint/gateway architecture.

9.2.4 Summary

The SDPng conference description language has been designed as a tool for providing the *interoperability service* for conferencing systems that require more elaborate capability negotiation mechanisms than the currently used RFC 3264 procedures. By not tying application semantics to the base framework and capability negotiation rules, SDPng provides extensibility for future applications and allows for the development of application-independent SDPng processors, such as generic endpoint controllers and gateway controllers.

The standardization of SDPng, which is still in progress, has shown that the requirements for a description and negotiation framework for different application domains can be partly contradictory. For example, the requirement for extensibility and expressiveness implies more powerful syntax mechanisms than those provided by SDP. We have addressed this by developing the XML based SDPng base syntax and by the use of corresponding XML mechanisms

such as XML namespaces. Obviously this requires implementations to process structured XML documents, e.g., by employing an XML parser. Certain conferencing systems, such as optimized IP telephony applications for small devices, run on rather limited hardware platforms; as a consequence developers of these systems typically shirk the assumed complexity involved with parsing XML documents and performing negotiation steps. We try to accommodate limitations of small devices by reducing the implementation requirements as far as possible, e.g., by not requiring validating parsers.

Another example relates to the SDPng feature for expressing constraints across different potential configurations, e.g., by expressing that a system can support a fixed number of codec instantiations at a time, possibly considering specific combinations of codecs. We have learned that while such a feature is useful for expressing the capabilities of media gateways (that can provide limited resources such as DSP capacity), it is not really interesting for user endpoints, where the simultaneous instantiation of multiple codecs is either not limited or not likely to be useful anyway. We have addressed this by making the constraints feature (and other special-purpose mechanisms) optional, i.e., conforming implementation do not have to support it.

One interesting side result of the SDPng development was the detailed analysis of the different usages and extensions of SDP that are available today (and are still being developed). The evolution of multimedia conferencing and the increasing commercial deployment have led to a huge set of new requirements that have never been anticipated when SDP was originally conceived. For example, security requirements lead to the development of key management procedures for SDP [Arkko03], connection-oriented media transport [Yon03] has been discussed very controversially and many other extensions have been or are still being defined. These developments confirm the need for a structured description language that guarantees future extensibility and interoperability between heterogeneous end systems.

For our Mbus-based endpoint-architecture, SDPng is a useful element for implementing the *interoperability service* in a generalized fashion, e.g., allowing controllers to obtain and process capabilities and configurations regardless of their specific type — a generalization that is especially important for systems with higher complexity such as media transcoding gateways.

9.3 Mbus Call Control

Building on the Mbus base protocol and the Mbus Guidelines interactions, we have defined an Mbus application profile for call control services. This profile defines a command set and corresponding interactions between application components for basic call control services, such as *call setup* and *call termination*. The set of basic call control commands also includes commands for redirecting or forwarding (proxying) call setup requests and is supplemented by a set of additional commands for supplementary services, such as *call hold* and *call transfer*.

The Mbus Call Control Profile is essentially a *protocol* that provides the service of controlling a call control engine. In accordance to the coordination model described in Section 9.1, the protocol abstracts from the specific type of the call control engine, i.e., the protocol can be used to control a SIP call control engine as well as an H.323 call control engine. Hence, it is intended to enable the development of general frameworks for conferencing endpoints and gateways that can abstract from specific call control protocols and to promote the re-use of call control components in different application contexts.

There are a number of different APIs, protocols and so-called service architectures that

provide related and partly similar functions. Without going too much into detail, we will briefly differentiate our approach against this work in the following.

TAPI, JTAPI: TAPI is an API for computer telephony integration (CTI) that provides abstractions for phone control and is intended for call center (and similar) deployment scenarios. Typical examples for TAPI functions are `tapiRequestMakeCall` and `lineDial`. Later versions of TAPI such as TAPI 3.0 [Microsoft99] have merged the classic telephony model with IP telephony and extended the notion of a *call* to H.323 sessions.

JTAPI (Java Telephony API) [Sun99] falls into the same class and is a Java API for basic call control services for call center scenarios. It provides similar concepts as TAPI: the main idea is that applications are built against the JTAPI and a so called *service provider* implements the call control actions, e.g., the application runs on a PBX and a vendor-supplied service provider implements the control of telephony devices.

Both TAPI and JTAPI are clearly APIs that are targeted at a specific platform (TAPI is a MS Windows API and JTAPI is a Java API).

JAIN and Parlay OSA: JAIN (Java APIs for Integrated Networks, [Sun03]) is not a single API but rather a comprehensive set of Java APIs that is in general targeted at telecommunications application development. The approach has evolved from a set of APIs for service creation in so called Intelligent Networks (IN) but is being extended to support Internet multimedia conferencing protocols as well.

The JAIN family of APIs is providing different interfaces for SIP and H.323 engines (amongst other protocols) and is providing so called application API specifications for applications such as instant text messaging and payment. In essence, the JAIN APIs represent standardized APIs for certain protocols and services and are intended to promote the development of telecommunications applications in the Java environment.

The Parlay Open Services Architecture (OSA, [Parlay03]) is, similar to JAIN, a comprehensive set of APIs for service creation in telecommunications networks. As such it also provides APIs for call control and similar functions. Parlay OSA defines the interfaces to services by the use of different interface definition languages such as CORBA IDL and WSDL (Web Services Definition Language).

The different APIs and abstract call control interfaces demonstrate a clear demand for standardized interfaces and generalized control facilities for call control engines that abstract from specific call control protocols. The Mbus Call Control Profile provides the same abstraction but is not an API for certain programming languages but rather a *protocol* for the coordination of call control components in a distributed conferencing system. In an endpoint or gateway system that is based on the Mbus Call Control Profile, a separate controlling component, denoted *controller* in the following sections, implements the application logic and controls one (or more) call control engines using the Mbus commands specified in the call control profile. This architecture is depicted in Figure 9.3.

The Mbus Call Control Profile specifies the communication mechanisms between a controller and a call control engine within an Mbus domain, based on the transport mechanisms specified in the Mbus transport specification (Section 6.2) and based on the interaction schemes defined in the Mbus Guidelines (Section 6.3), i.e., the different interactions are mainly defined

in terms of RPCs and event notifications. In order to accommodate other call signaling protocols besides SIP, the interactions that are defined here provide a sufficient level of abstraction from specific call control protocols. This abstraction implies that not every feature of every specific call control protocol can be provided. The trade-off between generality and functionality/specificity results in a call-control model that

- supports basic, common call control services;
- provides a small set of advanced functions, such as supplementary services;
- uses universal addressing schemes for callee addresses and other parameters; and
- provides hooks for call control protocol specific extensions, such as optional parameters.

For an architecture such as the one depicted in Figure 9.3, the provided generality would allow to replace the SIP call control engine by an H.323 engine without having to change the implementation of the controller, e.g., in order to implement a SIP *back-to-back user agent* (B2BUA, a specific implementation is described in Section 10.3.3).

In Section 9.3.1, we describe the fundamental abstractions that the Call Control Profile is based on, e.g., the *call abstraction*. In Section 9.3.3 we describe the protocol procedures for certain basic call control features, and in Section 9.3.2 we present the implementation of these procedures into Mbus commands. We summarize the main ideas and report on our experiences with this work in Section 9.3.4. The complete specification of the Call Control Profile is provided by [Ott01].

9.3.1 Concepts

The call control profile relies on a set of concepts, abstractions and identifiers that are used by the presented call control model. This includes:

- identification of calls (through a call and conference identification as described below);
- addressing concept for participants and endpoints (relying on URIs as described in Section 9.3.2); and
- call state manipulation (through a set of Mbus RPCs and event notifications as described in Section 9.3.2).

Controlling a call control engine by a controller uses the notion of *a call*, which is an abstraction that represents the state of a call control relationship that is setup, modified and terminated by means of message exchange between a controller and a call control engine. In order to disambiguate multiple calls that are managed by a system, call identifiers are employed. Different types of identifiers are used:

Call Identifier: A call identifier is used to identify calls uniquely. In this model, *a call* represents a call control relationship between two endpoints. If an endpoint has a call control relationship to two other endpoints at the same time, two different call identifiers will be used to disambiguate the call states. The concept of a globally unique call identifier is

prevalent in most call signaling protocols as well. For the Mbus call control commands, the call identifiers are generated by the call control engine and are considered opaque values by other components, e.g., a controller. The appearance of the call identifier depends on the call signaling protocol. See H.225.0 [ITU00] and SIP [RFC3261] for details.

Call Leg Identifier: Call leg identifiers allow for a more fine grained control of call control relationships. A call control engine may try to setup more than one outgoing call at a time in order to establish a call control relationship to a participant, e.g., when the call control engine is a component in a forking proxy system. In order to disambiguate the different call legs that are created for a single call, the notion of call leg identifiers is introduced.

Conference Identifier: While a call identifier is used to identify individual call control relationships, there are also more persistent states, e.g., multi-party conferences. In some models, multi-party conferences can be implemented by creating a full mesh of calls between all participants. In this case, the individual calls would be disambiguated with call identifiers, while the conference itself is identified by a *conference identifier*. In the Mbus Call Control Profile, this identifier is also used to implement call transfer. The transfer of a call is implemented by having the transferor initiate a new call to the transferred-to party, which results in a new call with a new call identifier. In order to be able to identify and track the call, it is assigned a persistent conference identifier.

9.3.2 The Mbus Call Control Profile

Based on the concepts described in Section 9.3.1, we have defined an Mbus application profile that provides an addressing scheme and different Mbus commands for implementing the call control interactions. In Section 9.3.2.1, we list some key terms and Mbus parameter types that are used in the application profile and in Section 9.3.2.2, we present the addressing scheme and the employed control class. Section 9.3.2.3 describes the general structure of the Call Control Profile.

9.3.2.1 Mbus Parameter Type Definitions

The following key terms are used in the profile definitions:

Call reference: In most of the Mbus commands, a call reference is used to identify calls. Call control engines can map the call reference to call identifiers of their call signaling protocol. The Mbus parameter data type for call references is `string` and abbreviated as `call-ref` in the specification below. References are created by `call` or `incoming-call` commands. Every newly created call reference is composed of the Mbus address of the creating entity and a second entity specific part in order to ensure uniqueness.

Some commands such as the call setup command and the incoming call indication make use of conference identifiers in order to group several calls within a single conference, i.e., for a conferencing bridge. Once a call context has been created (and the conference identifier is one element of such as context), the call context is referred to using the call

reference identifier. In addition, some commands, e.g., the call acceptance indication and the corresponding command to finally establish a call of a specific call leg, can provide a call leg identifier in order to distinguish multiple call legs of a single call, e.g., for a forking proxy that performs parallel searching.

Address: Some commands require the specification of an address (or address list) for users. These addresses are self-contained URIs that allow to identify the call control protocol domain and the call control domain specific information that is required to setup a call control relation to the specified user. One of following scheme identifiers are used:

- `sip:` for SIP URIs;
- `h323:` for H.323 URIs; and
- `tel:` for telephony URIs as specified by [RFC2806].

The scheme specific part of an address URI contains the protocol specific information that is required for establishing a call control relationship.

The Mbus parameter type for an address is called `address` in the specification below. The Mbus type for `address` is `string`. Address parameters are used in requests to call control engines that should be able to translate them into native addresses of their corresponding call signaling protocol.

Address list: For some commands, more than one address needs to be passed as a parameter. The type `address-list` is defined as a `list of address` and is used as a parameter type for requests where more than one address can be specified.

Logical address: A logical address is an informational address that denominates the user that a caller is trying to call. The logical address is not necessarily identical to the address URI described above. For example, in a SIP `INVITE` request, the Request-URI may be `sip:123434565@big-company.foo` (which may have been obtained from a location server), whereas the logical address is `sip:support@big-company.foo` (in SIP, this could be the content of a `To` header field). As the `To` header field in SIP, the logical address can be augmented by a *display name* that can be presented to a user by a user agent. As an Mbus parameter, the logical address is therefore represented as a list of two elements (both of type `string`), where the first element is the display name and the second element is the address URI, for example:

```
("Help Desk" "sip:support@big-company.foo")
```

In the command specification below, the type for logical address parameters is called `logical-address`.

Status codes: Some of the commands defined below can be parameterized with status codes and reason descriptions that represent error conditions (or other status information). On the Mbus, this information is represented as a list of two `strings`, where the first element is a numerical status code and the second element is a textual description. In the command specification below the type for status information parameters is called `status`. The status codes are derived from SIP status codes.

Media: Some commands provide a media parameter list and/or a capability list for media settings for the call. SDP or SDPng is used for describing session parameters and capabilities. The Mbus parameter type `media` is a `pair of (symbol, data)`, where the first element identifies the type of the description language and the second element is the actual description.

In order to allow for expressing preferences with SDP, some commands use a `list of media` for media description parameters. In these lists, the order of the media elements (each of which represents a stand alone SDP description) defines their relative preference.

Table 9.1 provides an overview of the parameter data types of the Mbus call control profile.

Table 9.1: Overview of the call control parameter types

Type name	Mbus type definition	Description
call-ref	string	Call Reference
address	string	Address URI
address-list	list of address	List of URIs
logical-address	pair of string	Logical Address
status	pair of string	Status Information
media	pair of (symbol, data)	Media Information

9.3.2.2 Mbus Addressing Scheme

The following Mbus address fields are used by implementations of the call control commands:

`function:` The address element `function` describes the general function of the component. The value is fixed to `call-control` for both controller and call control engine.

`cc-module:` The address element `cc-module` describes the type of the component. The possible values are `controller` (to be used by controller entities) and `engine` (to be used by call signaling engines). Figure 9.6 depicts a sample Mbus address for a controller, and Figure 9.7 depicts a sample Mbus address for a call signaling engine.

```
(function:call-control cc-module:controller id:123-4@192.168.1.1)
```

Figure 9.6: Mbus address for a controller

The default destination address that is used for event notifications by call control engines not yet controlled, is `(function:call-control)`. In Section 6.3, we have described different control classes for applications consisting of modules with controller-controllee relationships. Implementations of the call control profile implement the control class `tight control`,

```
(function:call-control cc-module:engine id:124-4@192.168.1.1)
```

Figure 9.7: Mbus address for a call signaling engine

which means that a controllee (a signaling engine) can only be controlled by one controller at a time. A controller will therefore take over the control of a call control engine — using the `mbus.register` command (Section 6.3.3) — before it can send commands to a call control engine. The command prefix for the call control commands is `conf.call-control`. This means, a controller registers itself for the `conf.call-control` hierarchy.

9.3.2.3 Mbus Commands

The Mbus Call Control commands can be divided into two classes:

- RPCs; and
- event notifications.

RPCs are sent from a controller to a call control engine. Call control engines must support all RPCs, i.e., they must be able to receive and understand them. Where possible, the imperative form has been chosen for RPC command names, e.g., `call` and `cancel`. Event notifications are sent from a call control engine to a controller. All event notifications must be supported by controllers, i.e., they must be able to receive and understand them. Where possible, the past (or present) participle form has been chosen for names of event notification commands, e.g., `connected` and `proceeding`.

In the following, we will briefly describe the commands of the Mbus call control profile. All of these commands are associated to a specific call context, which is expressed by a call reference parameter. Note that we do not mention this parameter explicitly in the following overview. In general, we also do not describe optional parameters that some commands provide. The detailed specification of each command is provided by [Ott01]. It should be noted that some of the commands described in the following sections provide *lists* of addresses, e.g., destination addresses for a `call` command, whereas for most basic applications, a single address parameter would actually be sufficient. However, in order not to exclude non-trivial applications, we have generalized the specification of destination addresses. A forking SIP proxy is a sample application where the specification of multiple destination addresses would be useful.[3]

Moreover, it should be noted that essentially all commands that are used for establishing calls can provide a media list (for conference configuration descriptions), even though this might not be required for every call setup process. This is motivated by the requirement for generality, because, e.g., H.323 allows changing conference configurations at any time (through H.245).

[3] A *forking* SIP proxy is a proxy that can send multiple `INVITE` requests for a single incoming `INVITE` request, which is useful for implementing *parallel searching*. A SIP proxy that obtains multiple possible contact addresses for an `INVITE` request can thus try to establish calls simultaneously and later cancel all calls except for the first successfully established call.

Another motivation is that, for transcoding gateway scenarios, the outgoing call is typically setup by *not sending* a capability description in the initial setup message, because the gateway has to learn the capabilities of both endpoints first in order to make a decision with respect to a suitable transcoding configuration. In such a scenario, the gateway would send its configuration in its second message to the callee, e.g., in a `SIP ACK` request. In order to support these different scenarios, the media list is provided by many call control commands in this profile, but may be empty in case it is not required, e.g., for the `conf.call-control.connect` RPC.

9.3.3 Basic Services

The Mbus Call Control Profile provides the following basic call control services:

Initialization: When a controller and a call control engine are brought into a common Mbus session, they locate each other through the Mbus entity awareness mechanism. Subsequently, the controller can register with the call control engine and perform the initial configuration. We describe the corresponding procedures in Section 9.3.3.1.

Call setup: A controller can make a call control engine initiate a new call using its native call signaling protocol. The call control engine will notify the controller of progress events, e.g., when the called party accepts the call. For a called endpoint, the call control engine will signal incoming call events it received via its native call signaling protocol, enabling the controller to react and eventually control the completion of the call setup by accepting the call. See Section 9.3.3.2 for a detailed discussion of call setup procedures.

Call redirection: After the call control engine has signaled an incoming call, the controller can request the call control engine to redirect the call to another endpoint. See [Ott01] for a detailed discussion of call redirection procedures.

Call forwarding and call proxying: After the call control engine has signaled an incoming call, the controller can request the call control engine to proxy the call to another endpoint. See [Ott01] for a detailed discussion of call forwarding procedures.

Call canceling and call rejection: Outgoing and incoming calls can be rejected and canceled by the controller at any time. See [Ott01] for a detailed discussion of call rejection procedures.

9.3.3.1 Initialization

During an initialization phase, the call control engine is bound to a specific controller and configured initially. This initialization involves the steps *discovery*, *establishment of the control relationship* and *configuration*, as described in Section 9.1.3.

The Mbus Call Control Profile relies on the *tight control* service model as described in Section 6.3.3. This means, a controller must register with a call control engine before it can start to send control commands. The registration also makes the call control engine adopt the controller's Mbus address as a destination address for event notifications. To some extent, the initial configuration process that takes place after the registration depends on the specific

system configuration and the application type. For example in Section 9.1.3, we have described how a controller performs an assessment of the application entities' capabilities and aggregates this into a comprehensive capability description for the whole endpoint. In a second step, the controller provides the call control engine with this capability description, as the call control engine would use this information for the later call initiation. The `caps-set` command of the Call Control Profile that we describe below is used for this purpose. The initial capability configuration process is however not applicable to all applications. For example, a signaling gateway that only translates signaling communication would typically not provide capabilities of application components on its own. Instead it would merely relay the capability negotiation communication between two endpoints that use the gateway as a mediator.

For the Mbus Call Control Profile, the initialization phase is also intended to configure the call control engine using call signaling protocol specific or even implementation specific configuration commands. The Call Control Profile provides the concept of generic, i.e., call signaling protocol independent, and call signaling protocol specific commands. By handling the protocol specific communication during the initialization phase, it is possible to perform the actual call state manipulation interactions in a generic, call signaling protocol independent fashion. An example for a protocol specific configuration is the `sip.register` command that a controller can send to a call control engine to register the SIP user agent with a SIP registrar.

conf.call-control.caps-set

The `conf.call-control.caps-set` RPC is sent by the local controller to the call control engine to configure a default capability description for the endpoint (typically during the initialization phase), which is used by the call control engine to negotiate conference negotiation during the call setup (typically of incoming calls). The `caps-set` RPC provides a media list parameter that contains the capability description. A default capability description enables call control engines to complete the call setup without requiring a complete description in a `conf.call-control.accept` RPC.

9.3.3.2 Call Setup

Figure 9.8 provides a schematic visualization of the Mbus communication for setting up and terminating a call. In particular, the figure shows the message flow for a calling party `A` as well as for a called party `B`. `A's` controller initiates the call setup with a `call` message sent to the call control engine. The call control engine would subsequently setup a call using its native call signaling protocol (SIP in this example). The most important parameters of the `call` message are the address of the callee and a media/capability description to be used for the call.

In case a call control relationship with the callee can be established, `A's` call control engine will notify the controller of call progress indications it received via its call signaling protocol. When `B` has accepted the call, `A's` call control engine will notify the controller with an `accepted` message, which must be acknowledged by sending a connect message back to the call control engine. In essence, this mimics a three-way-handshake model that allows some basic form of call parameters negotiation, as employed by, e.g., SIP [RFC3261]. For this purpose, both the `call` and the `accepted` message can be parameterized with media/capability descriptions.

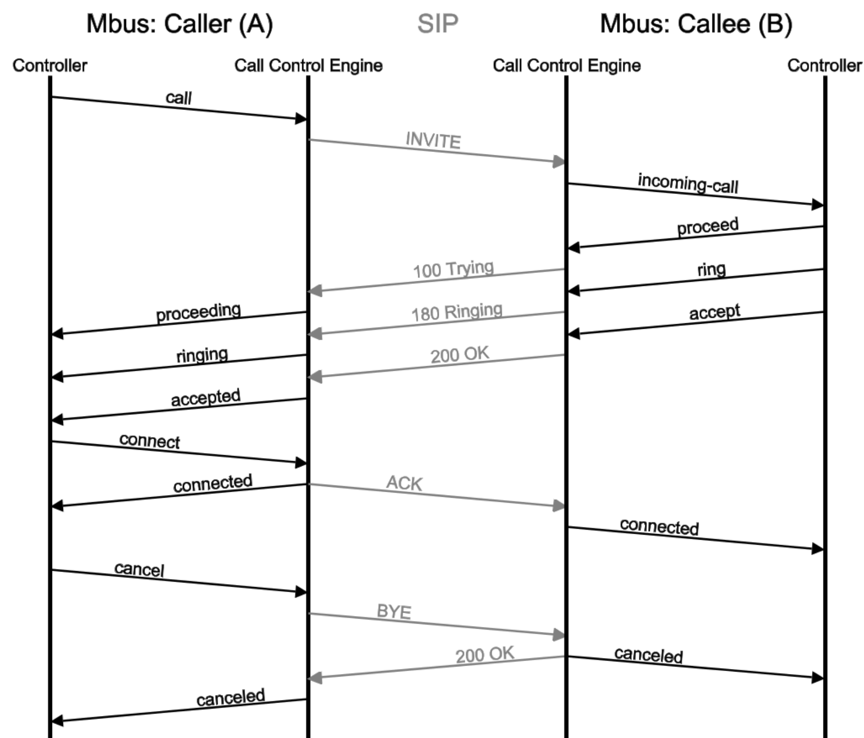

Figure 9.8: Call setup model

In this example, the call is terminated by `A's` controller that is sending a `cancel` message to its call control engine, which subsequently terminates the call control relationship to `B`. Both `A's` and `B's` call control engine notify their controllers with a `canceled` message. A `cancel` message can be sent at any stage of a call setup phase in order to terminate the call and cancel the call control relationship. The following Mbus commands are used for establishing and terminating a call as depicted in Figure 9.8:

conf.call-control.call

The `conf.call-control.call` RPC is used to setup a new call and is sent by the local controller to the call signaling engine. The `call` RPC provides parameters such as the destination address, a media list and a caller identification. The RPC returns unique call and conference identifiers for the new call.

conf.call-control.incoming-call

The `conf.call-control.incoming-call` event notification is sent by the call control engine to the local controller to indicate a call request from another endpoint. The `incoming-call` event notification provides caller addresses, callee addresses, a media list, and call and conference identifiers as parameters.

conf.call-control.proceed

The `conf.call-control.proceed` RPC is sent by a local controller to a call control engine in order to indicate that the call setup that has been signaled with an `incoming-call` RPC is still proceeding. A call control engine should restart its timers for call setup timeouts (if applicable) and translate this command to a protocol specific message, e.g. a `SIP-TRYING` or a `Q.931-CALL-PROCEEDING` message[4] , that is to be sent to the originating party.

conf.call-control.proceeding

The `conf.call-control.proceeding` event notification is sent by a call control engine to a local controller in order to indicate that the call, which has been initiated with a `call` command, is still proceeding. The call control engine will usually send this command after it has received an according message in its call control protocol, e.g. a `SIP-TRYING` or `Q.931-CALL-PROCEEDING` message. The reception of a `proceeding` command does not imply that a user has already been contacted. It merely expresses that the call setup is still in progress. The `proceeding` event notification provides a peer endpoint address list as a parameter.

conf.call-control.ring

The `conf.call-control.ring` RPC is sent by the local controller to the call control engine. The `ring` RPC indicates that the controller is willing to accept the incoming call and is now alerting the user. A gateway or proxy system should translate incoming `ringing` notifications into `ring` RPCs that are to be sent to the call control engine the incoming call was received from. The `ring` RPC provides a list of destination addresses (as URIs) as a parameter that represent the different callee URIs that are being tried, e.g., in case of a forking proxy. For an endpoint, the list will provide exactly one parameter.

conf.call-control.ringing

The `conf.call-control.ringing` event notification is sent by the call control engine to the entity it received the corresponding `call` RPC from. The `ringing` notification indicates that one or more endpoints have been contacted and are now alerting the user. The `ringing` event notification provides a list of destination addresses (as URIs) as a parameter.

conf.call-control.accept

The `conf.call-control.accept` RPC is sent by the local controller to the call control engine that has indicated an *incoming call*. By sending this RPC the controller indicates that the call should be accepted. The `accept` RPC provides a media list (the conference configuration description) as a parameter.

[4] Q.931 is the ISDN call signaling protocol as standardized by the ITU.

conf.call-control.accepted

The `conf.call-control.accepted` event notification is sent by the caller's call control engine to the local controller to indicate that the party has accepted the call.

conf.call-control.connect

The `conf.call-control.connect` RPC is sent by the local controller to the call control engine to complete the call setup after having received a `conf.call-control.accepted` notification. The `connect` RPC provides a call reference parameter.

conf.call-control.connected

The `conf.call-control.connected` event notification is sent by a call control engine to the local controller to indicate that the call has been established successfully. The `connected` event notification provides an address list with addresses of the peer endpoint and a media list as parameters.

conf.call-control.cancel

The `conf.call-control.cancel` RPC is sent by the local controller to the call control engine to indicate that the specified call is to be canceled. It can also be used by the local controller to inform the call control engine that a call has already been terminated by out-of-band communication, e.g., through horizontal conference control communication. The `cancel` RPC provides a reason specification (see Section 9.3.2.1) as a parameter.

conf.call-control.canceled

The `conf.call-control.canceled` event notification is sent by the call control engine to the local controller to indicate that the call was canceled. The `canceled` event notification provides a reason description as a parameter.

9.3.4 Lessons Learned

The Mbus Call Control Profile is our Mbus based protocol to coordinate call control engines in decomposed conferencing endpoints and conferencing gateways. It abstracts from specific call signaling and call control protocols and can thus be used to control different types of call control engines with a generic controller.

One technique to achieve this generalization is a clear distinction between generic and application specific communication. In general, the Mbus Call Control Profile is truly generic, however it is limited a set of fundamental call management operations. In order to allow for the implementation of applications beyond the scope the Mbus Call Control commands, we have sometimes used additional commands, arranged under a separate command name hierarchy, and have used these commands for specific applications. For example, we have currently not generalized the user registration process, but are using protocol specific commands instead, e.g., `sip.register`. Our deployment experience has shown that it is usually possible to configure

and control a call control engine with these specific commands during the initialization phase and then use the generic commands later on for the actual call control communication.

Another technique to achieve this generalization is the provision of generic command interfaces, i.e., parameter lists that are designed to support a wide range of applications, e.g., by allowing for optional parameters (using the corresponding Mbus Guidelines mechanisms described in Section 6.3).

The fundamental model of the profile provides a call control engine and a controller, where a controller can control multiple call control engines at the same time. The call control engines can stay application independent, i.e., they can be viewed as SIP or H.323 protocol stack components that can be used for both endpoints and gateways. In other words, they can be viewed as units of third-party composition. It is always the controller that provides the application specific logic and coordinates call control engines accordingly. The controller on the other hand does not necessarily have to know the exact type a call control engine and can largely abstract from the call control protocol specific mechanisms.

The Mbus Call Control Profile is based on the Mbus Guidelines, relies on the use of Mbus RPCs and event notifications and imposes the *tight control* model of the Mbus Guidelines. Our deployment experience has shown that this is a useful model for realizing a wide range of applications, from endpoints to gateway systems, some of which we will describe in Chapter 10, *Mbus in Projects*.

However, we have also experienced two problems that have to be mentioned: performance issues and lacking support for call control specific commands.

With respect to the number of interactions, a distributed approach can naturally not achieve the same performance as an optimized, integrated solution. Our discussion of the call setup procedure in Section 9.3.3.2 has illustrated the granularity of the communication, and our measurements of the RPC roundtrip performance in Section 8.2 have provided some absolute numbers for different scenarios. For a call setup process as depicted in Figure 9.8, there are two RPC messages and four event notifications. Assuming an RPC roundtrip time of 7 milliseconds and a transmission and processing delay of 3 milliseconds for one-way event notifications, this adds up to a total of 26 milliseconds delay introduced by the Mbus communication. Clearly, this is only a small fraction of the total call setup time (which is largely dependent on the delay caused by the wide-area call signaling communication[5]); however it can be a factor for systems that process many calls in parallel, i.e., proxy systems. Such specialized systems do usually not require the generality and dynamic extensibility of an Mbus Call Control based approach; hence, it is typically more appropriate to implement such systems as integrated applications. However, it is still possible to construct such an integrated system from a set of Mbus components by integrating the Mbus components into a single program context. Optimization strategies such as the *virtual Mbus interface* of the Java Mbus implementation as described in Section 7.4 can be used to map message-oriented Mbus communication into local functions calls, which can be done transparently for the involved Mbus module. With these observations in mind, it is fair to say that, although the call setup time for a single call is not dramatically increased by the distributed, component-based Mbus approach, the Mbus Call Control approach is not applicable to every application. It is suitable for systems that can benefit from a flexible and generalized architecture such as endpoints, transcoding gateways and conference bridges, but is less useful

[5]For example, the SIP call setup procedure consists of a three-way-handshake.

for specialized single-purpose systems such as SIP proxies.

We have seen that the *generalization* of the Mbus Call Control commands enables the development of generic, re-usable components. Call signaling engines such as SIP and H.323 engines implement the same Mbus Call Control commands and can thus be used by a generic controller, and the call signaling engines themselves can be used in different applications, e.g., endpoints and gateways. The generalization comes at a cost: the abstraction from call control specific features. We have to restrict the Call Control Profile to a set of commonly supported abstractions — basic call control interactions and a small set of commands for supplementary services. We have already mentioned that we have provided for separate command name hierarchies, for generic and for protocol-specific Mbus commands. In an endpoint, a controller that has the option to implement certain specific functions, e.g., supplementary services as defined by the H.450 series of recommendations, by sending H.323-specific commands. The controller could determine the type of the call control engine in advance and then adapt itself and the rest of the application. For endpoints, this loss of generality — although not optimal — could possibly be tolerated. For gateway systems however, problems can arise, because, even if one call control engine supports additional, protocol-specific commands, interoperability can only be achieved if both (or, more precisely: all) call control engines support the corresponding functionality. Moreover, in a call signaling gateway, it is not desirable for a controller to translate between the two protocol domains relying on protocol-specific interfaces. In practice, Mbus-based call signaling gateways will therefore rather be limited to the generalized baseline feature set of call control commands. We conclude that a balance between generalization and specific support must be found. For the current version of the Mbus Call Control Profile, we have defined a set of baseline commands for fundamental call control services, augmented by a set small set of commands for supplementary services. These commands have been defined with respect to the features that can be achieved with SIP and H.323/H.450 today. Future work will extend the command set as new SIP methods are defined that correspond to existing H.450 functions.

Next Steps

The current version of this profile is solely intended for so called *single-controller environments*, i.e., systems where exactly one controller is in command of a specific call control engine. For some applications it can be beneficial to allow for more than one controller. For example, in addition to a normal controller as we have described it, a user interface could directly manipulate call state, e.g., by sending `cancel` RPCs to the call control engine without having to route these RPC via the controller.

In [Meyer01], Meyer has described the architecture of an Mbus based IP PBX, i.e., an Mbus Call Control based supplementary telephony service module. The IP PBX module can be added as an Mbus module to existing telephony endpoints and act as an additional controller in order to implement the supplementary services. Meyer has noted that such a *multi-controller environment* requires additional coordination and synchronization mechanisms in order to guarantee consistency between the involved Mbus modules.

The Mbus Guidelines service models, e.g., the *tight control model* that is used here, already provides for registering multiple controllers at a single entity. The Mbus Guidelines specify the distribution of event notifications for these cases, i.e., the controlled entity forwards event notifications to all registered controllers. However, still there are consistency issues that need to be addressed. While there is no need to introduce ISIS-like causal ordering mechanisms

(because the controlled entity can act as central sequencer and impose a message order for the whole system), the Mbus Call Control Profile needs to be extended in two ways:

- all RPCs that cause a change of the call state (essentially all RPCs) must trigger an event notification that is sent to all registered controllers, thus updating the call state at all controllers; and

- RPCs (especially their return commands) must provide a way to indicate that the corresponding action cannot be performed because the call state has changed in the meantime, i.e., by an RPC that has sent concurrently.

We would accomplish this by the following (rough) design sketch: The call control engine maintains the call state and is the only authoritative source of call state update events. Controllers can only send RPCs and each RPC triggers a state update that is multicast to all registered controllers. Each state update provides a sequence number that is increasing monotonically. Each RPC must reference the call state it is referring to by providing this sequence number as a parameter. The call control engine can use this sequence number to decide whether the requested action can be performed and notify the sender if this is not the case.

9.4 Summary

In this chapter, we have presented a model and corresponding technologies for decomposing multimedia conferencing systems into independent re-usable components that can be coordinated in a distributed system using the Mbus protocol. Our approach is certainly not the only way to build conferencing systems and is certainly also not the only way to modularize applications. However, our analysis of existing protocol families and architectures for multimedia conferencing in Chapter 2, *Conferencing Architectures*, our use case considerations in Section 3.1 and our description of existing APIs for call control in this chapter have led us to the following observations that, as we believe, are valid regardless of the specifics of the decomposition and coordination technologies that are derived from these insights:

- The control protocols and the application protocols of the H.323 and Internet Multimedia Conferencing Architecture domain are intended to provide interoperability between heterogeneous conferencing systems (the main objective for developing a standard).

- Multimedia conferencing systems are inherently modular systems and do typically not provide a monolithic structure. Instead there are application entities that provide a well defined function, there are signaling protocol implementations, user interfaces and there are often local coordinating entities.

  The technologies by which these modules are integrated into coherent applications can differ. For example, the modules can be represented as object code libraries and be linked to a single program, they can be components of some local component framework such as COM or they can be processes in distributed systems that employ a protocol for coordination.

- A fundamental concept for this modular approach is the notion of *internal management* that we have discussed in Section 9.1: the local coordination services can be classified into certain function groups. There is the *interoperability service* that aims at generating interoperable configurations of conferencing systems and conferencing sessions, and there is the *coordination service* that aims at achieving consistency during the operation of conferencing systems.

- Developments such as simple CTI solutions but also more ambitious approaches such as JAIN demonstrate that there is a need for decoupling applications from conference control implementations. Obviously the motivation is to facilitate the application development by providing well-defined, standardized interfaces, thus allowing application developers to acquire and deploy third-party modules.

With these considerations in mind (the need for modular systems, for internal management and for standardized interfaces), we have designed a distributed coordination framework and have presented two selected aspects of internal management services in this chapter: the SDPng framework as a tool for realizing the interoperability service and the Mbus Call Control Profile as a protocol for coordinating call control components of conferencing systems.

The main motivation of the SDPng work was to provide a framework for conferencing systems that allows for the dynamic assessment, aggregation and inter-system negotiation of interoperable configurations of conferencing systems and conferencing sessions. In our local conferencing system scenario, SDPng is intended to allow a generic controller to query and process the capabilities of its application entities, to aggregate them into a comprehensive capability description of the endpoint and to have this description be used by a call control engine during the conference initiation phase in order to obtain an interoperable conference configuration.

For the coordination service, we have selected the call control function as a specific class of components that have to be coordinated in a conferencing system. The increasing maturity and deployment of call signaling and call control protocols such as SIP and the H.323 protocols underlines the requirement for a well-designed architecture and standardized interface to these components as many, if not all conferencing systems require their services. Our Mbus Call Control Profile is a *protocol* for the control of call control components in locally distributed systems and not a API for a particular environment. The service of the protocol is the distributed management of call state, and as a protocol, the Call Control Profile is independent of programming languages, operating systems and (within the limits of an Mbus session) independent of the location of protocol entities. It provides a high degree of abstraction and is also independent of specific call control protocols. The generality of the protocol allows for re-using call control components in different application scenarios, e.g., in endpoints and in gateways, and it allows for developing generic controllers and user interfaces that can be implemented independent of the specific type of a call control engine.

We have also discussed some limitations, namely the overhead introduced by a distributed approach with a fine-grained structure of interactions and the lack of support for specific call control functions, e.g., supplementary services that are only defined for a specific wide-area call control protocol. We have concluded that, given these limitations, the applicability of the Mbus Call Control Profile depends on the application type: for applications that require flexibility and extensibility, such as endpoints and call signaling and media gateways, the distributed, component-based Mbus approach is suitable, while specialized and optimized systems, such

as a call control protocol specific proxy server, would rather be implemented as an integrated application.

The Call Control Profile has been presented as a *sample* Mbus application profile for coordination in conferencing systems in order to demonstrate the definition of such as profile with respect to the general, application-independent rules that we have described in Section 6.3. The feature set we have presented in Section 9.3 is limited to basic services (and some selected supplementary services such as call transfer) but has shown the general applicability of Mbus based coordination to the problem of distributed call state managing. We have discussed possibilities for future extensions, i.e., support for *multi-controller environments* in Section 9.3.4.

Besides call control we have been working on other Mbus based coordination protocols such as an Mbus profile for RTP engines that we do not describe in this document. In addition, there are third-party Mbus based developments for conferencing systems. We describe some of this work, and also the application of the Mbus Call Control Profile to the development of actual conferencing systems in Chapter 10, *Mbus in Projects*.

Chapter 10

Mbus in Projects

The Mbus framework has been deployed by several projects at the University of Bremen and at other institutes and has in general shown to be a usable component integration mechanism for different application areas. Most of the Mbus applications that have been developed are related to the multimedia conferencing domain, e.g., decomposed endpoints and gateway systems. At our research group at the University of Bremen, the Mbus framework has essentially evolved into *the* application development and integration framework for component based systems. I.e., essentially all of our recent developments are Mbus based and are composed of re-usable Mbus components that are typically implemented in different programming languages.

Many (but not all) applications involve call control functions and are relying the Mbus Call Control Profile we have described in Section 9.3. We can classify the different Mbus application areas as follows:

Endpoint decomposition: The development of component-based multimedia conferencing systems has been the main motivation for the Mbus framework. In Section 10.1, we describe the *CONTRABAND* system, the first Mbus-based conferencing endpoint, and the *Bonephone* SIP endpoint, a SIP endpoint that uses the Mbus protocol to control its media engines. We also describe the Mbus-based audio engine *RAT* and the design of a newly developed Mbus-based video engine.

Mbus in desk area environments: Based on the idea of extending the scope of a component-based conferencing system to a local network, we have developed and implemented the concept of *Desktop Telephony Integration* (DTI), using the Mbus and the Dynamic Device Association Procedures (Section 6.4) as main building blocks. The projects *DTI* and *FETA* (*Functional Enhancements using external Telephone Applications*) have been conducted in cooperation with a major manufacturer of IP telephony systems and are described in Section 10.2.

Gateways: The Mbus Call Control Profile has been developed not only considering the requirements for endpoint decomposition but also with respect to gateway architectures. In Section 10.3, we describe different implementations of Mbus-based call signaling and media gateways that we have developed in several projects.

Home-networking and ad-hoc communication: The ad-hoc and group communication features of the Mbus protocol suggest a wider application area than conferencing applications

only. In several projects, we have studied the applicability to other application domains such as home-networking and device control, which is described in Section 10.4.

It should be noted that we cannot describe all Mbus related projects here — instead we have selected representative developments that demonstrate the application of Mbus to specific application areas and provide results of deploying the different Mbus mechanisms.

In Section 10.5, we summarize the main results of studying the application of Mbus to projects.

10.1 Endpoint Decomposition

Endpoint decomposition has been the primary Mbus application area at the beginning of its development. The main idea is to use Mbus as a *vertical control protocol* for conferencing systems, i.e., as a successor to protocols such as PMM and the LBL control bus that we have described in Section 4.2.

The re-use of components has been one of the driving motivations for the Mbus based approach. For example, one major goal of many systems is to re-use existing application entities such as RAT, the robust audio tool, in new contexts. Many of these applications have originally been designed as stand-alone media tools that can be used for loosely-coupled conferences, e.g., SAP announced conferences, either with or without additional entities. In order to develop applications with *integrated user interfaces*, there must be a way for an application to take over the control of an application entity's *engine*, i.e., the module that provides the actual functionality and to replace the original user interface with an application specific one. Solutions to this and related issues will be presented in the following.

In Section 10.1.1, we describe the CONTRABAND conferencing system, our first Mbus based application and in Section 10.1.2, we describe the audio application RAT as an example of an application entity that is itself Mbus based and offers an Mbus interface to external controllers. One of the many applications that re-uses RAT through its Mbus interface, is the SIP endpoint *Bonephone* that we describe in Section 10.1.3.

10.1.1 CONTRABAND

In the *CONTRABAND* project (*Conferencing for Transport Breakdown and Networking of Dispatchers*, 1997-1999), we have developed a multimedia conferencing framework that we used to instantiate different end user applications, each of which could be adapted to specific application needs. We have named this concept *application tailored conferencing*.

One of CONTRABAND's application scenarios is accident management that can be supported by experts who remotely coordinate appropriate damage control measures. The analysis of video data transmitted using conferencing technologies allows for immediate response to unforeseen events. By eliminating the need for physical presence, sessions can be established quickly between the experts required; and by employing a set of appropriate media tools, observations can readily be shared among the participants. In the case of an accident, a customized conferencing environment provides for fast and uncomplicated conference set-up by automating the initiation process and the start of media tools etc. The application tailorability can be achieved by two mechanisms:

1. User interface components are decoupled from other functional modules and can thus be adapted or replaced for specific instantiations of the system. The user interface components are implemented in the scripting language Tcl, hence allowing for easy adaptation as well.

2. CONTRABAND provides conference control mechanisms, e.g., it uses a simplified variant of SCCP (Section 2.2.2.3) for horizontal conference control, and allows to model different types of conferences such as *emergency call* and *consultation* by so called *conference policies*, i.e., a set of rules that is implemented though conference control mechanisms.

The Mbus was used as CONTRABAND's local coordination mechanism. The system was decomposed into a conference control and user interface component, the SCCP conference controller, application entities, and call control and user location entities.

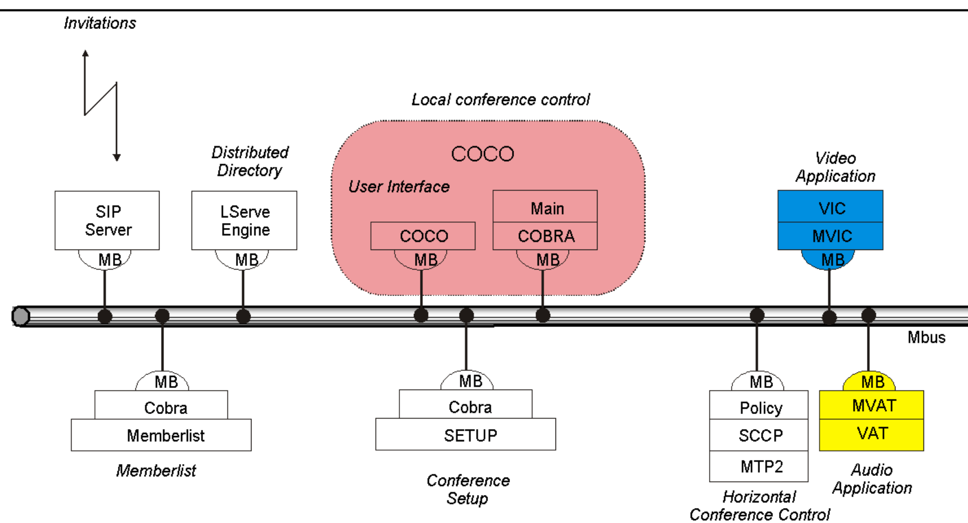

Figure 10.1: The CONTRABAND system architecture

Figure 10.1 depicts the CONTRABAND system architecture. The main control functionality is provided by the conference control component (COCO) that also represents the GUI. Some of the Mbus components have been implemented using the Mbus Tcl implementation we have described in Section 7.5, which is named *Cobra* in this diagram. In the early development phase of the CONTRABAND system, there were no Mbus enabled application entities. Hence we have developed an *Mbus adapter* in order to integrate legacy applications such as VAT. The Mbus adapter was typically a Mbus Tcl script that represented the corresponding application on the Mbus and could start, configure and terminate the application. For the audio component, the adapter has later been obsoleted by deploying RAT as an Mbus enabled application.

The local coordination service for conference control has been implemented by a direct mapping of wide area conference control messages to Mbus commands. For this purpose, we have developed an Mbus addressing concept that provides three required Mbus address elements for CONTRABAND components: `media` (for specifying the media type of an application entity), `module` (for specifying individual modules within an entity) and `app` for specifying

the application's name. Relying on the Mbus multicast transport through wildcard addressing, this addressing concept allows a controller to send a control message to all application entities of a certain media type, e.g., `(media:audio)`, to all application sub-components of a specific type, e.g., `(module:ui)` or to all sub-components of a specific application entity, e.g., `(app:rat)`. In this addressing model, application entities are conceptually structured into three sub-components: a media transport and rendering engine `(module:engine)`, a user interface component `(module:ui)` and a controlling component `module:control`.[1] Based on this convention and on generic control commands such as `mute` for enforcing floor control, we have implemented a basic coordination service. All coordination messages have been sent via Mbus multicast, i.e., using wildcard addresses, and have conveyed basic Mbus commands, i.e., there have been no higher layer interactions such as RPCs or event notifications.

We have developed different instantiations of the CONTRABAND framework, e.g., a desktop multimedia conferencing system. One of the more interesting instantiations is the development of a wearable, wireless conferencing end system that was intended as a mobile terminal in accident management scenarios: an endpoint (and its user) can be considered as a mobile video source that can provide direct visual impressions of the accident situation and can be directed via audio communication. The wearable CONTRABAND instantiation provides a reduced set of application entities (only audio and video), where video can only be sent but not received and rendered. The focus for this application lies on minimizing the need for manual configuration and user initiated control during a conference. Instead, conferences can be initiated by a single button activation (to a pre-configured callee). Figure 10.2 depicts the wearable system during a demonstration. The video camera and the display are head-mounted, and the wearable computer is mounted at the user's waist belt.

Figure 10.2: The wearable CONTRABAND conferencing endpoint

Summarizing, we can state that the CONTRABAND system represents a first Mbus based design of our endpoint architecture that we have described in Section 9.1. It addresses the re-

[1]This addressing model builds on earlier similar design as pursued by CCCP and PMM and is also used for the RAT audio application.

quirement for component re-use by its decomposition into application entities, control entities and user interface components. In the following section, we describe such Mbus based components and their interfaces that allow to integrate them into new application contexts.

10.1.2 Mbus-based Application Entities

The first set of widely deployed *Mbone tools* that includes the audio tool VAT and the video tool VIC incorporated the relatively simple LBL control bus that was used for lip synchronization related coordination. It was not possible to implement substantial coordination services based on this control mechanism.

RAT, the *Robust Audio Tool*[2] has been designed as an Mbone audio tool with a focus on enhanced audio quality that is implemented by mechanisms such as interleaving or redundant audio coding [RFC2198]. In 1998, the LBL conference bus in RAT was replaced by a first Mbus implementation. At that time, RAT was, just like other Mbone tools, a monolithic application: the main functionality for audio transport and audio rendering was implemented in C and the graphical user interface was implemented in Tcl/Tk.

In 1999, RAT was redesigned and decomposed into three modules: an audio engine, a graphical user interface and a controller that coordinates the whole application. All of these components run as individual processes that are coordinated in a common Mbus session. At the same time, the Mbus interface for controlling RAT as an application entity has been extended as well. I.e., RAT has become the first conferencing application entity that does only provide an Mbus interface in order to allow for external control, but is in itself a decomposed Mbus based application. The Mbus communication between the three RAT components relies on the addressing model described in Figure 10.1 (the obligatory `id` address element is not shown here):

- `(app:rat media:audio module:control)` is the Mbus address of the controller;
- `(app:rat media:audio module:engine)` is the Mbus address of the audio engine; and
- `(app:rat media:audio module:ui)` is the Mbus address of the user interface.

RAT uses basic Mbus messages for its communication, i.e., no RPCs or other higher layer interactions (although some message exchanges actually have RPC semantics). The Mbus communication between the three RAT components can be separated into three phases: initialization, operation and termination.

Figure 10.3 depicts the communication in a RAT Mbus session during the initialization phase. The dashed lines represent Mbus broadcasts. We have adapted some of the Mbus command names and omitted some messages for the sake of clarity. The RAT application is started by executing the control process, which in turn starts the user interface and the engine. All of these entities attach to the Mbus. For synchronizing the entities, RAT relies on the synchronization commands `mbus.waiting` and `mbus.go` that we have described in Section 6.2.6.

[2] RAT has been developed by Orion Hodson, Colin Perkins, Vicky Hardman, Isidor Kouvelas and others and the University College London.

When starting the user interface and the engine, the controller passes its own Mbus address and an opaque token as part of the program argument list. The controller starts the periodic sending of an `mbus.waiting` message with the specified token (starting with the token that was given to the user interface).

When the user interface is ready to communicate, it releases the synchronization lock by sending the corresponding `mbus.go` command. Subsequently it sends an `mbus.waiting` command itself that is to be answered, when the engine is ready. The controller broadcasts an `mbus.waiting` command that provides the token it has passed to the engine process as a program argument before. The engine responds by sending an `mbus.waiting` command with the same token. Subsequently, the controller provides the engine with a first (provisional) transport configuration by sending the `rtp.addr` command. After receiving this command the engine releases the synchronization lock of the controller by sending an `mbus.go` message, which makes the controller release the synchronization lock of the user interface and of the engine component.

The engine and the user interface are entering their own rendezvous process now. This is initiated by the engine that broadcasts an `mbus.waiting(ui-requested)` command, which is answered by the user interface. The user interface is now obtaining the capabilities and the initial configuration of the engine by sending the commands `tool.rat.settings`, `audio.query` and `rtp.query` (not shown in detail). The engine responds with a set of commands, each of which *adds* a certain capability (or configuration item) to the user interface state. All of these commands are sent using reliable Mbus transport. There is no explicit command to indicate that all parameters have been sent, i.e., the initialization phase is not formally ended but the engine could send new items at any time.

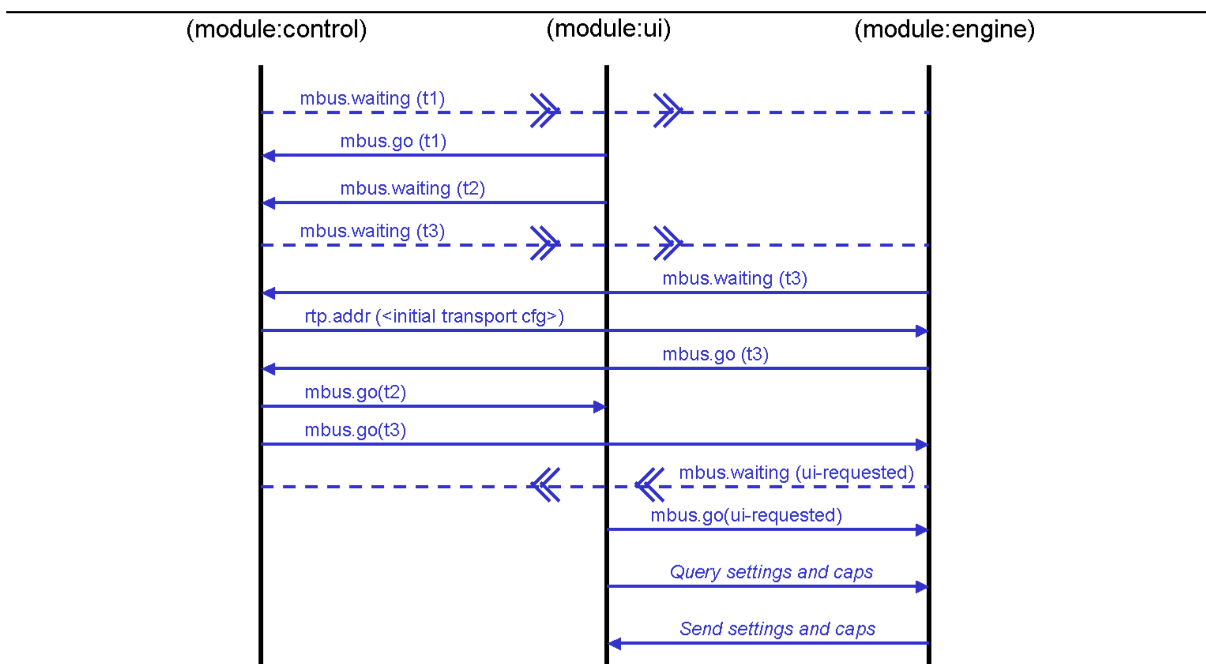

Figure 10.3: RAT initialization communication

Figure 10.4 depicts the RAT communication after the initialization is complete. Essentially,

the user interface can now send a set of control commands, e.g., setting the input and output gain, and the engine can send event notifications and status changes. The control commands from the user interface are sent using reliable transport mode while the messages sent by the engine are sent using best effort mode. The engine's messages are essentially *soft-state updates*, i.e., they are sent periodically, e.g., triggered by external events such as the reception of an RTCP receiver report.

When the user terminates the application by activating a corresponding control element on the GUI, the user interface sends an `ui-detach-request` command to the engine, which is acknowledged if the engine is ready to detach from the user interface. Subsequently, the user interface sends both the controller and the engine the `mbus.quit` command that causes both entities to terminate themselves. The user interface terminates as well and the application is closed.

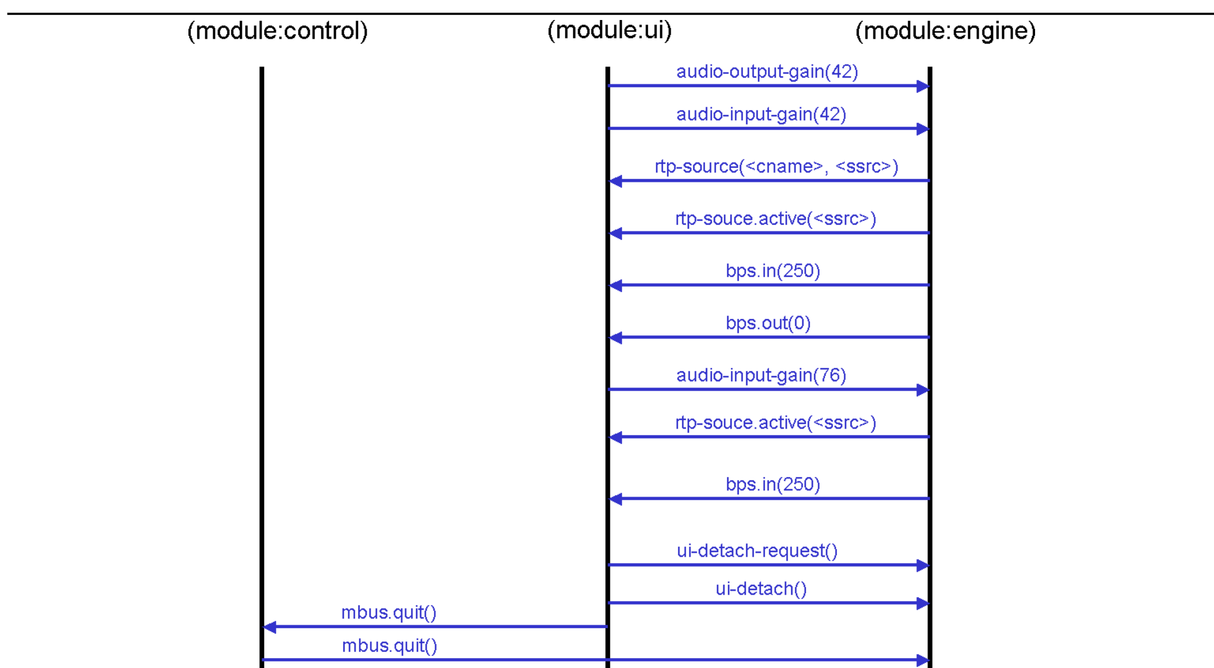

Figure 10.4: RAT runtime communication

It is worth noting how RAT utilizes the Mbus synchronization commands for establishing the application session and for initializing the entities in the correct order. One reason for this procedure is that the controller does not know the Mbus addresses of the other entities in advance (as the `id` element is determined by the protocol implementation). In order to make sure that the right entities are brought together (considering that there might be multiple RAT instances of the same user, all of which use the same Mbus configuration), the controller must initiate the rendezvous by broadcasting its `mbus.waiting` message with the token that it has been set before, when starting the entities.

It is also interesting to note how RAT relies on *soft-state communication* for updating the user interface. The idea is that the information that changes from time to time anyway, such as receiver statistics and activity states does not have to be sent in reliable transport mode but can be sent periodically. The user interface (depending on its implementation) can simply

apply the updates to its internal state management or display the updated information directly, if applicable. RAT's modular approach and the well-defined interface of the engine component allow re-using the audio-engine for other applications, e.g., by employing the engine for an integrated conference application that provides its own GUI. In fact, we have re-used the RAT engine for a H.323 conferencing terminal and as a transcoding engine in a conferencing gateway. In the following section, we describe a SIP endpoint that employs the RAT media engine.

10.1.3 Bonephone

One example for an endpoint application that uses RAT's media engine through its Mbus interface is the SIP endpoint *Bonephone*.[3] Bonephone is a SIP endpoint with a graphical user interface that allows users to conduct SIP audio calls. Users can configure codec preferences, select SIP users from an address list, establish SIP calls and perform minimal configuration functions during a call, such as setting the input and output gain.

The Bonephone application comprises two processes: the RAT media engine and a GUI process that also incorporates the control and SIP functionality. Bonephone uses the NIST SIP stack[4], a Java SIP implementation, and TZI's Java Mbus implementation (Section 7.4) as building blocks that are augmented by a user interface and the corresponding control functionality.

Bonephone's control component uses RAT's Mbus interface to initialize the media engine and to control it during a session. The codec capabilities that Bonephone learns from the Mbus exchanges during the initialization phase are filtered by a user-defined preference configuration and are then used for negotiating conference configurations through SIP. Bonephone provides GUI control elements for setting the input and output gain; user actions are mapped to the corresponding Mbus control commands. The Bonephone application is interesting because it demonstrates the realization of a component based telephony endpoint by relying on readily available components that are "glued" together using the Mbus framework. It is also noteworthy to see that it is possible to develop consistent applications using Mbus modules that are implemented in different programming languages, in this case C and Java. This separation of functionality, where the core functions are implemented in C or C++ and the user interface is implemented in Java or a scripting language, has proved to be a useful design strategy in order to combine run-time efficiency with portability.

There are several other endpoint developments that are based on this model, e.g., in [Meyer01], Meyer has described the more advanced Mbus based H.323 telephony application *Wipone* that is implemented in C++ (using the Mbus C++ implementation described in Section 7.3).

[3]Bonephone has been developed at the Fhg FOKUS Research Institute for Open Communication Systems (`http://www.fokus.gmd.de/`). It is available at `http://www.iptel.org/products/bonephone`.

[4]The NIST SIP stack has been developed by the National Institute of Standards and Technology (`http://www-x.antd.nist.gov/nistnet/`).

10.2 Mbus in Desk Area Environments

The Mbus based endpoints and media engines that we have described in the previous sections are examples of decomposed applications that use the Mbus for coordinating their individual components, i.e., the focus has been on *component re-use* and *local coordination*. In Section 1.3.1, we have presented the idea of integrating multimedia conferencing systems into the computing environment in order to utilize different specialized functions that can be provided by personal devices of a particular user. This concept goes beyond the idea of using Mbus for decomposing traditionally integrated applications on a single host because we extend the scope of a user's application coordination session to a network of devices.

With respect to the Mbus communication, these devices can be viewed as application components, comparable to the endpoint Mbus components we have described so far. However, the dynamic integration of devices into application contexts imposes some other requirements, e.g., the discovery of services and the dynamic establishment of Mbus sessions. Opposed to Mbus sessions for closed applications, these dynamic scenarios cannot rely on a common configuration, i.e., the *bootstrapping* problem has to be addressed. Another task is to develop concepts for extending running Mbus coordination sessions dynamically by integrating new components, e.g., new control components that take over the control of the system.

In the following sections, we describe two projects in which we have pursued the dynamic integration of services into Mbus sessions. We have taken our endpoint decomposition architecture as a basis and have developed solutions for extending the functionality of originally closed endpoint systems, relying on the Mbus protocol and the *Dynamic Device Association* extensions that we have described in Section 6.4 and Section 6.5.

Section 10.2.1 describes the *Desktop Telephony Integration* (DTI) project that focused on the integration of IP telephony systems into a user's computing environment by making them controllable from personal user devices such as PDAs. The *Functional Enhancements using external Telephone Applications* (FETA) project that we describe in Section 10.2.2 extends the DTI work but concentrates on extending the telephony endpoint device itself, e.g., by integrating new application entities dynamically. Section 10.2.3 provides a summary of the main lessons learned during these projects.

10.2.1 Desktop Telephony Integration

The DTI project had two primary goals: On the conceptual side, it was intended to further the development of the Mbus protocol infrastructure as underlying means for local group coordination (among the various components involved) with respect to the specific needs for integrated desk area environments. This includes extending the fundamental Mbus mechanisms for auto-configuration and dynamic device association; defining appropriate Mbus messages for call control and other high-level interactions. As a practical part, the Mbus has been integrated into a real IP telephone set and two demonstrator application scenarios have been implemented based upon the Mbus communications infrastructure.

The first scenario is called *automatic conference creation* and was intended to demonstrate the possibility of using the Mbus to remote-control an IP phone in order to automate the process of inviting multiple users into multi-party conferences.

The second scenario was called *personal presence application*. In this scenario, the phone's functionality has been extended to disseminate status events to one or more interested Mbus

entities that have registered with the phone before. Such information could then be aggregated by presence agents on behalf of the user and could be sent to authorized communication partners using transport protocols for personal presence information.

For both scenarios, we have used a phone control application on a user's PDA for enabling users to locate and to associate to available phones using the DDA protocol and to remote-control a phone over the Mbus. A user may walk up to an "arbitrary" phone and, if permitted by the phone, dynamically associate her PDA with it and use the phone for calling individuals or creating a conference using her conference management application. The identification of the phone is done by means of a human-readable phone name that may e.g. specify a phone's location (such as "North Entrance, Phone booth #3"). If multiple phones are available, the user chooses one of the list and manually initiates the association process as depicted in Figure 10.5.

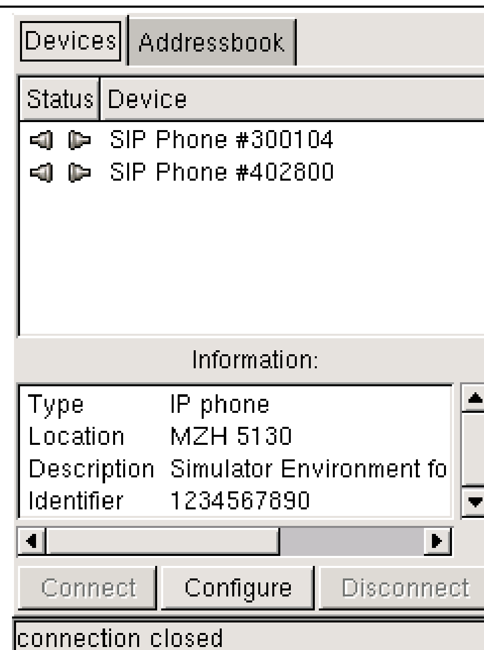

Figure 10.5: PDA GUI for dynamic device association

In Section 10.2.1.1, we describe the fundamental concepts for the DTI projects and in Section 10.2.1.2, we present the conference creation scenario in more detail. In Section 10.2.1.3, we discuss our design decisions for implementing these scenarios.

10.2.1.1 Concepts

The DTI project has been conducted with a manufacturer of IP telephony systems, and the motivation for the project was to enhance the functionality of an IP telephone by seamlessly integrating it into a common Mbus session with additional devices, e.g., a PC with video and application sharing functionality or a user's PDA, each of which can provide a specific useful function within a conferencing session.

Such an environment shall enable users to dynamically combine their available personal and desktop appliances to create their optimal collaboration environment in an ad-hoc fashion.

For example, phone features (wireless handset, headset, convenient look and feel for audio conversations) may be combined with sophisticated functions on powerful PCs (such as word processing, shared whiteboards), with personal databases on PDAs (e.g. address books), with email history and presence status on the laptop computer, etc.

This was the basic motivation for the DTI project: a user's various personal appliances shall automatically become aware of each other and form an integrated desk area environment. The same piece of functionality may be provided by different components (e.g. soft phone vs. IP phone) and the user may decide dynamically which components to use. Unlike the traditional CTI approach to local phone configuration and control, there is no fixed master-slave or client-server relationship, nor is control limited to a single device controlling another. The group of components shall act in concert with only minimal user administration and configuration required (plug and play).

In such a scenario, the vendors can design devices optimized for certain tasks and do not have to worry about providing all-in-one solutions. Instead, they can supply building blocks complementing each other. In particular, an IP telephone does not need to provide a high-resolution color display, a web browser, a Java interpreter, etc. in a desperate attempt to imitate a workstation, albeit at lower quality in many respects.

10.2.1.2 Automated Conference Creation

For multi-party conferences, the manual invitation of participants by a user who calls the participants using his phone can become a tedious and time-consuming task: All participants have to be called and invited to the conference (which in general incurs a significant number of call-transfer, call-hold, etc. operations and thus is prone to errors), addresses that are not in the phone's address book have to be remembered and typed, etc. Conference calls that use a pre-configured conference bridge distribute this load across all participants but introduce latencies at the beginning (while people already on the conference call need to wait for the remaining ones to join), incur a higher administrative overhead to perform the reservation, and, finally, are typically not suitable for ad-hoc sessions. Hence, pre-configured conference calls are considered a different (complementary) scenario and are not discussed further.

In order to automate the setup process, a conference management application external to the phone is introduced to automate the setup process. This application takes control of the phone via Mbus. The application receives a list of conference participants to be called (including their SIP addresses) and calls them one by one. Each participant is — without a specific audio announcement — transferred into the conference. For each participant, the current status is monitored by the conference management application and displayed to the user. This user interface also allows adding further participants. If the conference mixer is implemented within the phone, the conference management application also allows for muting/unmuting and disconnecting individual participants.

An address book outside the telephone is used to maintain names and addresses of the conference participants. As an optional enhancement (which is not being incorporated into the demonstrator), an associated calendar application could be used to determine the feasibility of an ad-hoc conference for the various participants and help finding a timeslot for a conference as soon as possible.

An overview of this demonstration scenario is depicted in Figure 10.6: A PDA of a nomadic user finds three IP phones out of which the user selects `phone #3` as device to dynamically

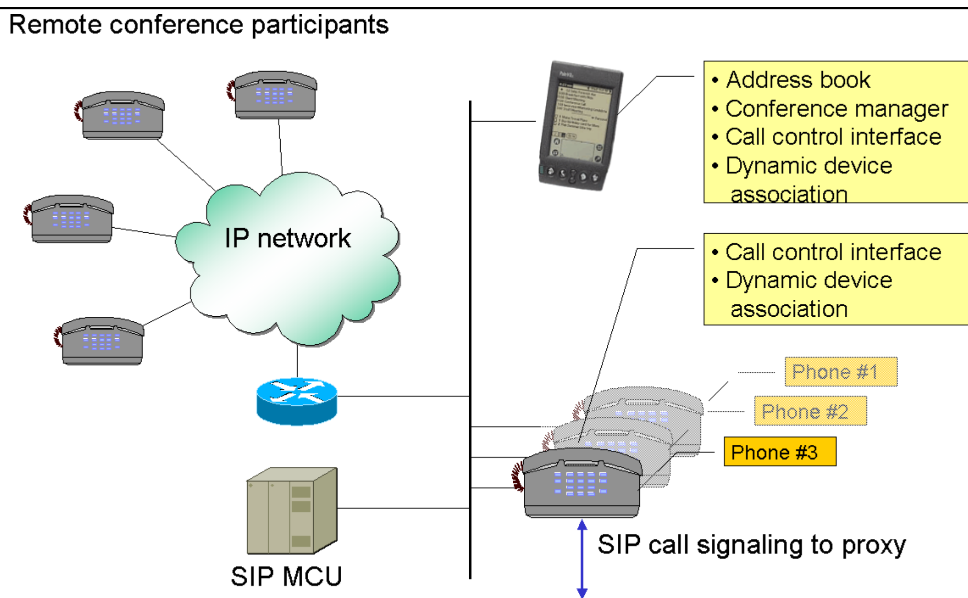

Figure 10.6: Automated conference setup

associate with. From her address book on the PDA, the user selects four remote participants she wants to include in the conference. The PDA uses Mbus call control functions (Section 9.3) to have `phone #3` initiate a call to the conference bridge and to invoke calls to these participants and transfer them to the SIP MCU (see Section 2.1 for a description of the term MCU), using a fixed SIP URI as conference identifier.

10.2.1.3 Implementation

For implementing these scenarios, we had to consider two main systems: the IP telephone and the PDA as the user device. We have been provided with a simulation environment for the IP telephone software that relied on the same software as the production system but could be developed and operated on PC platforms. An investigation of the software architecture has largely confirmed our general conferencing architecture considerations that we have described in Section 3.1.1: The phone software consisted of more or less individual modules for call control, user interface and overall coordination of the whole system. In this case, the call control and RTP module have been third-party components and the user interface and coordination module have been used both for SIP and for H.323 variants of the phone system.

Indeed the decoupling of the different phone modules has been accomplished by relying on interprocess communication mechanisms (UDP communication for the simulator environment and a message queue mechanism for the production system). This clean separation of modules made it comparatively easy to integrate the Mbus functionality into the phone software. We have ported the Mbus C++ implementation (Section 7.3) to the phone platform and integrated it into the phone application software. This included an integration of the DDA server functionality and an implementation of the Mbus Call Control Profile. We have mapped the Mbus Call Control Profile messages to functions and event handlers of the telephone software and have thus made the phone fully controllable by outside controllers, such as the PDA system.

The PDA system has been implemented using the Mbus Python implementation (Section 7.5) and provided the corresponding DDA client functionality. For the development of the GUI based PDA software, the Mbus Python implementation has proved to be appropriate because it is a native implementation, i.e., does not depend on additional functionality except for a Python implementation, and because it is easy to combine it with existing Python GUI toolkit bindings. In this case, we have implemented the software on a Compaq Ipaq running Linux, using the Python GTK binding.

For the conference creation scenario, the PDA has controlled the phone through the Mbus Call Control commands to initiate a SIP call to each participant. After the call has been established, the PDA has made the phone transfer the participant to the conference bridge by sending the `conf.call-control-transfer` command. Figure 10.7 depicts the transferring of one participant to the conference bridge. After the PDA has transfered all participants it establishes a direct call from the phone to the conference bridge and the conference establishment is finished.

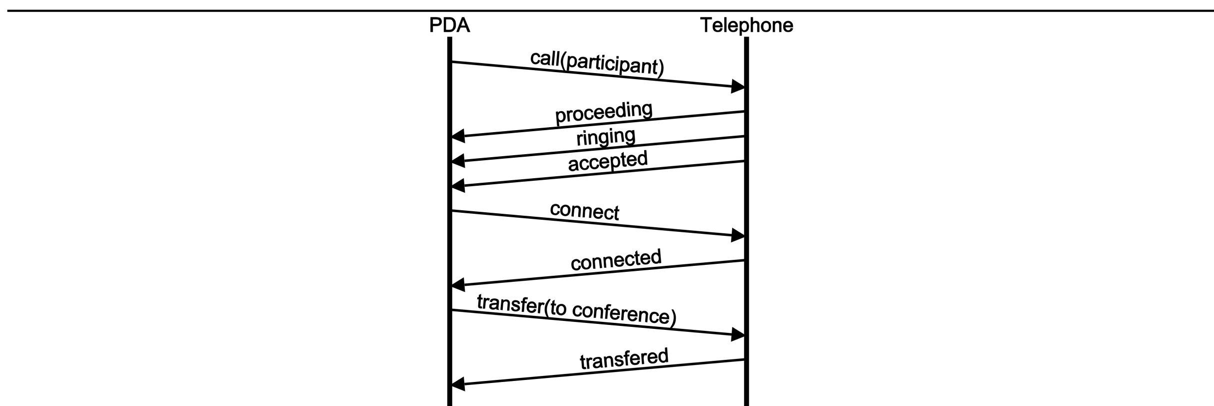

Figure 10.7: Mbus Call Control for conference creation

10.2.2 Functional Enhancements using external Telephone Applications

The DTI project has created the foundation for the development of Mbus based applications that are composed of multiple services and devices in a user's computing environment. While the DTI project has concentrated on creating an Mbus enhanced telephony system that can be dynamically associated through DDA and be controlled by a user's PDA, the project *Functional Enhancements using external Telephone Applications* (FETA) has focused on investigating the extension of a phone's feature set by adding external components, such as new application components, into its environment.

The basic idea is that the phone itself provides the basic telephony service but can locate and utilize additional components that could be installed on a user's PC. The components could be provided by the vendor (as optional supplements to the product) or they could be provided as *plug-ins* from third parties (or the user himself). Two applications have been developed by the FETA project:

- the integration of multimedia clients on a user's workstation; and

- the integration of conference bridges on a user's workstation.

We describe the integration of multimedia clients in Section 10.2.2.1, the design of the endpoint-based conferencing system in Section 10.2.2.2 and present some implementation considerations in Section 10.2.2.3.

10.2.2.1 Integration of Multimedia Clients on a Workstation

In the FETA project we have chosen a video application entity as an example for a multimedia application that is integrated into a conferencing system. The usage scenario can be described as follows:

A user has multiple devices in her computing environment, including an IP telephone, a workstation and a PDA. The IP telephone, in this case a SIP phone, is Mbus and DDA enabled and can communicate with other Mbus entities in its environment. The phone vendor distributes the phone with accompanying software: application entities that can run on a user's workstation, in this case a video application that is also Mbus enabled. The video application can be integrated into the phone's conferencing system, as an additional, external application entity. The integration can either be initiated by the phone itself, i.e., automatically, or it can be initiated by the user from its PDA. The discovery and integration of the video application relies on the advanced DDA mechanisms as described in Section 6.5. Different from the DTI scenario, where we have associated a PDA to a phone by requesting the Mbus configuration from the phone through DDA, there is no single client-server relationship. For the FETA project, we assume that the phone remains the center of a user's Mbus environment, and for integrating the video application into the conferencing system, the phone system *transfers* an existing configuration to the application entity — by the means of a DDA invitation request.

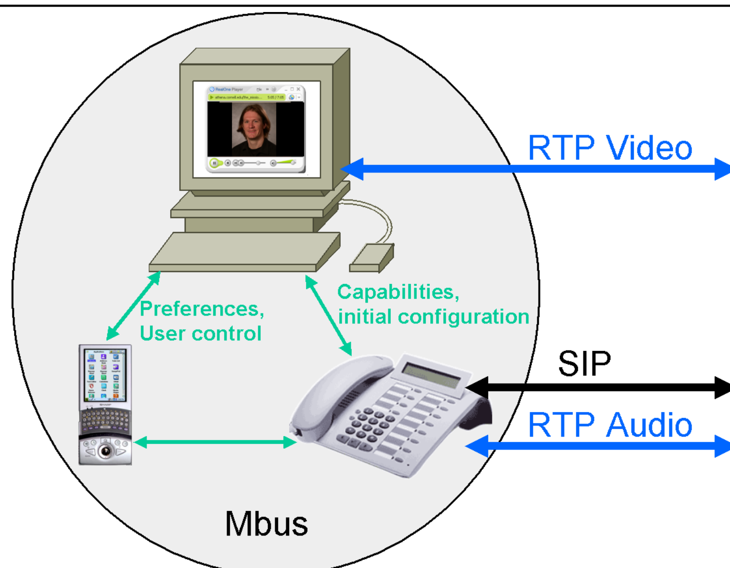

Figure 10.8: FETA application integration scenario

Figure 10.8 depicts a scenario with an already established Mbus session. After the video application has joined the Mbus session, the phone can start the initialization and configuration

process as conceptually described in Section 9.1.3, i.e., the phone queries the video application's capabilities and performs the initial configuration of the application entity.

When new calls are initiated, the phone call control engine can offer its video capabilities to other endpoints and negotiate a configuration for a multimedia conference, i.e., including audio and video conferencing. Using regular Mbus configuration commands, similar to the RAT based endpoints described in Section 10.1, the video application is coordinated, e.g., to start transmitting upon the start of the conference. Logically, the communication relationship between the endpoint controller (on the phone) and the video application (on the user's workstation) does not differ from the local endpoint scenario. While the phone is enabled to integrate the video application automatically, it may not be able to offer the user the full set of control possibilities on its limited user interface. For this purpose the user can additionally configure the video application through its native GUI on her workstation or employ her PDA as a control device to configure the application, e.g., setting the frame rate and other codec parameters.

10.2.2.2 Endpoint-based Conferencing

For the endpoint-based conferencing scenario, we extend the capabilities of the phone system in a different way as described for the video application scenario. The conferencing capability cannot be characterized as an application entity such as a video engine, but can rather be viewed as a special purpose service that influences the call management logic on the phones (if integrated and deployed). The scenario can be described as follows:

Similar to the video scenario, the conferencing service can run on a user's workstation, as an optional extension to the phone's feature set. Similar to the video application it is a Mbus/DDA entity that is to be integrated into an existing Mbus session and to be controlled through Mbus communication. A dedicated conferencing entity provides conferencing services to the user (and to other parties that are invited to a conference by the user).

The conferencing entity is essentially a *conference bridge*, i.e., an RTP based media processing system, where for audio streams, the processing is limited to mixing streams.[5] For larger conferences, the mixing of many streams can be resource-demanding and can not be provided by the usually limited hardware (and software) platforms that are used for telephone systems. Therefore, the user can run a conference bridge on her workstation and have its phone take advantage of the additional service, thus being able to host multi-party conferences without requiring external conference services.

We have designed the following architecture for integrating this service: in addition to audio mixing, the conference bridge implements a full SIP UA and thus terminates the SIP signaling to each participant.[6] The conference creation works similar to the approach chosen in DTI: the IP phone transfers all conference participants to the conference bridge which then takes control. Figure 10.9 depicts this architecture. We can see the phone, the PDA (for controlling

[5]For video, other operations such as switching would be conceivable, but we restricted the scope to audio for this project.

[6]There are different design alternatives for conference bridges. In general, there are two variants: conference bridges that perform media processing (e.g., audio mixing) only, and conference bridges that appear as a complete user agent, incorporating a SIP user agent *and* the media processing functionality. While in the second case, all conference participants can join the conference by simply establishing a call to the conference bridge, the first case requires an external entity to terminate the SIP call signaling and to manage the conference.

the phone) and the conference bridge as entities of a user's Mbus based desk area environment. For all conference participants (including the local user), the conference bridge manages both SIP call signaling and RTP media sessions.

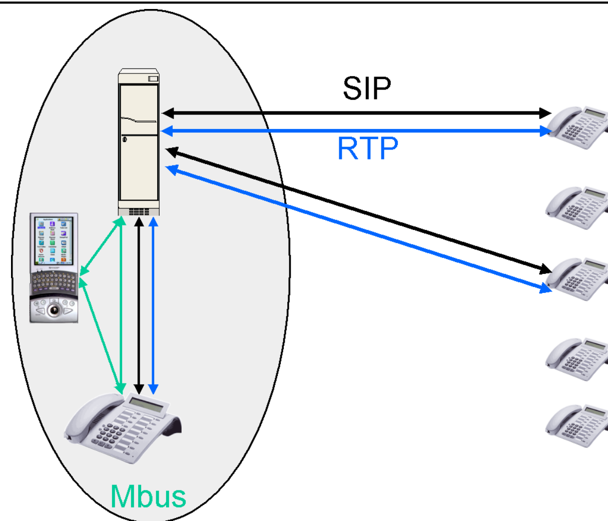

Figure 10.9: FETA endpoint-based conferencing scenario

This approach is attractive, because it does not impose any requirements for the phone with respect to managing conference state. Instead, the phone would simply transfer all selected participants to the conference bridge and then join the conference by establishing a new direct call to the conference bridge. Some conference control communication, such as floor control and management of membership can still be realized by an Mbus-based control interface that is employed by the phone.

The Mbus interface of the conference bridge provides functions for creating conferences, for querying transport parameters and managing the membership of the conference. In addition, the conference bridge can generate event notifications such as RTP events, e.g., when a member joined or when a member starts a talk spurt. Due to the limited user interface capabilities of our specific phone system, we have decided to delegate the control of the conference bridge to the PDA (or another computer in the desk area environment).

The PDA control application registers as an Mbus controller with the conferences bridge and controls it through Mbus Guidelines mechanisms, i.e., Mbus RPCs and event notifications. For establishing the conference, the PDA software creates an Mbus context and sends the `conf.setup` RPC for creating a conference and for specifying the SIP users that are going to be invited and shall be admitted to the conference. The conference returns the complete conference URI in the RPC return command. This conference URI is used as a destination for the call transfer to the conference bridge, which the user's phone initiates, i.e., the user's phone requests the called parties to establish a call with the conference bridge, and the conference bridge is denoted by the conference URI.

The PDA control application can now start to bring the selected users into the conference by performing the same operations as described in Section 10.2.1.2 for the automated conference creation. However, different to the DTI scenario, the PDA application will not receive event

notifications about the joining of all participants from the conference bridge and can use this information to inform the user. During the conference, the PDA application can request the conference bridge to add or remove users by sending dedicated Mbus RPCs.

10.2.2.3 Implementation

For implementing these two scenarios, we have extended the existing Mbus and DDA implementations on the phone and the PDA and have developed a simple Mbus/DDA enabled video application and an Mbus/DDA enabled conference bridge.[7]

The video application scenario required some deviations from the original concept: different to the DTI conference creation scenario, the video integration scenario requires a more intrusive manipulation of the call control handling on the phone because the capabilities and configuration settings of the video application have to be considered for the call setup process that is conducted by the phone call control engine. The call control engine must negotiate a suitable configuration and configure the video application accordingly after the call has been established.

However, in our specific phone system, the call control engine was really a third-party component in the strictest sense: it was integrated as a software library and could only be controlled through a well-defined API. The API did not allow incorporating any other media types into a conference and did not allow for accessing the SDP description directly. Instead the call control engine was coupled to the RTP audio engine (both have been developed by the same vendor) and performed the control and configuration of the audio engine itself.

The integration of an updated call control engine was not feasible in the FETA project, and therefore we have designed a workaround that allows the phone to create multimedia conferences without requiring intrusive changes: A personal Mbus based SIP intermediary system (a *back-to-back user agent*, B2BUA), performs the negotiation and establishment of the multimedia conference on behalf of the phone. The phone uses the B2UA as an outbound SIP proxy, which allows the integration of the video application to be done transparently to the phone. The video application is therefore integrated into the B2BUA application, not into the phone itself. However, the conceptual considerations with respect to querying capabilities, configuring the video application appropriately and coordinating it during the conference still remain valid. We cannot discuss the B2BUA solution here in much detail, but the application is derived from the multimedia conferencing gateway that we describe in Section 10.3.

10.2.3 Lessons Learned

The DTI and FETA projects represent milestones in our development of Mbus-based architectures for conferencing applications. Building on our experiences from initial conferencing projects such as CONTRABAND, we have been able to validate our concepts and demonstrate their applicability for product-level systems that are developed for specialized hardware platforms.

The first result of these projects was an analysis of the design of a specific product-level telephony system. This analysis has largely confirmed our use case scenario description in Section

[7] The conference bridge is an extension of an Mbus-based multimedia conferencing gateway that we describe in Section 10.3.

3.1.1 and our system model that we have described in Section 9.1.3: The internal architecture of the phone system provides the same degree of modularization as we have expected and the design of the existing communication mechanisms for coordinating the modules relied on well-defined interfaces and even exhibited a rudimentary form of message oriented coordination.

Hence, the second result was the integration of our Mbus code into the system, which provided no significant difficulties at all. We have chosen our Mbus C++ implementation as the phone software itself is implemented in C++. Based on the Mbus transport and Guidelines implementations we added an Mbus Call Control interface to the phone, which we have been able to demonstrate successfully for the conference creation application.

The third result was the design and implementation of the Dynamic Device Association (DDA) concept, which extended the applicability of the protocol Mbus from statically configured environments such as single workstations to federations of devices and services that belong a user's environment but are less tightly coupled than application entities on a single workstation.

The implementation of the video integration mechanisms for the FETA project has shown that extending an endpoint's media capabilities is per se feasible but requires more intrusive changes to the application logic and call control process than the addition of a remotely controllable call control interface. For the implementation of the Mbus Call Control concepts on the phone, we have been able to rely on existing CTI interfaces, which has made the extension of the phone comparatively easy. However, the integration of new application entities goes beyond the scope of CTI. We have learned that the necessary changes to the details of the call management, e.g., access to the SDP description exchange and to the setup of media sessions, are not anticipated by developers of standard SIP and RTP stacks for IP phones. The required changes to the call control module would not have been very intrusive but were not feasible in the FETA project, because the different components were not adaptable and not available as source code modules.

Although we have been able to implement the desired functionality with a different design, we have certainly learned something about interface design of call control engines for conferencing systems: A call control engine should enable its clients to take control of the call setup and configuration negotiation phase, e.g., by making the session description accessible, in order to allow for flexibly extending the system's functionality.

Another lesson we have learned is more subtle: the general idea for the FETA work was to develop an architecture that allows IP telephony vendors and their customers to extend the feature set of otherwise limited devices. By providing users with additional components that can run on a user's workstation and be integrated into the conferencing system, these devices become *extendible* according to users' needs.

The FETA project has deliberately focused on studying the fundamental mechanisms for enabling the feature set extension, and chosen two specific extensions: the video application and the conference bridge. For these extensions, we have defined appropriate Mbus interfaces and extended the phone's and the PDA's GUI and application logic to take advantage of the additional components. Except for the mentioned implementation difficulties this was relatively easy to accomplish as we have programmed the application to fit our own interfaces.

When we *generalize* the concept of extending conferencing systems, we can conceive scenarios where users extend the feature set of their IP telephone by a self-developed component that was not anticipated by the original developer. One interesting question in this context is, in

which ways do we have to enhance the current Mbus/DDA framework in order to accomplish this dynamic extension?

For the FETA project, we have studied two different types of extensions: a media application extension and the conferencing extension. For the media application entity, we can refer to our generic endpoint architecture that we have described in Section 9.1: by relying on a generic capability and description mechanism and on well-defined interfaces for media application components we can generalize most of the coordination task between a controller and an application entity. For example, in the FETA scenario, the phone controller would determine the type of the entity (video engine in this case) by its Mbus address (and/or by its DDA service description) and could use generic Mbus commands for media engines to query its capabilities and initial configuration and to perform generic coordination tasks during a conference.

However, already the video component might provide specific configuration parameters that require manual configuration, e.g., the frame rate or quality of the video coding. The user configuration must be performed through an appropriate user interface. This is even more an issue for applications that do not fall under the standardized media engine category, such as the conferencing application. One reason why we implemented the conference bridge configuration on the PDA and not on the phone was because the phone's user interface is limited and not easy to extend. However, for a new design of a corresponding system, it would be very desirable to be able to use the conference bridge without an additional control system such as the PDA and to configure it directly through the phone's user interface.

Providing such user interfaces for configuration without knowing the application in advance requires the dynamic generation of these user interfaces according to a specification that is obtained from the application itself. In Section 4.4, we have analyzed how typical component frameworks address this issue: components provide interfaces that consist of standardized interaction types, usually RPC-like commands, properties and event notification. Interface definitions are specified in terms of these interaction types and can be queried and evaluated at run-time from applications that want to use a component's service.

A similar approach is conceivable for developing a truly generic service extension framework that generalizes the concepts we have applied to the FETA system. Of course, such a framework would be applicable beyond the application domain of IP telephony, respectively multimedia conferencing: In Section 10.4, we describe the design of an Mbus-based architecture for home automation, where similar concepts are applied for enabling users to integrate arbitrary components into the system at any time and control them using interfaces that they learn of from dynamically obtained interface descriptions.

10.3 Gateways

In addition to the previously described application of Mbus to endpoint coordination and endpoint extension, we have also developed Mbus based architectures for multimedia conferencing gateways. In the broadest sense, the term *conferencing gateway* refers to intermediary systems (i.e., not to terminals) that can provide useful functions for a multimedia conference by processing the control and/or the media communication in some way. For example, a SIP/H.323 *signaling gateway* performs call signaling translation between (otherwise non-interoperable) SIP and H.323 terminals. This is possible due to the high degree of semantic congruency between the call signaling and call control functions of both protocols, as we have already noted

in Section 9.1.4. A pure signaling gateway would not process the media streams in any way; instead they would flow directly between the terminals and the call signaling gateway would only translate the call setup communication that is required to establish a usable configuration of the media sessions.

A *media gateway* in contrast can be used to perform specific operations on the media streams, i.e., it would terminate and originate media streams and apply a certain function, e.g., media transcoding. These two different functions, signaling gateway services and media transcoding, are not completely orthogonal, because for many applications such as IP telephony, some form of signaling is required to configure the use of media gateways. For this reason, multimedia conferencing gateways for these application domains potentially provide both functions.

We have developed different gateways architectures and specific implementations in different research projects, starting from monolithic H.323-Mbone gateways to flexible signaling and transcoding gateways that can be adapted to different requirements. We cannot describe all these developments here in detail but will focus on the latest system, concentrating on the Mbus related aspects. This gateway is a SIP media transcoding and IPv4/IPv6 translation gateway that we have developed in the European research project 6WINIT.

In Section 10.3.1, we provide a brief overview of the project itself, and in Section 10.3.2, we present the gateway architecture, and in Section 10.3.3, we discuss some implementation aspects. In Section 10.3.4, we summarize the lessons we have learned during our gateway related research activities.

10.3.1 IPv6 Wireless Internet Initiative

The European *IPv6 Wireless Internet Initiative* (6WINIT)[8] has validated the introduction of the Mobile Wireless Internet in Europe. 6WINIT has investigated the setup of native IPv6 wireless networks employing different network technologies such as IEEE 802.11 WLAN and 3G technologies. The goal of the project was to demonstrate the applicability of these technologies for real-world applications, and the project has developed network architectures and applications for the *health care* domain that addressed requirements such as mobility, security and quality of service.

One of the applications in this context was multimedia conferencing. In the 6WINIT project, we have developed an Mbus-based gateway system, the *TZI-Gateway*, a signaling and media gateway for both SIP and H.323 with support for IPv4 and IPv6. It supports IPv4/IPv6 interworking by providing application layer translation and media forwarding. More general information on the 6WINIT project is available at the project's home page[9], and details on the gateway and possible deployment scenarios are provided by [6WINIT02].

10.3.2 Gateway Architecture

The previous discussion of scenarios and requirements for interoperability services has shown that there are in fact different possible scenarios, each of which provides different requirements

[8] http://www.6winit.org/

[9] `http://www.6winit.org/`

for the gateway services that need to be provided. For example, an IPv6 only SIP-H.323 gateway obviously has to provide the call signaling translation service but does typically not have to process media streams at all. A pure IPv4/IPv6 interoperability service for SIP based IP telephony must translate both signaling and multimedia communication with respect to the used IP version but does not have to perform call signaling protocol translation. Then there are different deployment scenarios with respect to the topology of the gateway network, which we have described in more detail in [6WINIT02].

We have addressed these requirements by developing an Mbus based gateway framework that allows for instantiating different specific systems by composing them from different standardized components. In [Pollem00], Pollem describes the design of a specific gateway system that we had developed during a previous project: the multi-protocol gateway *Stargate*. Stargate already provides an Mbus based architecture and has been designed as a signaling gateway between H.323, ISDN and SIP. The gateway provided an optional audio transcoding function that has been realized by integrating the RAT audio engine (see Section 10.1.2).

For the TZI-Gateway that has been developed within 6WINIT, we have extended and generalized the Mbus based architecture and have advanced the media processing function by developing an extensible Mbus based media processing system. Figure 10.10 depicts the architecture of the Mbus-based TZI-Gateway.

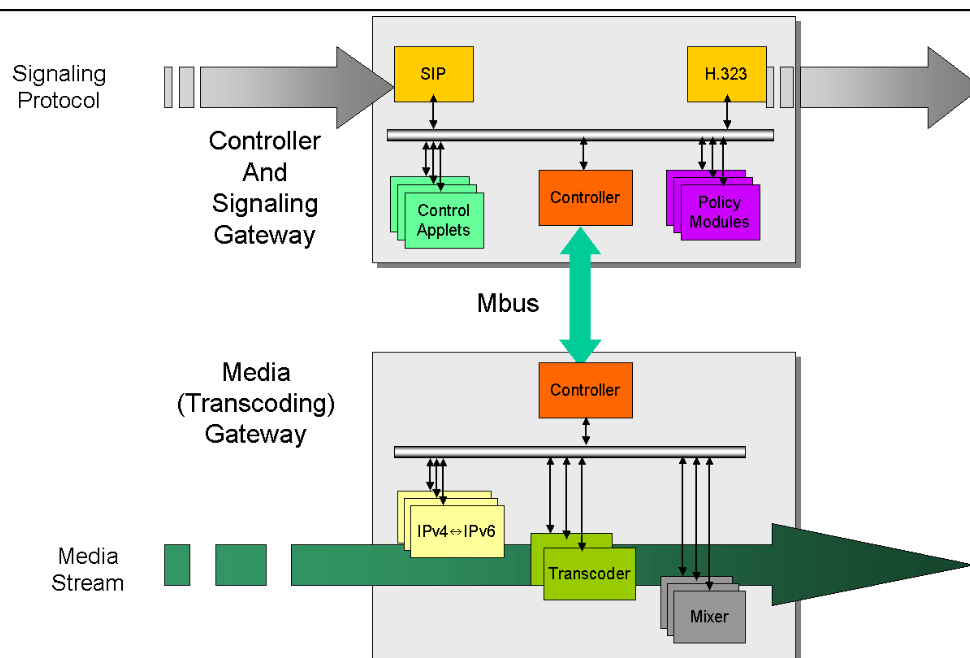

Figure 10.10: System architecture of the Mbus-based TZI-Gateway

The architecture is constituted of two main elements:

- A signaling/control entity is responsible for basic signaling services (including signaling protocol gatewaying) and for control services. Depending on the specific type of the application, this control entity governs a media data forwarding/transforming element.

  The signaling component is an Mbus based application in itself, where the components employ the Mbus Call Control Profile and other Mbus interfaces for coordination.

- The second element (the media processor) can provide different media stream transformation and media gatewaying features, e.g. IPv4-to-IPv6 (RTP/RTCP) translation, mixing of media streams, transcoding of media streams, robustness functions (redundancy encoding, forward error correction, interleaving). The media processor provides a flexible *plug-in architecture* that allows the composition of gatewaying and media transformation systems from a set of re-usable modules and allows for independent development of additional modules.

  The *plug-in architecture* defines interfaces that media processing modules have to implement. The different media handling modules in a gatewaying/transcoding application are co-ordinated by a controlling module that is responsible for instantiating media handling entities for specific streams. Different types of media handling modules exist: there are originating, translating and terminating modules.

As depicted in Figure 10.10, the controller and signaling gateway uses Mbus to control the media processor (that provide an Mbus Guidelines conforming interface).

10.3.3 Implementation

The functions of the signaling and the media processing components are largely orthogonal. In the following, we describe the implementation of the Mbus based signaling engine. For details the Mbus based media processor, we refer to Büsching's description in [Buesching01].

The signaling gateway and control component is based on our generic gateway architecture that we have described in Section 9.1.4 but extends this architecture by a more fine-grained structure of Mbus modules. In the following, we describe the signaling component with a SIP-based instantiation of the system as an application example. Figure 10.11 depicts this architecture for a SIP-only configuration that would be used for providing the signaling functionality for a SIP gateway providing IPv4/IPv6 translation and media transcoding services, where the media transcoding entity is not shown in the figure.

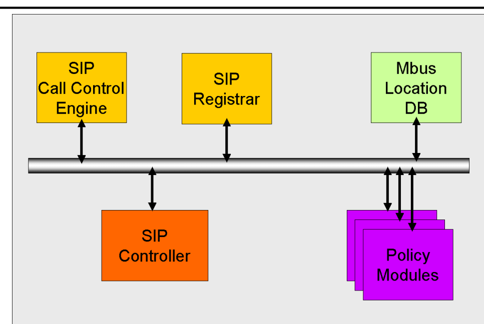

Figure 10.11: SIP-based signaling component

The *SIP protocol engine* is a SIP call control engine that implements the Mbus Call Control Profile, i.e., it generates event notifications on the reception of call-state change indications and can accept RPCs for initiating and controlling SIP transactions.[10] The call control engine

[10]The SIP call control engine has been implemented by Olaf Bergmann and Eilert Brinkmann.

relies on the *tight control model* of the Mbus Guidelines, i.e., it waits for a controller to register. However it can nevertheless be configured to accept SIP requests without a controller. In this case, it will send event notifications on call control events such as `incoming-call` to a well-defined Mbus group address, and it will redirect these notifications to the controller component as soon as it registers with the call control engine.

SIP `REGISTER` requests are forwarded, via SIP, to a dedicated SIP registrar entity (we have described different SIP services in Section 2.2.1.2). The SIP registrar entity processes these requests and can map registration requests to Mbus commands that are sent to a dedicated Mbus-based location database. This entity can maintain address-to-URI mappings in a call control protocol independent way and provides a simple Mbus interface for creating mappings and for performing address lookups. The rationale for the separation of the SIP call control engine and the SIP registrar is again component re-use: the registrar is an optional component that is not necessarily required by a call control engine, e.g., a user agent implementation would not support the `REGISTER` request. For our specific 6WINIT instantiation, we have included a registrar component, because the gateway system is intended to provide registrar as well as gatewaying functions in mixed IPv4/IPv6 environments.

The SIP controller controls the call control engine through Mbus Call Control communication, i.e., it registers as an Mbus controller, receives event notifications and sends Mbus RPCs. The controller is the entity that decides how to process incoming-calls, e.g., by determining whether a call needs to be relayed between an IPv4 and an IPv6 endpoint or whether transcoding functions need to be applied. For relaying calls, the gateway acts as a SIP *back-to-back user agent* (B2BUA), i.e., for any call, it terminates an incoming call and establishes a new outgoing call that is addressed to the desired contact address. For both parties, the gateway appears to be the user agent that terminates the call. In principle, the controller acts as a controller in a multi-protocol gateway, however, in this SIP-only instantiation, the SIP call control engine processes both calls at the same time.

The decision making how to process incoming calls can be processed by the use of *policy modules* that are also depicted in Figure 10.11. For example, we have developed an Mbus-based implementation of an CPL (*Call Processing Language*, [RFC2824]) interpreter.[11] CPL is a call control protocol independent, rules-based definition languages that allows users, vendors, and operators to specify the behavior of a call processing system such as a SIP proxy. For example, a user could upload a CPL script to a proxy that specifies how the proxy should handle incoming-calls for the user, with respect to caller address, date or time of day, and other criteria. A similar approach can be pursued in a gateway, in particular the decision whether to apply media processing such as transcoding could be made (initially) by a corresponding policy module. In the current version of the TZI-Gateway this decision is however made by the controller.

For developing controllers for Mbus Call Control Profile applications, we have designed a generic framework for user agent and gateway controllers: a *finite state machine* that manages the different state transition that a single call can go through. Formally, the state machine mechanism implements a *deterministic finite automaton*, i.e., for each state there is at most one transition for each possible input. Essentially, there is a fixed number of possible call states such as `calling`, `proceeding` and `connected`, and the state transitions are triggered by receiving Mbus Call Control events.

[11] The CPL Mbus component has been implemented by Anja Prelle.

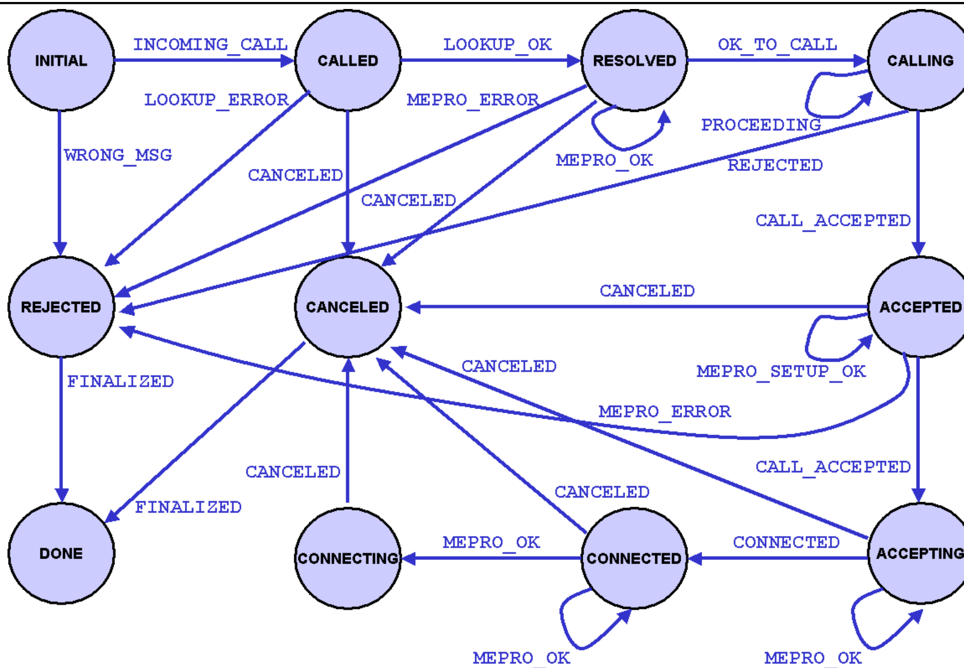

Figure 10.12: Incoming call state diagram

There are different state machines for incoming and for outgoing calls. Figure 10.12 depicts the call state diagram for an incoming call in the specialized gateway controller. The state transitions are partly directly inferred from received Mbus Call Control events such as `canceled`. We can see that, in accordance with the Call Control Profile, the call can be canceled any time, e.g., when a `cancel` event is received. We can also see that the number of states is not very high, however, some states provide four different possible transitions. In this special media gateway setup, there are sometimes transitions that lead to the original state, e.g., for the `RESOLVED` state, which is caused by additional an transition for communicating with the media processor.

A manual implementation of this directed graph into functions and switch statements would be too error-prone and would particularly impede the re-use and adaptability of controller implementations. For this reason, we have generalized the state machine management and designed a generic finite state machine as a C++ template class. The fundamental idea is that a programmer specifies the different possible states and declares the state transitions for each state, rather than programming this logic manually with low-level language constructs. A state transition declaration also involves the specification of a transition predicate — a function that is parameterized with the current state and a program specific input parameter. The function essentially checks whether the state transition is possible, taking the input parameter into account. For example, in the gateway controller, the predicate input parameter is essentially the latest received Mbus Call Control message.

We have conceived the fundamental design of managing the processing of Mbus Call Control communication with this state machine approach for the *StarGate* gateway, our first Mbus-based gateway architecture. For the TZI-Gateway, we have adopted the fundamental state machine abstraction and extended the controller for accommodating the requirements of control-

ling additional components, such as the media processor.

10.3.4 Lessons Learned

The gateway architecture and the corresponding implementation that we have described in this section depict another use case for our generic controller concept and the Mbus Call Control Profile. We have seen that the complexity of gateway systems can be significantly higher than that of endpoints: not only can a typical gateway system provide more Mbus entities (multiple call control engines, policy modules, media processing entities), but there is also a different usage of the Call Control commands. In particular, a gateway will typically create and manage state for two calls, one for each leg of the call. For a single call control engine scenario as we have described for the SIP-gateway, this means that not only the controller but also the call control engine must be able to distinguish and manage two (or more) calls at the same time. In fact, considering gateway systems as usage scenarios for the Call Control Profile right from the beginning has helped to improve its generality and its applicability to different call control protocols.

The *finite state machine* approach has helped us to make controllers for such applications more manageable and easier to adapt and re-use in other applications. For single-threaded applications, the robust management of the different asynchronous events is a major task, e.g., the controller in the TZI-Gateway has to maintain state for at least two calls and the corresponding Mepro terminations. In addition, there may be policy modules that influence the decision making process. It should be noted that all message exchanges are completely *asynchronous*, i.e., essentially every interaction that involves two or more messages requires some form of state machine, e.g., for correlating responses to requests. For distributed state management, which is the fundamental model of the Call Control commands, a formalism as the state machine approach has proven to be quite appropriate.

Recent experiences with the development of gateways has also led to some ideas for further advancements: in multi-protocol signaling gateways, e.g., SIP-H.323 gateways, we have identified a need to *update* the conference configuration at arbitrary times during a call. This has initially been motivated by the H.323 concept of opening and configuring *logical channels* at arbitrary times, but appears to be useful in general. For example, SIP also provides an `UPDATE` request. Although this request is not frequently deployed for simple voice calls, we have conceived some interesting scenarios of this kind, e.g., in the light of dynamically adding application entities to a conference, as we describe in Section 10.2.2.

Furthermore, we still have to explore the dynamic assessment of Mepro capabilities (through the abstractions that Mepro provides). In the current gateway version, the controller has specific knowledge of Mepro's capabilities and uses this information for negotiating conference configurations and for deciding whether to use transcoding. For a more generic approach, it would be desirable to obtain Mepro's capabilities dynamically, to maintain a list of possible transcoding configurations and to consider this information for negotiating conference configurations. One sample application where this is required, is the deployment of a modified gateway instance as a conference bridge, again as described for the FETA project.

10.4 Mbus for Device Coordination

Mbus-based endpoints and Mbus-based gateways can be characterized as rather tightly coupled, locally distributed applications. Despite our concept of generic controllers and dynamically discovered application entities, corresponding applications still rely on a common understanding of supported Mbus commands such as the Call Control Profile that each call control engine is required to support. Even the DDA-based association of dynamically discovered devices still relies on common vocabulary and some form of standardized interfaces.

We have also applied Mbus to more loosely coupled scenarios, where entities need to communicate in some form, however without relying on common interfaces. The Hausgeist project [Hausgeist02] was a two year student project at the University of Bremen that has developed Mbus-based architectures, protocols and demonstration applications for home automation, i.e., scenarios where multiple devices attach to a network and perform some useful functions for a user through some form of coordination.

The Hausgeist project has investigated the application of the Mbus protocol to a rather unusual application domain (compared to the endpoint and gateway coordination domain that Mbus has originally been designed for). The main proposition for the project was to validate the concept of soft-state-based group communication for home automation, thus deliberately deviating from existing field-bus-oriented technologies such as EIB[12], LonWorks[13] or even CAN (see Section 3.1.2). Hausgeist's objective was to develop an Internet technologies-based architecture that can be deployed with different link layer technologies, such as Ethernet and WLAN that are becoming more and more pervasive in today's homes.

Deploying Mbus for this application domain has been useful for validating the fundamental protocol design, e.g., the scalability considerations for the awareness mechanism as described in Section 6.2.2. While it is not a huge difference (with respect to processing costs) to have two or five entities, it makes a significant difference to establish Mbus sessions with 100 or more entities. The Hausgeist project has periodically performed interoperability events, where a large number of entities have been run simultaneously. The first main result of these activities was an improvement of the involved Mbus implementations (with respect to robustness and correctness of the timer calculation for `mbus.hello` messages). The second main result was the experience that larger Mbus sessions are viable. However we have also learned that for certain scenarios, e.g., when a large number of entities run on an overloaded host, packet loss and delays in message delivery will impede the adaptation of the `mbus.hello` sending rate, as a single entity does simply not "see" all other entities. In this scenario, many entities, not knowing the actual number of entities, send too frequently, which increases the system load and the loss rate further. Consequently we are currently considering improvements that allow for a faster and more robust convergence in such scenarios, which we discuss in Section 11.4.

For applications that are not as tightly coupled as desktop applications such as a conferencing endpoints, there has to be some form of service discovery and service description. The Hausgeist approach provides some interesting similarities to the component framework approach and to the coordination and ad-hoc communication ideas that we have discussed in Chapter 4, *Foundations and Related Work*: interface discovery relies on a small set of manda-

[12]European Installation Bus (`http://www.eiba.com/`)

[13]`http://www.echelon.com/`

tory commands for querying interfaces and for establishing the initial communication based on a interface description system, where entities can describe their interfaces in terms of standardized interaction types (method invocation, properties and event notification). Hausgeist has taken the service discovery idea a step further and has provided SLP-like (see Section 4.3.2.1) services for locating services depending on certain properties they might provide.

We have learned that *entity association* is an important element for establishing sessions ad-hoc, e.g., when integrating a new device into a home network. While the Hausgeist project has concentrated on the bootstrapping and configurations aspects, we have shown in our discussion of the DDA work in Section 6.4 how to accomplish secure service associations for referentially and temporarily uncoupled devices.

Finally, we have learned that in large device groups where there is no established trust relationship between all devices and where the communication between two entities is not secure, additional security mechanisms have to be provided. However, this introduces new requirements for key management and configuration. While the Hausgeist approach of manually configuring public keys has been intended as a first solution for a prototype system and is not likely to be usable in real-world scenarios, the alternative of operating a public-key infrastructure within a home network is also not very attractive. It introduces central entities to a otherwise decentralized system and imposes additional requirements for users to deploy the whole system.

Summarizing, we can state that the initial proposition, the concept of a home network coordination framework that is based on soft-state group communication, has been (partially) validated. We have seen many use cases where an entity can perform useful functions relying on the soft-state updates that a certain device, e.g., a sensor, generates. On the other hand, we have also learned that reliable point-to-point communication, employing the RPC paradigm, is also an absolute requirement, especially for controller-controllee scenarios, e.g., for crucial control communication. For example, when the user wants to lock the house and sends a corresponding command to the central locking system, an RPC message seems to be the natural way to implement this.

10.5 Summary

The applications and projects that we have presented in the previous sections have revealed different usages of the Mbus protocol for the different application types. For example, Mbus communication for loosely coupled devices in ad-hoc communication scenarios imposes additional requirements for service and interface discovery, as we have discussed.

The fundamental Mbus transport mechanisms and the Mbus Guidelines interactions have proven to provide useful services for all the presented applications. For both tightly coupled application entities and more loosely coupled cooperating devices, the Mbus service set of simple soft-state messaging, for point-to-point and group communication, and more structured interaction schemes has turned out to be a good combination. For example, in many applications there are some coordination tasks that can best be realized by disseminating event notifications via multicast to an interested group of receivers, whereas for other coordination tasks the “traditional” RPC model is just adequate.

In our discussion of Mbus-based conferencing endpoints in Section 10.1.1 and Section 10.1.2, we have described the design of different systems and application entities. We have analyzed how the Mbus-based design of the RAT application enables its re-use in different ap-

plication contexts. RAT and CONTRABAND were one of the first Mbus-based applications and can be considered as proof-of-concept applications that have focused on the decomposition of applications. The prevalent usages of RAT as an application module suggest that this model is successful. For example, the Bonephone application is just one example of an Mbus-based endpoint application that is made out of existing components "glued" together by Mbus communication.

The applicability of Mbus to the development of decomposed conferencing applications has also been demonstrated by our work in the DTI and FETA projects, where we have been able to integrate Mbus into a specific market-ready IP telephony endpoint device. While the DTI project has concentrated on validating the concepts of Mbus-based endpoints and the Mbus Call Control model, the FETA project has focused on dynamically *extending* the capabilities of endpoints by adding new components. The foundation for adding components dynamically, the DDA framework, has been developed as a mechanism to discover and to associate to Mbus services on a local network in order to integrate them into a user's Mbus session.

The endpoint and gateway related applications have first led to the development of the Mbus Call Control Profile and our generic endpoint/gateway architecture concept, which we have been able to refine and validate during several research projects. In the DTI and FETA projects, we have used the Call Control Profile as a way to integrate an IP telephone into a user's computing environment by making the phone controllable through dedicated control applications. In our gateway work, we have employed the Call Control Profile as one building block of our generic controller framework: the Call Control Profile defines a call control protocol independent interface that is implemented by specific call control engines. A controller can coordinate these engine through the Call Control commands, without having to be know the specific call control protocol.

We have seen that gateway architectures and the involved coordination tasks can be significantly more complex as it is the case for endpoint architectures. We have presented our implementation design that is based on a finite state machine concept, which has proven to be helpful to systematize the distributed state maintenance in a strictly asynchronous application.

The Hausgeist project has investigated the deployment of Mbus for home automation — an interesting application domain, that provides significant different characteristics and requirements than the conferencing application domain. In home networks, the Mbus is not used to coordinate application components in a locally distributed application but provides a communication channel for rather loosely coupled devices. We have learned that dynamic interface and capability discovery is a required mechanism in these environments and have presented the Mbus based approach that Hausgeist has pursued for implementing this mechanism. Interestingly, our work in the FETA project for developing an architecture that allows users to integrate application entities into a conferencing system without manual configuration or special controller support, has indicated that some form of dynamic interface discovery is useful for this application as well. It is likely that we will address this issue by future research work.

Chapter 11

Conclusions

In this thesis, we have developed a local coordination architecture for interpersonal communication systems, which includes the definition of transport and association protocols and the development of an application specific architecture that is based on the fundamental protocol specifications. The research work that has been performed in the context of this thesis has two origins:

- The initial objective was the development of a system architecture for conferencing and gateway systems that accommodates the high degree of modularity and heterogeneity that we have identified for these systems. In particular, we wanted to create mechanisms for the *internal management services* that are required, especially for more tightly-coupled systems.

  Considering existing conferencing architectures and first attempts of providing local coordination services, we have concluded that, given the vast heterogeneity of specific endpoint applications and control models, our solution must provide a general-purpose coordination service, on which application-specific semantics can be based. Consequently, we have decided to clearly separate the general coordination mechanisms from application semantics, and have designed a general-purpose message-oriented coordination mechanism.

  The design of this mechanism has been based on certain assumptions and requirements that we have derived from analyzing existing systems and from estimating future trends for the application area of multimedia conferencing. In particular, the most important objectives that stem from this analysis are:

  1. providing simple and efficient group coordination mechanisms for application developers without neglecting robustness and consistency requirements;
  2. being able to coordinate application entities that reside on different hosts; and
  3. building on Internet technologies and considering the specific requirements of real-world IP networks.

- The second objective of our research work stems for the idea of *generalizing* the local coordination service and from the observation that in pervasive computing environments, there is a distinct need for service coordination in order to exploit the capabilities of

networked devices: many devices that provide a certain service for a user, provide a network interface (of some form) and can benefit from other devices' services.

We have considered the personal desk area computing environment of office workers as an example, because it is related to our conferencing background, as IP telephony and desktop multimedia conferencing systems are currently gaining deployment momentum as specific elements in these environments.

We have stated that, although the fundamental multimedia technology can be considered quite mature, there has not been significant deployment of conferencing technologies until recently. Besides other factors, we have ascribed this to the limited usability and tailorability of desktop multimedia systems and to the fact that they have not been designed for taking advantages of application entities that might be available in a user's computing environment.

Consequently, we have addressed service discovery and coordination requirements for desk area computing environments in general and have developed Mbus-based architectures and association protocols for this purpose.

In the following, we summarize the main conceptual achievements in Section 11.1, and we describe our main engineering results in Section 11.2. In Section 11.3, we compare Mbus to other existing coordination frameworks, and in Section 11.4, we list some possible optimizations, that we would recommend on the basis of the lessons learned during developing and deploying Mbus-based applications and provide some prospects on next steps on the conceptual level and on possible future deployment scenarios.

11.1 Conceptual Achievements

Starting off from an analysis of relevant technologies for distributed computing, we have described merits, limitations and typical usage scenarios for protocols and protocol frameworks such as RPC, ISIS and TIBCO Rendezvous. In this thesis, we have argued that for effective coordination in groups of application components, multiple communication patterns must be supported, including asynchronous group communication and point-to-point RPC-like communication.

We have designed the Mbus protocol as a group communication mechanism that is not layered on top of RPC communication but employs message oriented communication in one-to-many multicast and unicast mode. The key idea is to provide flexible addressing mechanisms on the Mbus transport layer that are inspired from *subject-based addressing* architectures such as TIBCO Rendezvous (Section 4.3.1) and that can be mapped to efficient network layer multicast and unicast transport.

In principle, the Mbus addressing concept relies on *receiver-based filtering*, i.e., receivers decide which messages to process based on the Mbus addressing rules. This fundamental concept allows for many different communication models, such as multicasting soft-state event notifications and traditional request-response communication between two peers.

With this general-purpose communication mechanism as a basis, we have designed a framework for the decomposition of applications into Mbus entities with the objective to support component re-use and component interoperability, regardless of specific platform and programming language constraints. Unlike other component frameworks, the Mbus model is not tied to

a specific platform or programming language, but can be viewed as a *protocol approach*: any entity implementing the basic Mbus transport mechanisms and providing a certain well-defined Mbus interface can be used as a component in an Mbus-based application. We have defined the *Mbus Guidelines* as a set of application-independent conventions and protocol mechanisms and have conceived the model of *Mbus application profiles* that are defined in terms on Mbus Guidelines abstractions. I.e., we do not employ formal interface descriptions and dynamic discovery of interfaces, but rely on informally defined interfaces (that an application developer has to be aware of), and Mbus entities indicate their support of a certain profile by their Mbus address.

In addition to the concept of Mbus-based decomposed applications with a rather tight coupling of entities, we have developed the *Dynamic Device Association* concept for allowing dynamic association of Mbus entities.

With the Mbus protocol, the Mbus Guidelines and the DDA protocol as fundamental building blocks, we have designed semantics for a specific application — the local coordination of conferencing endpoint and gateway systems. Considering existing conferencing architectures and protocol families, we have developed a local coordination model for decomposed conferencing applications. The core concept of this model is the generalization of the internal coordination, embodied by the *generic controller* concept. In our local coordination model, generalization is achieved by relying on a two-pronged approach: We have defined standardized Mbus-based protocols for the communication between controllers, application entities and call control engines, and we have developed a standardized description language for application and system capabilities.

11.2 Engineering Results

The conceptual work has led to a set of results in the form of protocol specifications and developed systems that we name in the following.

Mbus Transport Protocol

A fundamental result of our research activities is the development of the Mbus transport specification.[1] The Mbus transport protocol has been developed and tested first for conferencing endpoints and applications such as the CONTRABAND system (Section 10.1.1) and RAT (Section 10.1.2). We have validated its design and the interoperability of the different implementations by conducting several interoperability and testing events. We have discussed the protocol specification in the IETF MMUSIC working group, which has led to its publication as RFC 3259 [RFC3259]. We have provided another description of the Mbus at [Ott00].

Implementations

The Mbus transport protocol and the Guidelines interactions have been implemented for different programming environments, and these implementation are in general publically available.[2] Experiences with the development and deployment of implementations have helped us to

[1] The Mbus transport specification has been developed together with Jörg Ott and Colin Perkins.

[2] A list of Mbus implementations can be found at `http://www.mbus.org/`.

advance the Mbus protocol specification and the existence of implementations for many different environments has promoted the usage of Mbus and the re-use of existing Mbus components. During our work on Mbus protocol implementations and corresponding applications for different programming environments we have developed a set of implementation strategies for the design of asynchronous, message-oriented distributed applications, which we have documented in Chapter 7, *Mbus Implementations*.

Dynamic Device Association

The Dynamic Device Association work [Kutscher03b] [Kutscher03c] provides the establishment of Mbus sessions in ad-hoc environments and has been developed to provide a means to securely integrate services and Mbus entities into Mbus sessions. In addition, the DDA framework provides Mbus extensions that allow for the deployment of Mbus in heterogeneous networks, with non-constant multicast connectivity between all Mbus entities. Both technologies have been implemented and applied to projects, which we have described in Section 10.2.

Endpoint and Gateway Architecture

The generic controller concept has been the basis for the development of our local endpoint and gateway architecture. The architecture that is described in Section 9.1 is based on the concept of *internal management services* that are provided by a controller that coordinate application entities, call control engines and potentially other Mbus entities. We have implemented this concept for different endpoint and gateway systems, which we have described in Chapter 10, *Mbus in Projects*. The work on our coordination architecture has raised interesting questions concerning communication patterns in coordination-based distributed systems, e.g., the question how to deal with event subscription in group communication scenarios and how to design multi-controller interactions (see also Section 11.4.2).

Mbus Call Control Profile

The Mbus Call Control Profile [Ott01] has been developed as one element of our endpoint and gateway architecture as a protocol for controlling Mbus-based call control engines such as SIP implementations. The main characteristics of the Call Control Profiles are its call control protocol independence and its applicability to both endpoints and gateways.

The Call Control Profile has been defined in terms of Mbus Guidelines abstractions and allows for managing a call state in a distributed fashion, based on event notifications and Mbus RPCs. We have developed implementation strategies for distributed state management in asynchronous environments such as the finite state machine-based approach that we have described in Section 10.3.3.

Session Description and Capability Negotiation

The second building block of our local conferencing architecture is SDPng, a description format for capabilities and configurations of conferencing systems [Kutscher01c] [Ott02b]. The key idea of SDPng is to provide a general framework for describing and for negotiating capabilities of conferencing systems. The framework is application independent, and corresponding processors know nothing about application semantics but are nevertheless able to process, i.e., to aggregate and to negotiate, configurations. In our local endpoint architecture, a controller

can thus query the capabilities of the present application entities, aggregate this information into a combined capability description of the whole endpoint and can then coordinate a call control engine through Mbus Call Control commands to consider this capability description for the negotiation process during a call setup, which is described in detail in Section 9.2.3.

11.3 Mbus Compared to Other Approaches

When comparing Mbus to other work, we have to distinguish between the Mbus coordination framework and the application-specific local endpoint/gateway architecture.

Our discussion in Chapter 4, *Foundations and Related Work* has considered different existing technologies that we have distinguished as *related work* (LBL Conference Bus, PMM) and *building blocks* (some elements of local coordination protocols and component frameworks). The analysis of the LBL Conference Bus and PMM has already shown that both technologies represent fairly simple, application-specific solutions that have not been intended to be complete, application-independent coordination frameworks. Table 4.1 provides a comparison of the two approaches and also lists all the requirements (from our requirements discussion in Section 3.2). The discussion of Mbus in this thesis should have made clear that, compared to the early coordination approaches for conferencing systems, Mbus is a general-purpose coordination mechanism that addresses important requirements such as security and structured communication through standardized interaction patterns.

In the following, we compare Mbus to two selected general-purpose coordination frameworks (TIBCO Rendezvous and UPnP) in order to delimit the different application areas and to highlight different design decisions.

TIBCO Rendezvous

TIBCO Rendezvous (Section 4.3.1) is a coordination framework that is targeted at large-scale coordination of process groups, i.e., coordination of applications in an enterprise domain. Similar to Mbus, TIBCO Rendezvous relies message-oriented coordination and uses self-describing messages.

Our design of the Mbus application layer addressing scheme provides some interesting similarities to the TIBCO Rendezvous *subject-based addressing* concept. Both Mbus and TIBCO Rendezvous rely on *receiver-based filtering* of messages that is based on an application layer addressing concept: In TIBCO Rendezvous, the filtering is performed on the *subject* of messages. Applications specify — for their local TIBCO Rendezvous protocol stack — which filters should be applied on incoming messages, where applications can use *wildcard expressions* to specify the selected subjects. For the Mbus, we also use receiver-based filtering of messages on the basis of application layer addresses, however, the specification of *wildcards* is not done by receivers but by *senders*: Mbus senders specify group destination addresses by constructing partly qualified addresses that can represent entity groups on the Mbus.

In the TIBCO Rendezvous model, entities can be completely referentially decoupled, because subjects are used as message destination specifications, which are not coupled to entity identities. For the Mbus, we pursue another approach: entities learn of the existence of other entities on the basis of the Mbus entity awareness mechanism and can constantly monitor the availability of communication peers. The entity awareness mechanism is used to maintain a list of Mbus addresses for the entities in the current Mbus session, and applications can tell at

any time, whether the required communication peers are present or not. An Mbus application that wants to send a message can use a unicast (fully qualified), a multicast (partly qualified) or a broadcast (empty) address, and the Mbus protocol layer selects the appropriate transport mechanism, e.g., UDP/IP unicast for a fully qualified destination address.

The NACK-based fault tolerance mechanism in TIBCO Rendezvous is also different from the Mbus ACK-based approach: In TIBCO Rendezvous, all messages are always sent to all session members, and receivers can detect message loss on the basis of gaps in the message sequence numbers per sender. Receivers that detect a lost message can request a retransmission, where the receiver cannot tell in advance, whether the message is interesting at all. Clearly, this approach can lead to unnecessary retransmissions and, in the presence of correlated message loss, to NACK-implosions when multiple receivers request a retransmission simultaneously. For the Mbus, applications can select reliable transport on a per message basis, and unicast messages can be sent using the reliable transport class that relies on a ACK-based retransmission mechanism. As a result, only messages that are explicitly sent using the reliable transport class will ever be retransmitted — typically this applies to messages that represent a request or a remote procedure invocation.

Summarizing, we can state that TIBCO Rendezvous and Mbus can both be categorized as message-oriented coordination-frameworks and provide some similar concepts; however, we can identify some differences with respect to specific protocol mechanisms. We conclude that TIBCO Rendezvous is rather targeted at message-oriented coordination in the enterprise domain, e.g., for the distribution of messages in a corporate network, which is also suggested by the recent adoption of PGM for large-scale group communication, i.e., message distribution beyond local network links. The Mbus on the other hand, is strictly limited to local coordination of application components — an application area where a (slightly) tighter coupling of entities is desirable and where the effort for achieving fault tolerance can be reduced in the interest of efficiency and scalability. In addition, the Mbus supports the dynamic, secure creation of coordination session by users through Dynamic Device Association — a concept that is not provided by TIBCO Rendezvous.

UPnP

Universal Plug and Play (Section 4.3.2.3) is a coordination framework that focuses on device control in local networks and relies on different protocols for different functions such as device discovery, device description, transmission of control commands, and event notification. With respect to the provided functionality, UPnP can roughly be compared to our Dynamic Device Association approach (Section 6.4) in conjunction with Mbus as an application session protocol.

While UPnP's service discovery and association mechanism provides similar features compared to our DDA approach, we have noticed some problems of SSDP (UPnP's *Simple Service Discovery Protocol*), especially the lack of scalability, which we have discussed in Section 4.3.2.3 and Section 6.4. SSDP is based on periodic service announcements that are sent in UDP multicast messages with an HTTP-like syntax, which are transmitted without bandwidth limitation, whereas the Mbus-DDA protocol employs SAP and its bandwidth limit for announcements.

For device control and service association, UPnP employs SOAP — an XML-based protocol for message-oriented point-to-point communication. SOAP, although not bound to a specific transport protocol, is today almost exclusively used with HTTP. Although HTTP itself has se-

curity concepts, such as Digest Authentication, and can itself be used over a secure channel through the use of TLS, secure communication with SOAP is not specified and effectively not used. As a result, security in UPnP is a problem yet to solve. For Mbus, security is built directly into the base protocol, and the DDA procedures address authentication and confidentiality explicitly. In fact, one of the goals for DDA is to securely distribute an Mbus transport and security configuration for integrating new entities into an existing Mbus session.

It is interesting to see how UPnP and Mbus map different interaction patterns to protocol mechanisms. In Mbus, there is essentially one fundamental messaging service that provides point-to-point and group communication and, on the basis of per-message selection, reliable message transport for point-to-point communication. On top of this basic service, we layer different interaction schemes, such as RPC communication and event notifications, where the set of predefined patterns can be extended by application programmers. UPnP on the other hand, uses different protocols for different communication aspects from the outset: SOAP is used for request/response communication, e.g., for invoking a remote procedure at a controlled device, and GENA, the *General Event Notification Architecture*, is used for distributing event notifications in an UPnP session. While SOAP is used in unicast mode only (HTTP), GENA is a message-oriented mechanism that can be used with unicast and multicast. As a matter of fact, GENA messages provide an HTTP-syntax but are sent via UDP, and SOAP is typically bound to HTTP as a transport protocol and is thus used with TCP.

Some of the differences of UPnP and Mbus can be ascribed to the different application domains and different assumptions and requirements for their operation. UPnP's lax security concept is motivated by the (initial) assumption that the protocol will only be used in closed domains, i.e., in home networks, where security has not been regarded as an issue. However, the evolution of network architectures, the increasing deployment of *always-on*, *always-connected* devices (as we have described in the introduction) has already disproved this assumption. Today, many home networks are already equipped with a WLAN infrastructure, and, given the security flaws in the currently often deployed IEEE 802.11 WEP mechanism [Borisov01], unauthorized access to home networks has become comparatively easy. For Mbus, that is targeted at the coordination of application components — a security-sensitive application — strong security has therefore been part of the base specification from the beginning.

11.4 Open Issues and Next Steps

By deploying the Mbus protocol in different application scenarios, we have identified some possibilities for advancing individual protocol functions and our local coordination architecture for conferencing systems, which we describe in Section 11.4.1 and Section 11.4.2. In addition, we have conceived new application areas for the Mbus framework, which we describe in Section 11.4.3.

11.4.1 Mbus Protocol

In the following sections, we describe some potential advancements of the Mbus transport mechanisms.

Entity Awareness

In Section 10.4, we have described some problems with very large Mbus sessions on overloaded systems: significant packet loss due to congestion can inhibit the adaptation mechanism for the sending rate of periodic `mbus.hello` messages, because an entity does not determine the correct number of entities in an Mbus session. As a result, the chosen sending rate will be too high, which will aggravate the situation. This is especially a problem in situations where many entities are started in parallel, each of which has no knowledge of the final number of entities, as entities keep appearing on the Mbus. Under normal conditions, the *timer reconsideration* mechanism would reduce the sending rate automatically superproportional in anticipation of an actual larger number of entities. However, this mechanism does not work when the observed number does not increase (significantly) due to message loss.

We have conducted some experiments in such scenarios, and have noted improvements when extending the timeout for received `mbus.hello` messages: entities would not remove other entities from their list of known entities as quickly, however this change does not help to solve the problem principally. It only increases the threshold by a certain degree that must be reached to destabilize the system. We have observed that the number of entities that each entity is aware of, differs from entity to entity, i.e., some entities see more entities than others. One possible solution to accelerate the process of converging with respect to the observed number of entities is to have each entity include a description of its current view of the session in the periodic `mbus.hello` messages. For example, if each entity announced the number of currently known entities, other entities could adopt that value (if it was higher than the own number) for the sending rate calculation. This would tend to result in reduced sending rates, which would in turn reduce the system load and buffer overflows and thus help to deliver more `mbus.hello` messages successfully.

Of course, it must be stated that overloaded systems and buffer overflows can be caused by different factors and that reducing the sending rate for `mbus.hello` message will not help in all cases. However, differing numbers for the group size are a good indication that a problem exists, and this mechanism could also be used to trigger other actions, e.g., user notifications.

Congestion Control

Another possible optimization is the introduction of congestion control mechanisms. In Section 6.2.1.5, we have discussed the general problem of providing congestion control for multicast communication without giving up scalability. The asynchronous communication model that is typically employed for multicast communication does not allow for implicit flow-control, i.e., a sender would not notice a congestion of network or receiver resources. This is especially a problem when entities reside on different network links with significantly different capabilities, e.g., in bridged environments. In this case, a subset of entities may experience congestion, whereas another subset does not, which makes it difficult to react properly.

One approach to avoid congestion without providing corresponding detection mechanisms is to apply traffic shaping techniques. For example, an Mbus interface could be configured to limit the number of messages per time and per destination. There could be a certain configured limit for Mbus messages that are sent over IP multicast and a certain limit for unicast messages. In a first version, these limits could be part of the Mbus configuration, i.e., be configured by the user, who might be aware of the network environment.

In a second version, one could try to adapt these limits on the basis of triggers from the Mbus transport layer. For example, if we extended the `mbus.hello` message as described

above, we could use differences in the observed number of entities as a congestion indication. Other indications could be the variability of `mbus.hello` message intervals per sender, and failure indications from the reliable transport layer, i.e., each time a retransmission is required.

The traffic shaping can be accompanied by a *type concept* for Mbus messages, and messages of different types could be processed differently with respect to traffic shaping. For example, soft-state updates of a certain variable could be assigned a lower priority than RPC messages, and a soft-state update that has not already been sent (due to traffic shaping constraints) could be obsoleted by a subsequent update for the same variable. In addition, applications could assign a *time to live* to outgoing messages, and if the transport layer cannot send the message in that time frame, the message will be discarded.

It should be noted that the management overhead for maintaining the required information to enforce the traffic shaping could seriously conflict with the simplicity goal: implementations would be required to maintain additional state per entity and would have to measure throughput etc., which may not be desirable in all cases.

Addressing Scheme

Recent experiments with Mbus in ad-hoc communication scenarios, where connectivity may not be constantly available, have indicated some potential for improvements of the Mbus addressing mechanism. In the current specification, an entity's Mbus address provides an `id` element that is used to uniquely identify the entity. The `id` element is generated from the host's IP address (for IPv6, it is generated from the lower 64 bits of the IP address).

In situations where the host IP address changes, e.g., caused by attaching to a new network, the Mbus address could thus change as well (if the Mbus implementation adopts the IP address change), which results in a loss of Mbus connectivity and Mbus control relationships (that are based on the Mbus address). For hosts with multiple interfaces, the mapping of IP address to Mbus `id` address element is also not optimal, because an Mbus application would typically provide only one Mbus interface that would be attached to the different IP interfaces, i.e., an IP address independent identifier would be more appropriate.

Finally, in our discussion of DDA multiparty peering in Section 6.5, we have described the establishment of Mbus sessions across several network links, some of which may not even be IP-based. These observations suggest that the `id` element should not be generated from an IP address, but from a unique identifier for a host. This *can* be an interface identifier, if is sufficiently unique, e.g., the interface identifier for IPv6 interfaces, or EUI64 identifiers in general.

Multicast Transport

The current Mbus protocol uses a single IP multicast group for native multicast communication. The corresponding IP address must be known to all session members in advance and can be configured as part of the Mbus configuration. In addition, there is a well-known, IANA registered default address. All Mbus group communication is mapped onto this IP multicast group. Mbus unicast communication can optionally be mapped to sending UDP datagrams directly to the IP address (and port) of the destination entity. While this model allows for simple sender implementations, it requires every receiver to filter received multicast messages based on their destination address.

In order to take advantage of hardware based filtering of IP multicast addresses (that can

be done by the network adapter), the Mbus protocol could be extended to employ multiple IP multicast groups, i.e., by mapping Mbus group addresses to IP multicast addresses. A message that is sent to a specific Mbus group (which is defined by sending to a partly qualified Mbus address) would not be sent to the default IP multicast group but to a unique IP multicast address. An entity that wants to receive a corresponding message has to join the respective IP multicast group. The received message can be delivered to the application directly, i.e., without filtering it on the basis of its Mbus address.

Two issues must be considered for implementing the mapping of Mbus group addresses to IP multicast addresses:

1. Entities must be able to deterministically compute an IP multicast from a given Mbus group address. For a given Mbus session, any list of Mbus address elements that does not represent one of the currently known Mbus entities, is an Mbus group address. I.e., the set of Mbus group addresses for a given Mbus session depends on the current set of Mbus entities and their addresses. In principle, the list of potential Mbus group addresses can be obtained by generating all permutations of the entities' address elements and by removing the completely qualified addresses later.

   For example, in an Mbus session with three entities `(id:0 A:1 B:2)`, `(id:1 A:1 C:3)` and `(id:2 B:2 D:4)` there are the Mbus groups `(A:1)`, `(B:2)` and the all entities group `()`.

   Relying on the Mbus entity awareness mechanism, this list can be determined by any Mbus entity. The mapping to IP addresses could be performed by applying a standardized hash function that calculates a fragment of an IP address that is combined with a session specific address prefix (assuming IPv6 addresses for these considerations).

2. In order to avoid address clashes for the use of dynamically determined multicast addresses, Mbus implementations must interface with multicast address allocation infrastructures. For example, [RFC2908] describes the Internet Multicast Address Allocation Architecture. Alternatively, there are zeroconf approaches such as the Zeroconf Multicast Address Allocation Protocol (ZMAAP) [Catrina02] for IPv4 and the concept of link scoped IPv6 multicast addresses as proposed by [Park02] that allow applications to obtain unique multicast addresses in the absence of multicast address allocation infrastructures.

Message Syntax

The Mbus message syntax as described in Section 6.2.3 and RFC 3259 is currently based on a separation of a message into header and payload, where the header fields are whitespace-separated. The Internet Message Format [RFC2822] is a syntax for text messages that is widely deployed by different Internet protocols such as SMTP, HTTP, SIP and RTSP. In essence, the Internet Message Format provides the concept of header fields that are CRLF separated and that consist of a field name and a field body (separated by ":"). In principle, it is conceivable to adopt this format for Mbus messages as well, e.g., in order to align Mbus with other text message-based protocols. However, there are not many compelling technical reasons to change the format, because the Mbus message header syntax is quite simple and can be parsed efficiently. Furthermore, a Mbus message header is a fixed set of fields, which means there is no

need to explicitly name header fields. Finally, even in the most optimized case, using single-character field names, the RFC 2822 syntax would still require an additional overhead of 4 bytes per header field (field name, colon separator, and CRLF), which would result in 28 bytes for a complete message header.

11.4.2 Conferencing Architecture

Based on the design of our local conferencing architecture that we have described in Section 9.1, we can conceive some future research work and engineering work for advancing the local co-ordination services, in particular, the Mbus Call Control Profile, for extending the coordination services for application entities, and for applying the SDPng framework to specific application types. Two advancements are planned for the Mbus Call Control Profile:

1. The development of more advanced H.323 and SIP call control engines and the consideration of more dynamic endpoint configurations has revealed a need for extending the Call Control commands with a mechanism for *changing* the conference configuration *during* a conference. For example, H.245 supports the addition and removal of application sessions at any time during a call, and SIP also provides the `UPDATE` request for updating the session description. In our FETA scenario, it would be quite useful to extend the capabilities of a endpoint during a call and re-negotiate the conference configuration accordingly.

   One of the next enhancements of the Call Control Profile will therefore be the definition of a `conf.call-ctrl.update` RPC and a corresponding event notification for indicating the reception of update requests. This is currently being implemented in our H.323 call control engine. For the long term perspective, we are considering a re-design of capability negotiation and configuration distribution model, e.g., by distinguishing capability descriptions and configuration descriptions on the Mbus.

2. In Section 9.3.4, we have already discussed the *multi-controller* extension of the Mbus Call Control Profile and presented a first design for a multi-controller call-control communication. In essence, the design relied on updating all registered controllers and on some support for dealing with concurrent state changes.

   The addition of multi-controller support will be an intrusive change and require an adaptation of all Call Control implementations.

For realizing our local coordination concepts, we have first concentrated on the Call Control Profile, as the coordination of call control engines is the fundamental building block that is required for establishing conferences in the first place. Another building block is the coordination of application entities. In Section 10.1.2, we have presented the Mbus interface of the audio tool RAT, probably the most often deployed Mbus application entity.

Although RAT's Mbus interfaces provides some generic, application-independent commands such as RTP-related commands, the operation of the media engine by controllers requires application specific knowledge, e.g., with respect to the initialization sequence. Building on the experiences with this interface, the next step is to design a second-generation coordination model, which includes an Mbus Guidelines-based interface and the definition of application-independent commands for configuration and coordination.

One crucial element for the advancement of the local coordination architecture is the *interoperability service*, i.e., the capability description and negotiation framework. In Section 9.2, we have described the SDPng framework as an application-independent framework for capability negotiation and configuration description. Building on this framework, one of the next steps will be the definition of application semantics, i.e., SDPng packages for different applications. In addition, new usage scenarios with respect to the configuration of conferencing systems will be investigated, e.g., dynamic changing of configurations and gateway scenarios: in order to decide which media transcoding configuration to use, a corresponding gateway must combine the endpoint's capabilities with its own capabilities and determine suitable transcoding configurations. Capability negotiation and conference configuration for multi-party conferences is another example.

11.4.3 New Application Areas

The Mbus transport protocol and the Guidelines interactions have been defined as application-independent mechanisms, and our use case discussion in Section 3.1 as well as our description of the Hausgeist project in Section 10.4 has already presented two potential application areas beyond the conferencing domain.

Building on our experiences with group communication for device coordination in the Hausgeist project, we are considering the application of the Mbus *concepts* to device coordination in general, e.g., for wireless sensor networks and wireless personal area networks such as IEEE 802.15 networks.

For many application scenarios of these networking technologies, we can identify similar concepts as those we have considered for the Mbus protocol. For example, in wireless sensing networks, there are a group of devices that can be brought together in an ad-hoc fashion and that need to discover each other. These devices provide a certain service, and it is conceivable that some information may be shared between devices. In addition, there are different interaction types, e.g., RPC-like communication for configuring and controlling sensors and event notifications for disseminating sensor information. Communication inside a wireless sensor network is limited to topological scope, usually the ad-hoc network that the devices establish.

The similarity of concepts is notable, and we believe that the Mbus messaging service for locally distributed applications can in fact be *conceptually* applied to sensor networks. However, there are some specific characteristics of wireless sensor networks and personal area networks that will require an adaptation of the Mbus transport *mechanisms*:

- In wireless ad-hoc networks, we have to consider intermittent connectivity, e.g., caused by mobility. Although the current Mbus awareness mechanism would help to detect that a device is no longer reachable, the rules for membership management would have to be relaxed in order to tolerate periods of interrupted connectivity.

- Some specific wireless networking technologies have significantly reduced bandwidth capabilities. For example, the IEEE 802.15 Working Group for WPANs[3] is developing a low data rate link layer technology for wireless personal area networks that is intended to allow for extremely battery-friendly operation (multi-month to multi-year battery life)

[3] http://www.ieee802.org/15/pub/TG4.html

and to impose very little complexity on implementations. This technology is intended for small devices in personal area networks and similar environments. The current specification defines data rates from 20 KBit/s to 250 KBit/s.

In order to allow for the application of an Mbus-inspired technology in such a network environment, both bandwidth efficiency and battery-friendly operation must be addressed. For example, it might be required to define a "battery-friendly" Mbus mode that does not require the periodic sending (and processing) of `mbus.hello` messages. Consequently, the entity awareness mechanism would have to be changed.

Furthermore, a more efficient and application-tailored message representation may be required. From our simulation results in Chapter 8, *Evaluation*, we have learned that the parsing of the text messages can be a substantial factor for the required message processing effort. In addition, techniques such as header compression could be investigated to reduce the required bandwidth.

We believe that in general, applications for wireless ad-hoc networks can benefit from a multicast-based coordination infrastructure that provides messaging services and accommodates ad-hoc communication scenarios. In addition to the Mbus-related considerations, there are of course other technical considerations such as the link layer mapping of IP and IP multicast for the corresponding network technologies. Then, there is the issue of providing IP connectivity for mobile nodes in an ad-hoc network, e.g., by relying on mobile ad-hoc routing protocols.

11.4.4 Lessons Learned

In Section 11.4.1, we have discussed some potential improvements that we have conceived based on our experiences so far. If we started the work on Mbus now, we would try to consider some these ideas right from the beginning, in particular:

- We would generalize the scope and would not tie Mbus to IP networks. In the subsection entitled *Addressing Scheme* in Section 11.4.1, we have explained why a mapping from IP address to the Mbus `id` address element is not desirable. For a re-design, we would define the protocol behavior independent of the underlying network.

- We would consider congestion control right from the beginning. Although, in most of our applications, congestion has never been an issue, it can be an issue for large-scale deployment, e.g., in home-networking scenarios.

- The Mbus Call Control Profile has initially been designed as a strict controller/controllee model, i.e., exactly one call control engine is controlled by exactly one controller. In situations where multiple entities were a) interested in call control events or b) wanted to participate in the management of the call control engine, we had to effectively cascade the control chain, e.g., by operating a user interface entity as a controller of the endpoint controller. In such as scenario, many interactions have essentially been performed twice. For example, when initiating a new call, the user interface component has to send a call setup request to the controller, which in turn sends the `conf.call-control.call` command to the call control engine. The communication and control relationships could

be simplified by relying on a multi-controller paradigm right from the beginning, as we have described in Section 11.4.2.

Concluding Remarks

The main subject of this thesis is the coordination of locally distributed component-based systems using a message-oriented coordination mechanism. Summarizing, we can state that the idea of applying group communication concepts to local coordination has resulted in an interesting architecture centered around message-oriented coordination with group communication *above* the IP layer. While the Mbus uses IP multicast as a fundamental distribution and rendezvous mechanism, it uses application layer addressing for qualifying entity destination groups and individual destination entities. Comparing Mbus to other message-oriented coordination frameworks, one of the attractive characteristics of our approach is that it considers requirements for a broad spectrum of application scenarios and host platforms — ranging from small one-chip computers with minimal IP implementations to embedded systems such as IP telephones, and to workstations. This broad applicability and the simple rendezvous mechanisms without the need for central network entities enable the development of Mbus-based distributed applications in a number of different application areas.

One such application area certainly is the local coordination of conferencing endpoints and gateways. In this thesis, we have demonstrated that Mbus (and the application-specific conferencing profiles, such as the Call Control Profile), are an adequate solution for implementing the local coordination and interoperability services in both protocol worlds: H.323 and SIP-based conferencing.

In addition, our generalized approach for both the Mbus transport mechanisms and the Dynamic Device Association procedures for bootstrapping Mbus sessions enable the application of the message-oriented coordination framework to a host of different application areas, some of which we have already portrayed in this thesis, e.g., the area of home networking and coordination of user devices in in-vehicle networks. In essence, we believe that for realizing the vision of *pervasive computing* that we have referred to in the introduction, a coordination framework is needed that provides at least some of the features that we have discussed in this thesis, e.g., rendezvous functions, group communication, application layer addressing, reliable transport mechanisms, and security. While it is yet unclear, how much of the original Mbus specification is eventually going to be employed for such an ubiquitous coordination framework, it is likely that the concepts and techniques that we have discussed in this thesis will be considered by future research and engineering work in that area.

Bibliography

[6WINIT02] *Advanced Aids to Deployment*, 6WINIT Project, 6WINIT Deliverable D17, available online at `http://www.6winit.org/`, December 2002.

[AMIC03] *AMI-C Release 2*, , available online at `http://www.ami-c.org/`, February 2003.

[Adams03] *Protocol Independent Multicast - Dense Mode (PIM-DM)*, Andrew Adams, Jonathan Nicholas, and William Siadek, Internet Draft draft-ietf-pim-dm-new-v2-03.txt, Work in Progress, February 2003.

[Ark99] *A look at human interaction with pervasive computers*, Wendy S. Ark and Ted Selker, IBM Systems Journal, Volume 38, Number 4, July 1999.

[Arkko03] *Key Management Extensions for Session Description Protocol (SDP) and Real Time Streaming Protocol (RTSP)*, Jari Arkko, Elisabetta Carrara, Fredrik Lindholm, Mats Naslund, and Karl Norrman, Internet Draft draft-ietf-mmusic-kmgmt-ext-07.txt, Work in Progress, February 2003.

[Aura97] *Strategies against Replay Attacks*, Tuamos Aura, 1997, IEEE Computer Scociety Press.

[Babaoglu93] *Consistent Global States of Distributed Systems: Fundamental Concept and Mechanisms*, Özalp Babaoglu and Keith Marzullo.

[Baugher03] *Group Key Management Architecture*, Mark Baugher, Ran Canetti, Lakshminath Dondeti, and Fredrik Lindholm, Internet Draft draft-ietf-msec-gkmarch-05.txt, Work in Progress, June 2003.

[Birell84] *Implementing remote procedure calls*, Andrew Birell and Bruce J. Nelson, ACM Transactions on Computer Systems, pages 39-59, February 1984.

[Birman93] *A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication*, Ken Birmann, October 1993.

[Birman94b] *Reliable Distributed Computing with the Isis Toolkit*, Kenneth P. Birman and Robert van Renesse, IEEE Computer Society Press, Los Alamitos, 1994.

[Birmann00] *The Horus and Ensemble Projects*, Kenneth P. Birmann, Bob Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robert van Renesse, Ohad Rodeh, and Werner Vogels, Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX '00), Hilton Head, South Carolina, January 2000.

[Birmann87a] *Reliable Communication in the Presence of Failures*, Kenneth P. Birman and Thomas Joseph, ACM Transactions on Computer Systems, February 1987.

[Birmann87b] *Exploiting Virtual Synchrony in Distributed Systems*, Kenneth P. Birmann and Thomas Joseph, Proceedings of the 11th ACM Symposium on Operating Systems Principles , November 1987.

[Birmann91] *Lightweight Causal and Atomic Group Multicast*, Kenneth P. Birmann, André Schiper, and Patrick Stephenson, ACM Transactions on Computer Systems, August 1991.

[Birmann94a] *RPC Considered Inadequate*, Kenneth P. Birman and Robert van Renesse.

[BluetoothSIG01] *Specification of the Bluetooth System, Version 1.1*, Bluetooth Special Interest Group, February 2001.

[Borisov01] *Intercepting Mobile Communications: The Insecurity of 802.11*, Nikita Borisov, Ian Goldberg, and David Wagner, Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking, 2001.

[Bormann01] *Simple Conference Control Protocol – Service Specification*, Carsten Bormann, Dirk Kutscher, Jörg Ott, and Dirk Trossen, Internet Draft draft-ietf-mmusic-sccp-01.txt, Work in Progress, February 2001.

[Bormann94a] *MTP-2: Towards Achieving the S.E.R.O. Properties for Multicast Transport*, Carsten Bormann, Jörg Ott, Hans-Christian Gehrcke, Torsten Kerkschat, and Nils Seifert, International Conference on Computer Communications and Networks (ICCCN 94), 1994.

[Bormann94b] *Xy and Xmc — Scalable Window Sharing an Mobility, or, From X Protocol Multiplexing to X Protocol Multicasting*, Carsten Bormann and Gero Hoffmann, The X Resource, Janurary 1994.

[Bormann99] *MTP/SO: Self-Organizing Multicast*, Carsten Bormann, Jörg Ott, and Nils Seifert, Internet-Draft draft-bormann-mtp-so-02.txt, Work in Progress, June 1999.

[Box00] *Simple Object Access Protocol (SOAP) 1.1*, Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer, W3C Note 08, `http://www.w3.org/TR/SOAP/`, May 2000.

[Buesching01] *Entwicklung einer verteilten Architektur für ein modulares System zur Adaptation von Steuer- und Medienströmen in Multi-Protokollumgebungen für IP-Telefonie und Multimediakonferenzen*, Andreas Büsching, Diploma Thesis, Universität Bremen, November 2001.

[Calhoun02] *Diameter Base Protocol*, Pat R. Calhoun, John Loughney, Erik Guttman, Glen Zorn, and Jari Arkko, Internet Draft draft-ietf-aaa-diameter-17.txt, Work in Progress, December 2002.

[Catrina02] *Zeroconf Multicast Address Allocation Protocol (ZMAAP)*, Octavian Catrina, Dave Thaler, Bernhard Aboba, and Erik Guttman, Internet Draft draft-ietf-zeroconf-zmaap-02.txt, Work in Progress, October 2002.

[Cheriton93] *Understanding the Limitations of Causally and Totally Ordered Communication*, David R. Cheriton and Dale Skeen, ACM SIGOPS'93, 1993.

[Cheshire02] *Dynamic Configuration of IPv4 Link-Local Addresses*, Stuart Cheshire, Bernard Aboba, and Erik Gutmann, Internet Draft draft-ietf-zeroconf-ipv4-linklocal-07.txt, Work in Progress, August 2002.

[Clark88] *The Design Philosophy of the DARPA Internet Protocols*, David D. Clark, ACM SIGCOMM Proceedings, pages 106-114, `http://citeseer.nj.nec.com/clark88design.html` , Stanford, CA, August 1988.

[Clark90] *Architectural Considerations for a New Generation of Protocols*, David D. Clark and David L. Tennenhouse, ACM SIGCOMM Proceedings, pages 200-208, September 1990.

[Crowcroft93] *Lightweight Protocols for Distributed Systems*, Jon Crowcroft, Phd Thesis, 1993.

[Deering91] *Multicast Routing in a Datagram Internetwork*, Stephen Deering, Phd Thesis, December 1991.

[Dolev95] *The Design of the Transis System*, Danny Dolev and Dalia Malki, Proceedings of the Dagstuhl Workshop on Unifying Theory and Practice in Distributed Computing , September 1995.

[EASTEEA02] *Embedded Electronic Architecture — Scenario 2005 and Study Report*, EAST-EEA Project, Deliverable D1.1, Version 3.0, April2002.

[Fenner03] *Protocol Independent Multicast - Sparse Mode (PIM-SM)*, Bill Fenner, Mark Handley, Hugh Holbrook, and Isidor Kouvelas, Internet Draft draft-ietf-pim-sm-v2-new-07.txt, Work in Progress, March 2003.

[Floyd97] *A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing*, Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, IEEE/ACM Transactions on Networking, Volume 5, Number 6, pages 784-803, December 1997.

[Gemmell03] *The PGM Reliable Multicast Protocol*, Jim Gemmell, Todd L. Montgomery, Tony Speakman, Nidhi Bhaskar, and Jon Crowcroft, IEEE Network Magazine, January 2003.

[Gevros01] *Congestion Control Mechanisms and the Best Effort Service Model*, Panos Gevros, Jon Crowcroft, Peter Kirstein, and Saleem Bhatti, IEEE Network Magazine, May 2001.

[Gillet00] *A Taxonomy of Internet Appliances*, Sharon Eisner Gillett, William H. Lehr, John T. Wroclawski, and David D. Clark, `http://citeseer.nj.nec.com/384855.html`, 2000.

[Gold95] *When My Father Mows The Lawn Is He A Cyborg?*, Rich Gold, Xerox PARC, 1995.

[Handley00] *The Internet Multimedia Conferencing Architecture*, Mark Handley, Jon Crowcroft, Carsten Bormann, and Jörg Ott, Internet Draft draft-ietf-mmusic-confarch-03.txt, Work in Progress, July 2000.

[Handley95] *The Conference Control Channel Protocol (CCCP): A scalable Base for Building Conference Control Applications*, Mark Handley, I. Wakeman, and Jon Crowcroft, SIGCOMM Proceedings, pages 275-287, 1995, `http://citeseer.nj.nec.com/handley95conference.html` .

[Handley97] *On Scalable Internet Multimedia Conferencing Systems*, Mark Handley, November 1997.

[Harney03] *GSAKMP*, Hugh Harney, Uri Meth, Andrea Colegrove, Angela Schuett, Patrick McDaniel, Gavin Kenny, Haitham S. Cruickshan, and Sunil Iyengar, Internet Draft draft-ietf-msec-gsakmp-sec-03.txt, Work in Progress, August 2003.

[Hausgeist02] *Hausgeist*, Hausgeist Project, Project Report, Universität Bremen, December 2002.

[IEEE99] *Information Technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Specific requirements —*, LAN MAN Standards Committee of the IEEE Computer Society, 1999, ANSI/IEEE Std 802.11, 1999 Edition.

[ITU00] *Call Signaling Protocols and Media Stream Packetization for Packet Based Multimedia Communications Systems*, ITU-T, ITU-T Recommendation H.225.0, 2000.

[ITU88] *Pulse Code Modulation (PCM) for Voice Frequencies*, ITU-T, ITU-T Recommendation G.711, 1988.

[ITU93a] *Narrowband Visual Telephone Systems and Terminal Equipment*, ITU-T, ITU-T Recommendation H.320, 1993.

[ITU93b] *Video Codec for Audiovisual Services at p x 64kBits/s*, ITU-T, ITU-T Recommendation H.261, 1993.

[ITU95a] *Adaptation of H.320 Visual Telephone Terminals to B-ISDN Environments*, ITU-T, ITU-T Recommendation H.321, 1995.

[ITU95b] *Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Guaranteed Quality of Service*, ITU-T, ITU-T Recommendation H.322, 1995.

[ITU95c] *Control Protocol for Multimedia Communication*, ITU-T, ITU-T Recommendation H.245, 1995.

[ITU96a] *Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Non-Guaranteed Quality of Service*, ITU-T, ITU-T Recommendation H.322, 1996.

[ITU96b] *Data Protocols for Multimedia Conferencing*, ITU-T, ITU-T Recommendation T.120, 1996.

[ITU96c] *Generic Application Template*, ITU-T, ITU-T Recommendation T.121, 1996.

[ITU96d] *Network Specific Data Protocol Stacks for Multimedia Conferencing*, ITU-T, ITU-T Recommendation T.123, 1996.

[ITU96e] *Speech coders : Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*, ITU-T, ITU-T Recommendation G.723.1, March 1996.

[ITU98a] *Multipoint Communications Service - Service Definition*, ITU-T, ITU-T Recommendation T.122, 1998.

[ITU98b] *Generic Conference Control*, ITU-T, ITU-T Recommendation T.124, 1998.

[ITU98c] *Multipoint Communications Service Protocol Specification*, ITU-T, ITU-T Recommendation T.125, 1998.

[ITU98d] *Multipoint Application Sharing*, ITU-T, ITU-T Recommendation T.128, 1998.

[ITU98e] *Text Chat Application Entity*, ITU-T, ITU-T Recommendation T.134, 1998.

[ITU98f] *Security and Encryption for H-Series (H.323 and other H.245-based) Multimedia Terminals*, ITU-T, ITU-T Recommendation H.235, 1998.

[Irda] *IrDA Serial Infrared Data Link Standard Specifications*, Infrared Data Association, `http://www.irda.org/standards/specifications.asp`.

[Jacobson92] *vat - LBNL Audio Conferencing Tool*, Van Jacobson and Steve McCanne, `http://www-nrg.ee.lbl.gov/vat/`, 1992.

[Jacobson94] *Multimedia Conferencing on the Internet*, Van Jacobson, Tutorial Notes, ACM SIGCOMM 1994, London, September 1994.

[Kaashoek92] *Group Communication in Distributed Computer Systems*, Marinus Frans Kaashoek, 1992.

[Kirstein93] *Piloting of Multimedia Integrated Communications for European Researchers (MICE)*, Peter Kirstein, Mark Handley, and Angela Sasse, Proceedings of the INET'93 Conference, 1993.

[Kuespert96] *Generalizing Distributed Sensing Networks*, Jens Küspert and Dirk Kutscher, June 1996, Second International Airborne Remote Sensing Conference and Exhibition.

[Kutscher01a] *The Message Bus: Guidelines for Application Profile Writers*, Dirk Kutscher, Internet Draft draft-ietf-mmusic-mbus-guidelines-00.txt, Work in Progress, February 2001.

[Kutscher01b] *An Mbus Profile for Internet Appliance Control*, Dirk Kutscher and Jörg Ott, Internet Draft draft-kutscher-mbus-ipac-00.txt, Work in Progress, February 2001.

[Kutscher01c] *Requirements for Session Description and Capability Negotiation*, Dirk Kutscher, Jörg Ott, Carsten Bormann, and Igor Curcio, Internet Draft draft-ietf-mmusic-sdpng-req-01.txt, Work in Progress, April 2001.

[Kutscher03a] *Session Description and Capability Negotiation*, Dirk Kutscher, Jörg Ott, and Carsten Bormann, Internet Draft draft-ietf-mmusic-sdpng-06.txt, Work in Progress, March 2003.

[Kutscher03b] *Dynamic Device Access for Mobile Users*, Dirk Kutscher and Jörg Ott, Proceedings of the Eighth International Conference on Personal Wireless Communications, accepted for publication, 2003.

[Kutscher03c] *Service Location and Multiparty Peering for Ad-hoc Communication*, Dirk Kutscher and Jörg Ott, 2003, submitted for review.

[Kutscher98] *Engineers and Accidents: An Adaptable Conferencing Platform and its Application to Harbour Management*, Dirk Kutscher, Jörg Ott, and Carsten Bormann, European Conference Marcom'98, 1998.

[Lamport78] *Time, Clocks, and the Ordering of Events in a Distributed System*, Leslie Lamport, Communications of the ACM, Volume 21, Number 7, pages 558-556, July 1978.

[Leen02] *Expanding Automotive Electronic Systems*, Gabriel Leen and Donal Heffernan, IEEE Computer Magazine, 2002.

[MOST02] *MOST Specification Rev 2.2*, MOST Cooperation, available online at `http://www.mostnet.de/`, November 2002.

[McCanne95] *Vic: A flexible Framework for Packet Video*, Steve McCanne and Van Jacobson, ACM Multimedia'95, Volume 21, Number 7, pages 511-522, November 1995.

[McCarthy60] *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, John L. McCarthy, Communications of the ACM, Volume 3, Number 4, pages 184-195, 1960, `http://citeseer.nj.nec.com/mccarthy60recursive.html`.

[Mccanne98] *Scalable Multimedia Communication with Internet Multicast, Light-weight Sessions, and the Mbone*, Steven McCanne, CSD-98-1002, page 32, http://citeseer.nj.nec.com/mccanne98scalable.html , 1998.

[Meyer01] *Anruf- und Mediensteuerung auf der Basis von Mbus und SDPng*, Dirk Meyer, Diploma Thesis, Universität Bremen, March 2001.

[Microsoft95] *The Component Object Model Specification*, Microsoft Corporation and Digital Equipment Corporation, Draft Version 0.9, October 1995.

[Microsoft99] *IP Telephony with TAPI 3.0*, Microsoft Corporation, 1999, http://www.microsoft.com/windows2000/techinfo/howitworks/communications/telephony/iptelephony.asp .

[Moyer01] *A Protocol for Wide-Area Secure Networked Appliance Communication*, Stan Moyer, Dave Marples, and Simon Tsang, IEEE Communications Magazine, October 2001.

[NIST01] *Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, Federal Information Processing Standards Publication 197, November 2001.

[NIST95] *Secure Hash Standard*, U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, Federal Information Processing Standards Publication 180-1, April 1995.

[NIST99] *Data Encryption Standard (DES)*, U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, Federal Information Processing Standards Publication 46-3, October 1999.

[Naewe01] *Entwurf und Implementierung einer Mbus-basierten Infrastruktur zur Steuerung und Laufzeitanalyse von Mbus-Komponenten eines Kommunikationssystems*, Stefan Näwe, Diploma Thesis, Universität Bremen, February 2001.

[OMG03a] *Catalog of OMG CORBAservices Specifications*, Object Management Group, http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm.

[OMG03b] *CORBAFacilities Specifications*, Object Management Group, http://www.omg.org/technology/documents/formal/corbafacilities_specs.htm.

[OMG03c] *CORBA Component Model, v3.0*, Object Management Group, http://www.omg.org/technology/documents/formal/components.htm.

[OMG03d] *CORBA IIOP Specification*, Object Management Group, http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[OSGI03] *OSGi Service Platform Release 3*, Open Services Gateway Initiative, available online at http://www.osgi.org/, March 2003.

[Obraczka98] *Multicast Transport Protocols: A Survey and Taxonomy*, Katia Obraczka, January 1998.

[Oki93] *The Information Bus — an Architecture for Extensible Distributed Systems*, Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen, ACM SIGOPS'93 Conference, 1993.

[Ott00] *The Message Bus: A Platform for Component-based Conferencing Applications*, Jörg Ott, Dirk Kutscher, and Colin Perkins, Proceedings of CBG2000, the CSCW2000 workshop on Component-Based Groupware, Robert Slagter, Henri ter Hofte and Oliver Stiemerling, December 2000.

[Ott01] *An Mbus Profile for Call Control*, Jörg Ott, Dirk Kutscher, and Dirk Meyer, Internet Draft draft-ietf-mmusic-mbus-call-control-00.txt, Work in Progress, February 2001.

[Ott02a] *DTI: Desk-area Telephony Integration*, Jörg Ott, Dirk Kutscher, and Andreas Büsching, Final project report (unpublished), Bremen, November 2002.

[Ott02b] *SDPng: A New Session Description Language for Multimedia Conferencing*, Jörg Ott and Dirk Kutscher, Upperside Conference SIP 2002, Januray 2002.

[Ott03] *Presence Aggregation in Endpoints*, Jörg Ott and Dirk Kutscher, Upperside Conference SIP 2003, January 2003.

[Ott94] *Multicasting the ITU MCS: Integrating Point-to-Point and Multicast Transport*, Jörg Ott and Carsten Bormann, International Conference on Computer Communications and Networks (ICCN 94), 1994.

[Ott97] *A Multipoint Data Communication Infrastructure for Standards-based Teleconferencing Systems*, Jörg Ott, Phd Thesis, 1997.

[Ott99a] *Capability description for group cooperation*, Jörg Ott, Dirk Kutscher, and Carsten Bormann, Internet Draft draft-ott-mmusic-cap-00.txt, Work in Progress, June 1999.

[Ott99b] *Requirements for Local Conference Control*, Jörg Ott, Colin Perkins, and Dirk Kutscher, Internet Draft draft-ietf-mmusic-mbus-req-00.txt, Work in Progress, December 1999.

[Ousterhout94] *Tcl and the Tk Toolkit*, John K, Ousterhout, Addison-Wesley, March 1994.

[Park02] *Link Scoped IPv6 Multicast Addresses*, Jung-Soo Park and Myung-Ki Shin, Internet Draft draft-ietf-ipv6-link-scoped-mcast-02.txt, Work in Progress, October 2002.

[Parlay03] *Parlay 4.1 Specification*, The Parlay Group, available online at `http://www.parlay.org/specs/index.asp`, 2003.

[Perkins02] *Ad hoc On-Demand Distance Vector (AODV) Routing*, Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir R. Das, work in progress, November 2002.

[Perkins03] *RTP, Audio and Video for the Internet*, Colin Perkins, Addison-Wesley Professional, June 2003.

[Pollem00] *Entwurf und Implementierung eines ausbaufähigen Gateways zur Umsetzung unterschiedlicher Verbindungssteuerungsprotokolle für IP-Telefonie*, Niels Pollem, Diploma Thesis, Universität Bremen, January 2000.

[Pusateri00] *Distance Vector Multicast Routing Protocol*, Thomas Pusateri, Internet Draft draft-ietf-idmr-dvmrp-v3-10, Work in Progress, August 2000.

[RFC1112] *Host Extensions for IP Multicasting*, Stephen Deering, RFC 1112, August 1989.

[RFC1301] *Multicast Transport Protocol*, Susan M. Armstrong, Alan O. Freier, and Keith A. Marzullo, RFC 1301, February 1992.

[RFC1321] *The MD5 Message-Digest Algorithm*, Ronald L. Rivest, RFC 1321, April 1992.

[RFC1521] *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, Nathaniel S. Borenstein and Ned Freed, RFC 1521, September 1993.

[RFC1541] *Dynamic Host Configuration Protocol*, , Ralph Droms, RFC 1541, October 1993.

[RFC1831] *RPC: Remote Procedure Call Protocol Specification Version 2*, Raj Srinivasan, RFC 1831, August 1995.

[RFC1832] *XDR: External Data Representation Standard*, Raj Srinivasan, RFC 1832, August 1995.

[RFC1833] *Binding Protocols for ONC RPC Version 2*, Raj Srinivasan, RFC 1833, August 1995.

[RFC1889] *RTP: A Transport Protocol for Real-Time Applications*, Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson, January 1996, RFC 1889.

[RFC1890] *RTP Profile for Audio and Video Conferences with Minimal Control*, Henning Schulzrinne, January 1996, RFC 1890.

[RFC2032] *RTP Payload Format for H.261 Video Streams*, Thierry Turletti and Christian Huitema, RFC 2032, October 1996.

[RFC2104] *HMAC: Keyed-Hashing for Message Authentication*, Hugo Krawczyk, Mihir Bellare, and Ran Canetti, RFC 2104, February 1997.

[RFC2131] *Dynamic Host Configuration Protocol*, Ralph Droms, RFC 2131, March 1997.

[RFC2198] *RTP Payload for Redundant Audio Data*, Colin Perkins, Isidor Kouvelas, Orion Hodson, Vicky Hardman, Mark Handley, Jean-Chrysostome Bolot, Andres Vega-Garcia, and Sacha Fosse-Parisis, RFC 2198, September 1997.

[RFC2234] *Augmented BNF for Syntax Specifications: ABNF*, David H. Crocker and Paul Overell, RFC 2234, November 1997.

[RFC2254] *The String Representation of LDAP Search Filters*, Tim Howes, RFC 2254, December 1997.

[RFC2327] *SDP: Session Description Protocol*, Mark Handley and Van Jacobson, April 1998, RFC 2327.

[RFC2365] *Administratively Scoped IP Multicast*, Dave Meyer, RFC 2365, July 1998.

[RFC2373] *IP Version 6 Addressing Architecture*, Robert M. Hinden and Stephen E. Deering, RFC 2373, July 1998.

[RFC2429] *RTP Payload Format for the 1998 Version of ITU-T Rec. H.263 Video (H.263+)*, Carsten Bormann, Linda Cline, Gim Deisher, Tom Gardos, Christian Maciocco, Donald Newell, Jörg Ott, Gary Sullivan, Stephan Wenger, and Chad Zhu, RFC 2429, Oktober 1998.

[RFC2462] *IPv6 Stateless Address Autoconfiguration*, Susan Thomson and Thomas Narten, RFC 2462, December 1998.

[RFC2533] *A Syntax for Describing Media Feature Sets*, Graham Klyne, RFC 2533, March 1999.

[RFC2554] *SMTP Service Extension for Authentication*, J. Myers Crocker, RFC 2554, March 1999.

[RFC2581] *TCP Congestion Control*, Mark Allman, Vern Paxson, and W. Richard Stevens, RFC 2581, April 1999.

[RFC2608] *Service Location Protocol, Version 2*, Erik Guttmann, Charles Perkins, John Veizades, and Michael Day, RFC 2608, June 1999.

[RFC2609] *Service Templates and Service: Schemes*, Erik Guttmann, Charles Perkins, and James Kempf, RFC 2609, June 1999.

[RFC2610] *DHCP Options for Service Location Protocol*, Charles Perkins and Erik Guttmann, RFC 2610, June 1999.

[RFC2616] *Hypertext Transfer Protocol — HTTP/1.1*, Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henry Frystyk Nielsen, Larry Masinter, Paul Leach, and Tim Berners-Lee, RFC 2616, June 1999.

[RFC2730] *Multicast Address Dynamic Client Allocation Protocol (MADCAP)*, Stephen Hanna, Baiju V. Patel, and Munil Shah, RFC 2730, December 1999.

[RFC2766] *Network Address Translation - Protocol Translation (NAT-PT)*, George Tsirtsis and Pyda Srisuresh, RFC 2766, February 2000.

[RFC2806] *URLs for Telephone Calls*, Antti Vaha-Sipila, RFC 2806, April 2000.

[RFC2822] *Internet Message Format*, , Peter W. Resnick, RFC 2822, April 2001.

[RFC2824] *Call Processing Language Framework and Requirements*, Jonathan Lennox and Henning Schulzrinne, RFC 2824, May 2000.

[RFC2885] *Megaco Protocol version 0.8*, Fernando Cuervo, Nancy Greene, Christian Huitema, Abdallah Rayhan, Brian Rosen, and John Segers, RFC 2885, August 2000.

[RFC2908] *The Internet Multicast Address Allocation Architecture*, Dave Thaler, Mark Handley, and Deborah Estrin, RFC 2908, September 2000.

[RFC2974] *Session Announcement Protocol*, Mark Handley, Colin Perkins, and Edmund Whelan, RFC 2974, October 2000.

[RFC3010] *NFS version 4 Protocol*, Spencer Shepler and et al., RFC 3010, December 2000.

[RFC3096] *RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed*, Carsten Bormann et al., RFC 3095, July 2001.

[RFC3208] *PGM Reliable Transport Protocol Specification*, Tony Speakman, Jon Crowcroft, Jim Gemmell, Dino Farinacci, Steven Lin, Dan Leshchiner, Michael Luby, Todd L. Montgomery, Luigi Rizzo, Alex Tweedly, Nidhi Bhaskar, Richard Edmonstone, Rajitha Sumanasekera, and Lorenzo Vicisano, RFC 3208, December 2001.

[RFC3259] *A Message Bus for Local Coordination*, Jörg Ott, Colin Perkins, and Dirk Kutscher, RFC 3259, April 2002.

[RFC3261] *SIP: Session Initiation Protocol*, Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, and Eve Schooler, RFC 3261, June 2002.

[RFC3264] *An Offer/Answer Model with the Session Description Protocol (SDP)*, Jonathan Rosenberg and Henning Schulzrinne, RFC 3264, June 2002.

[RFC3550] *RTP: A Transport Protocol for Real-Time Applications*, Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson, July 2003, RFC 3550.

[RFC3551] *RTP Profile for Audio and Video Conferences with Minimal Control*, Henning Schulzrinne and Stephen Casner, July 2003, RFC 3551.

[RFC741] *Specifications For The Network Voice Protocol (NVP)*, Danny Cohen, RFC 741, November 1977.

[Renesse94] *The Horus System*, Robert van Renesse, Kenneth P. Birmann, Robert Cooper, Brad Glade Cooper, and Patrick Stephenson.

[Rivest97] *S-Expressions*, Ronald L. Rivest, Internet Draft draft-rivest-sexp-00.txt, Work in Progress, May 1997.

[Rosenberg02] *Models for Multi Party Conferencing in SIP*, Jonathan Rosenberg and Henning Schulzrinne, Internet Draft draft-ietf-sipping-conferencing-models-01.txt, Work in Progress, July 2002.

[Rossum02] *Python Reference Manual*, Guido van Rossum and Fred L. Drake, Release 2.2.2, available online at `http://www.python.org/`, October 2002.

[Saif2002] *Communication Primitives for Ubiquituous Systems or RPC Considered Harmful*, Umar Saif and David J. Greaves, Proceedings of 21st International Conference of Distributed Computing Systems (Workshop on Smart Appliances and Wearable Computing) , 2001.

[Scheifler92] *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, Xlfd*, Robert Scheifler, James Gettys, and David Rosenthal, Digital Press, February 1992.

[Schenker95] *Managing Shared Ephemeral Teleconferencing State: Policy and Mechanisms*, Scott Schenker, Abel Weinrib, and Eve Schooler, Internet Draft draft-ietf-mmusic-agree-00.txt, Work in Progress, July 1995.

[Schneier96] *Applied Cryptography*, Bruce Schneier, Edition 2, 1996.

[Schulzrinne95] *Dynamic Configuration of Conferencing Applications using Pattern-Matching Multicast*, Henning Schulzrinne, 1995.

[Schulzrinne98] *A Comparison of SIP and H.323 for Internet Telephony*, Henning Schulzrinne and Jonathan Rosenberg, 1998.

[Seifert94] *Multicast-Transportprotokoll Version 2*, Nils Seifert, Diploma Thesis at the Technische Universität Berlin, November 1994.

[Siemens03a] *OpenScape Whitepaper*, Siemens Information and Communication Networks, April 2003.

[Siemens03b] *Siemens Enterprise Workgroup Colloboration, Strategy and Development Directions*, Siemens Information and Communication Networks, Whitepaper, 2003.

[Singh00] *Interworking Between SIP/SDP and H.323*, Kundan Singh and Henning Schulzrinne, Proceedings of the 1st IP-Telephony Workshop (IPTel 2000), April 2000.

[Sisalem97] *The Multimedia Internet Terminal*, Dorgham Sisalem and Henning Schulzrinne, April 1997.

[Sun01] *Jini Architecture Specification Version 1.2*, Sun Microsystems, available online at http://java.sun.com/products/jini/, December 2001.

[Sun02] *Enterprise JavaBeans^{TM} Specification Version 2.1*, Sun Microsystems, Proposed Final Draft, available online at `http://java.sun.com/products/ejb/docs.html`, August 2002.

[Sun03] *The JAIN^{TM} APIs*, Sun Microsystems, available online at `http://java.sun.com/products/jain/api_specs.html`, 2003.

[Sun97] *JavaBeans^{TM} Specification Version 1.01*, Sun Microsystems, available online at `http://java.sun.com/products/javabeans/docs/spec.html`, August 1997.

[Sun99] *Java^{TM} Telephony Specification*, Sun Microsystems, 1999, `http://java.sun.com/products/jtapi/`.

[Syverson94] *A Taxonomy of Replay Attacks*, Paul Syverson, 1994, IEEE Computer Society Press.

[Szyperski99] *Component Software*, Clemens Szyperski, Addison-Wesley, 1999.

[TIBCO02] *TIBCO Rendezvous^{TM}*, TIBCO Software Inc., Software Release 7.1, `http://www.tibco.com/`, October 2002.

[Tanenbaum2002] *Distributed Systems*, Andrew S. Tannenbaum and Maarten van Steen, Prentice Hall, Upper Saddle River, NJ, 2002.

[Trossen00] *Scalable Group Communication in Tightly Coupled Environments*, Dirk Trossen, July 2000.

[UPnP03] *UPnP^{TM} Device Architecture 1.0*, UPnP Forum, Version 1.0.1, available online at `http://www.upnp.org/`.

[Waldo94] *A Note on Distributed Computing*, Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, Sun Microsystems Laboratories Technical Report SMLI TR-94-29, November 1994.

[Weihl93] *Transaction-Processing Techniques*, William E. Weihl.

[Weiser99] *The origins of ubiquitous computing research at PARC in the late 1980s*, Mark Weiser, John Seely Brown, and Rich Gold, IBM Systems Journal, Volume 38, Number 4, 1999.

[Yon03] *Connection-Oriented Media Transport in SDP*, David Yon, Internet Draft draft-ietf-mmusic-sdp-comedia-05.txt, Work in Progress, March 2003.

[Zimmermann99] *Wireless networked digital devices: A new paradigm for computing and communication*, Thomas G. Zimmermann, IBM Systems Journal, Volume 38, Number 4, July 1999.

Appendix A

Colophon

This document has been written using *DocBook XML 4.1.2*, an XML document type.[1] The text has been edited using Lennart Staflin's wonderful *PSGML mode*[2] for the Emacs editor.

The PDF document has been produced using Ramon Casellas' *DB2LaTeX* XSL transformation stylesheets, [3] the *xsltproc* tool of the GNOME project's *libxslt*[4] and *pdflatex* of the teTeX distribution.[5]

The message flow diagrams have been created using the *msgflow tools*, a simple XML DTD and a corresponding XSLT stylesheet I have developed for this purpose. The *msgflow* stylesheet generates SVG graphics that have been translated to PNG using the *Batik* toolkit of the Apache XML project.[6] The source code excerpts have been typeset with the *GNU a2ps* PostScript filter[7], and some of the figures in this document have been prepared with *Microsoft Powerpoint*.[8]

The printed copies have been produced as a DIN A5 booklet using the PSUtils [9], a collection of utilities for manipulating PostScript documents. The original PDF file has been prepared with LaTeX's *twoside* option (different margin sizes for even and odd pages). The PDF file has been converted to PostScript using Adobe Acrobat 6.0 [10]. The resulting PostScript file has then been processed as follows:

[1] `http://www.oasis-open.org/docbook/xml/`

[2] `http://www.lysator.liu.se/projects/about_psgml.html`

[3] `http://db2latex.sourceforge.net/index.html`

[4] `http://xmlsoft.org/XSLT/index.html`

[5] `http://www.tug.org/teTeX/`

[6] `http://xml.apache.org/batik/`

[7] `http://www.infres.enst.fr/~demaille/a2ps/`

[8] `http://www.microsoft.com/office/powerpoint/default.asp`

[9] `http://knackered.knackered.org/angus/psutils/`

[10] `http://www.adobe.com/products/acrobat/`

```
psbook thesis.ps | pstops '2:0L@.7(21cm,0)+1L@.7(21cm,14.85cm)' |
    pstops '2:0(0.05cm,0cm),1' | pstops '2:0,1U(21cm,29.7cm)'
    >thesis-book-up.ps
```

The thesis has been printed with an hp color LaserJet 4600 dtn. The first `pstops` step is the actual `psnup` operation, the seconds `pstops` step was necessary to compensate a horizontal displacement caused by the printer and the last `pstops` step is performed to rotate the backside pages for the duplex unit.

The thesis has been published online and is available through the E-LIB system at the Staats- und Universitätsbibliothek Bremen (`http://elib.suub.uni-bremen.de/`).