MONITORING MIDDLEWARE FOR DISTRIBUTED APPLICATIONS

A THESIS
SUBMITTED TO THE SCHOOL OF COMPUTING SCIENCE
OF THE UNIVERSITY OF NEWCASTLE UPON TYNE
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Simon Parkin
November 2007

NEWCASTLE UNIVERSITY LIBRARY
----
206 53355 3

Thesis L8652

ABSTRACT

MONITORING MIDDLEWARE
FOR DISTRIBUTED APPLICATIONS

Simon Parkin
Ph.D. in Computing Science
Supervisor: Dr. Graham Morgan
November 2007

With growing maturity Internet services are proving integral to the provision of computer services. To provide consistent end-user experiences these services are increasingly augmented with some notion of 'Quality-of-Service' (QoS), which typically requires the management of computing resources to maintain a predictable level of service performance.

It is difficult to guarantee consistent service provision in dynamic and open environments such as the Internet. However service monitoring can be used to inform compensatory actions by collecting meaningful service performance data from strategic points in an active service environment.

Due to the unpredictable nature of the Internet distributed monitoring mechanisms face challenges with respect to the various communication protocols, application languages, and monitoring requirements associated with a service environment. With the growing popularity of Internet services creation of monitoring solutions on a per-service basis becomes time-consuming and misses opportunities to re-use existing logic. Ideally monitoring solutions would be domain-agnostic, automatically generated and automatically deployed.

This thesis progresses these ambitions by providing a generic, distributed monitoring and evaluation framework based on Metric Collector (MeCo) components. These components can transparently gather measurement data across a range of service technologies as used within E-Commerce service environments. MeCo components form part of a framework which can interpret Service Level Agreements (SLAs) to automatically provide tailored service monitoring.

The evaluation paradigms of the MeCo Framework are re-appropriated for use in Distributed Virtual Environments (DVEs). Quantifiable QoS requirements are established for Interest Management mechanisms (which limit message production based on object localities within a DVE). These are then incorporated into a DVE Simulator application. This application allows DVE application developers to evaluate Interest Management configurations for their suitability. Extensions to the DVE Simulator are exhibited in the Evolutionary Optimisation Simulator (EOS), which provides automated optimisation capabilities for DVE configurations through utilisation of genetic algorithm techniques.

// Acknowledgements

I would like to thank my mother Marilyn and my brother Ross for being there for me regardless

I would like to thank my supervisor, Dr. Graham Morgan, for his immeasurable advice and encouragement throughout my studies

I would like to thank my close friends for reminding me of my humanity in so many ways

Table of Contents

ABSTRACT i

Acknowledgements iii

1. Introduction 1

   1.1 Internet Services 1

   1.2 Competition & Quality 2

   1.3 Monitoring 3

   1.4 Evaluation 4

   1.5 The Heterogeneous Nature of the Internet 5

   1.6 Contribution of Thesis 6

   1.7 Summary 8

2. Background 9

   2.1 Service Provision and Consumption over the Internet 9

      2.1.1 Overview of Service Provisioning 9

      2.1.2 Realising the Provider/Consumer Relationship 11

   2.2 Observing the Quality of Service Interactions 12

      2.2.1 Quality of Service 13

      2.2.2 Monitoring of Service Quality 15

      2.2.3 Monitoring Concerns 16

      2.2.4 Monitoring Service Performance in the Internet 17

      2.2.5 Monitoring Service Traffic 19

      2.2.6 Creating an Overview of System Performance – Gathering Monitoring Information 20

      2.2.7 Making Monitoring Information Meaningful – Electronic Contracts 21

      2.2.8 QoS Evaluation – Accountability 23

   2.3 Message Dissemination Mechanisms 24

      2.3.1 The Client/Server Model & Socket Layer 24

      2.3.2 Remote Procedure Calls 25

      2.3.3 Web Services 26

2.3.4 Message-Oriented-Middleware (MOM) 28
2.3.5 Summary 30
2.4 A Different Service Domain – Distributed Virtual Environments 31
2.4.1 Introduction 32
2.4.2 Interest Management 33
2.4.3 Message Exchange 34
2.4.4 Missed Interactions 36
2.4.5 Avoiding Missed Interactions 38
2.5 QoS Provision – Case Studies 39
2.5.1 E-Commerce Systems 40
2.5.2 Distributed Virtual Environments 43
2.6 Related Work 45
2.6.1 E-Commerce & Web Services 45
2.6.2 Commercial Research 47
2.6.2.1 Pruyne 47
2.6.2.2 QuO 48
2.6.2.3 WSLA Monitoring & Evaluation Framework 49
2.6.2.4 Business Management Platform (BMP) Agent Network 52
2.6.2.5 QoS Monitoring Framework for Traffic Engineering in IP Differentiated Services 55
2.6.3 Academic Research 57
2.6.3.1 Nahrstedt et al 57
2.6.3.2 Smart Proxies 58
2.6.3.3 CQoS 59
2.6.3.4 SLAng 60
2.6.4 SLA Monitoring Requirements 62
2.6.4.1 Contractual Heterogeneity 62
2.6.4.2 Domain Heterogeneity 63
2.6.4.3 Accommodation of Enabling Technologies 64

2.6.4.4 Scalability towards Participant Entities and Service Contracts ... 65

2.6.4.5 Transparent Deployment and Operation ... 66

2.6.4.6 Ease of Deployment and Modularity ... 67

2.6.5 Distributed Virtual Environments ... 68

2.6.6 Summary ... 68

2.7 Outline of Goals ... 70

2.8 Summary ... 71

3. E-Commerce ... 73

3.1 Introduction ... 73

3.2 SLA Monitoring Architecture ... 75

3.2.1 Monitoring Architecture ... 75

3.2.2 Scalability Considerations ... 77

3.2.3 Deployment Considerations ... 79

3.2.4 Heterogeneity Considerations ... 80

3.3 Implementation ... 81

3.3.1 Overview ... 81

3.3.2 Implementation Assumptions ... 83

3.3.3 Metric Collector (MeCo) Interceptors (Provider-Side) ... 85

3.3.3.1 MeCo Interceptor Implementations ... 87

3.3.3.2 Provider Environment ... 88

3.3.3.3 Provider-Side MeCo Deployment and Initialisation ... 90

3.3.4 Metric Collector (MeCo) Probe ... 94

3.3.4.1 Probe Descriptors ... 96

3.3.5 Messaging Service ... 97

3.3.5.1 Event Notification within the Messaging Service ... 98

3.3.5.2 Implementing the Messaging Service ... 99

3.3.6 Measurement Service ... 102

3.3.6.1 The Contract Manager ... 103

3.3.6.2 Measurement Service Contract Configuration ... 105

3.3.6.3 The Measurement Service Configuration File 108

3.3.6.4 Measurement Service Visual Component 109

3.3.6.5 Additional Scalability Measures in the MeCo Framework.. 111

3.4 Satisfaction of Requirements 114

3.4.1 Contractual Heterogeneity.. 115

3.4.2 Domain Heterogeneity 116

3.4.3 Accommodation of Enabling Technologies 116

3.4.4 Scalability towards Participant Entities and Service Contracts 117

3.4.5 Transparent Deployment and Operation 118

3.4.6 Ease of Deployment and Modularity 118

3.5 Summary 119

4. Distributed Virtual Environments... 122

4.1 Re-Appropriating the MeCo Framework 122

4.1.1 Achievements in SLA Monitoring 122

4.1.2 A Different Domain – Distributed Virtual Environments................. 123

4.2 Implementation... 124

4.2.1 Implementation Assumptions... 124

4.2.2 The DVE Simulator.. 125

4.2.3 Object Classes 126

4.2.4 Simulating Object Behaviour 128

4.2.5 Modeling Resource Constraints 129

4.2.6 The DVE Simulator Interface 130

4.2.7 DVE Configuration 131

4.2.8 Interaction Detection 133

4.2.9 DVE Simulator Interface... 134

4.3 DVE Simulator – Evolutionary Component 136

4.3.1 Evolutionary Optimisation Simulator Overview............................... 137

4.3.2 Evolutionary Optimisation – Fitness Function................................. 138

4.3.3 Evolutionary Optimisation – Crossover, Mutation and Elitism......... 139

4.3.4 The Evolutionary Optimisation Algorithm 140
4.3.5 Evolutionary Optimisation Simulator Interface 142
4.4 Application of DVE QoS Measures 144
4.4.1 Assumptions 145
4.4.2 Monitoring DVE Performance 146
4.4.3 Monitoring DVE Provision 148
4.4.4 Augmenting the DVE Evaluation Framework 150
4.4.5 Different DVE Configurations 151
4.5 Summary 152
5. Experimental Results 154
5.1 MeCo Framework 154
5.1.1 Test Configuration 154
5.1.2 Experiments 156
5.1.2.1 Deployment Profiling – Provider-side MeCo 156
5.1.2.2 Deployment Profiling – Measurement Service 157
5.1.2.3 Operational Performance Profiling – Multiple Services 158
5.1.2.4 Accuracy Testing – Correctness of Measurements 160
5.1.3 Experimental Results 161
5.1.3.1 Deployment Profiling – Provider-side MeCo 161
5.1.3.2 Deployment Profiling – Measurement Service 162
5.1.3.3 Operational Performance Profiling – Multiple Services 165
5.1.3.4 Accuracy Testing – Correctness of Measurements 169
5.1.4 Performance Analysis 170
5.2 DVE Simulator 172
5.2.1 Experiments 172
5.2.2 Experimental Results 174
5.2.2.1 Heartbeat Interval 175
5.2.2.2 Object Aura Size 178
5.3 Evolutionary Optimisation Simulator 180

5.3.1 Simulator Settings 181
5.3.2 Population Size 182
5.3.3 Mutation 184
5.3.4 High-Frequency Message Interval 187
5.3.5 Different Scenarios 188
5.4 Summary 191
6. Conclusion 192
6.1 Thesis Summary 192
6.2 Contribution of Thesis 194
6.3 Future Work 195
6.4 Summary 198
7. Bibliography 199
8. Appendix A – MeCo Installation Guide 209
8.1 Provider-Side MeCo Deployment 209
8.1.1 Apache Axis Configuration 209
8.1.2 JBoss Configuration 209
8.2 Measurement Service Deployment 211
8.2.1 Measurement Service Installation Files 211
8.2.2 Additional Files 212
8.2.3 Using the Measurement Service 213
8.3 SLAng Contract Configuration 214
8.3.1 Service Clients 215
8.3.2 Provider Definition 215
8.3.3 Contract Schedule 215
8.4 Example Configuration Files 216
8.4.1 measurement-service.xml 216
8.4.2 EJB Probe Configuration – Fibonacci_EJB.wsdl 217
8.4.3 Probe Descriptor – Fibonacci EJB Service 218
9. Appendix B – Sample SLAng Contract File 219

Table of Figures

Figure 1 The basic elements of a computer network (computer users, network elements such as routers, and application servers) 9
Figure 2 Provider/consumer relationship 11
Figure 3 Communication patterns: (a) direct addressing; (b) service endpoints; (c) indirect messaging 11
Figure 4 How different machines make up a network 13
Figure 5 Low-level monitoring techniques: (a) local to an end-system device; and (b) local to a router 17
Figure 6 Application-level monitoring techniques 18
Figure 7 (a) passive and (b) active monitoring techniques 19
Figure 8 Collection of monitoring data at a centralised location 20
Figure 9 Gathering service-oriented data 21
Figure 10 Socket communication 24
Figure 11 Remote Procedure Calls (RPCs) 25
Figure 12 Web services provide an abstraction layer between the application client and the application code [Snell02] 27
Figure 13 How Message-Oriented Middleware (MOM) may be deployed [Haefal01] 28
Figure 14 (a) one-to-one and (b) publish/subscribe MOM communication 29
Figure 15 Areas-of-influence 33
Figure 16 An object crossing region boundaries 35
Figure 17 Example of a missed interaction 36
Figure 18 Illustration of a binary session involving two objects with auras 37
Figure 19 Communication pattern in an E-Commerce environment 40
Figure 20 Various approaches to QoS integration in middleware 42
Figure 21 Machine users within a virtual environment 43
Figure 22 Peer-to-peer (P2P) communication pattern in a DVE 43
Figure 23 Centralised communication pattern in a DVE 44
Figure 24 Applying a generic contract representation to dissimilar business relationships 62

Figure 25 Applying a monitoring framework to different communication and application technologies........64
Figure 26 Service clients dynamically entering and leaving a service environment...65
Figure 27 The system components that must be considered when deploying a monitoring infrastructure........67
Figure 28 How contracts bind service participants........73
Figure 29 Enforcing an electronic contract........74
Figure 30 Architecture for the unilateral monitoring and enforcement of inter-organisational SLAs........75
Figure 31 SLA monitoring architecture........81
Figure 32 Provider-side MeCo placement........85
Figure 33 Implementation of Provider-side MeCo's........87
Figure 34 MeCo Provider Environment........88
Figure 35 How the Measurement Service configures the Provider Environment.......92
Figure 36 How the Provider Environment is configured based on Measurement Service initialisation actions........93
Figure 37 Sample Probe Descriptor File........96
Figure 38 Probe initialisation process........97
Figure 39 How measurement data is processed between the Provider-side MeCo and the Measurement Service........99
Figure 40 The Measurement Service and its sub-components........102
Figure 41 The GUI chart window........109
Figure 42 The GUI Violation Data window........110
Figure 43 The GUI Message Contents window........111
Figure 44 Staged and coupled metric processing........112
Figure 45 Generic object pooling........113
Figure 46 Users interacting within a Distributed Virtual Environment (DVE).........123
Figure 47 Different forms of simulated object movement........127
Figure 48 Influencing object movements by assigning targets........128
Figure 49 The DVE Simulator interface........130
Figure 50 Interaction detection logic........133
Figure 51 A chart of true aura overlaps produced by the DVE Simulator........134
Figure 52 A chart of high-frequency message exchange produced by the DVE Simulator........135

Figure 53 EOS internal algorithm logic 141
Figure 54 The Evolutionary Optimisation Simulator interface 142
Figure 55 Chart of aura-size against heartbeat interval 143
Figure 56 Enlarged segment of aura-size against heartbeat interval chart 143
Figure 57 Chart of missed interactions against messages 144
Figure 58 How DVE monitoring would work in practice 147
Figure 59 How DVE provision would be monitored 148
Figure 60 Incorporating contract evaluation into the DVE monitoring infrastructure 150
Figure 61 Memory usage during deployment of the Measurement Service 163
Figure 62 Memory usage in the Measurement Service 166
Figure 63 Chart update duration during operation of the Measurement Service 168
Figure 64 Average server response time measurements with introduced delays 169
Figure 65 Round-trip-time measurements with introduced delays 169
Figure 66 Object and target distribution within a sample DVE simulation 174
Figure 67 Number of messages sent as function of heartbeat interval 175
Figure 68 Quotient of missed interactions as function of heartbeat interval 176
Figure 69 Quotient of partial missed interactions as function of heartbeat interval 177
Figure 70 Number of messages sent as function of aura size 178
Figure 71 Quotient of complete missed interactions as function of aura size 179
Figure 72 Quotient of partial missed interactions as function of aura size 180
Figure 73 Chart of generations against average fitness for different population sizes 183
Figure 74 Chart of generations against variance for different population sizes 184
Figure 75 Chart of generations against average fitness for different mutation levels 185
Figure 76 Chart of generations against variance for different mutation levels 186
Figure 77 Chart of generations against average fitness for different high-frequency messaging intervals 187
Figure 78 Chart of generations against variance for different high-frequency messaging intervals 188
Figure 79 Chart of generations against average fitness for different DVE scenarios 189
Figure 80 Chart of generations against variance for different DVE scenarios 190

1. Introduction

This work describes a generic, distributed SLA monitoring framework and its constituent features. This framework is built around the principles of application provision over the Internet. As such it is necessary to first discuss how people use the Internet, and how the experiences of users and service providers alike can be improved through service monitoring.

1.1 Internet Services

The Internet and network services have become an important part of many people's lives, influencing how they work as well as how they choose to spend their free time. Computers are commonplace in the developed world, and the prevalence of networked computer applications is only set to keep growing. People choose to use the Internet for many different tasks. These include accessing e-mails, transferring files (perhaps within the workplace, or through a file-sharing application), and generally reading the countless millions of web pages currently available on the World-Wide Web.

One aspect of the Internet (and networked environments in general) that is gaining in importance is that of providing network services to users. Here a network-accessible artefact allows a computer-user to essentially delegate some task to an application residing on another machine elsewhere. Examples of Internet services include currency-conversion applications, RSS feeds [Rsswiki], and Internet-based multiplayer games (such as Unreal Tournament [Unreal] and World of Warcraft [Wow]). These examples follow a model of providing something to the end-user that they have asked to have provided to them (i.e. that they have 'requested'). For example, in the case of currency-conversion applications, it is feasible that someone can carry out their own research into current currency-conversion rates. However, this could be negated if there were an Internet service available with access to up-to-date conversion rates, and which is accessible directly from the user's home computer. Such examples of effort being moved to a service provider illustrate why Internet services are becoming ever-more widespread.

1.2 Competition & Quality

With respect to networked services the growth of the Internet is evident in many ways. More people are using networked services, and more services are appearing to both meet existing needs and provide new ways of using the Internet. With this, competition between similar services becomes more of an inherent factor in how successful networked services are.

Modern networked service applications can adapt to their environment in ways that preserve transparent, simple and problem-free interactions with their respective users. Service developers are finding that they have to offer greater consistency guarantees to prospective users in order to remain competitive. A potential customer might hesitate to use an online store again if their order takes too long to be processed, just as a computer games enthusiast might choose to go elsewhere if their connection to a game server is sporadically slow and prone to failure. With this premise in mind, service developers need to be able to ensure the 'quality' of their offerings. Provision of a service is meaningless if users cannot connect to the associated web server, hold a connection with it, and conduct meaningful and timely communications with that server.

Service developers need a means to guarantee consistency in the way that their services behave, while ensuring that the experiences of end-users reflect their intentions. There has been great interest in how the underlying resources supporting Internet services can be manipulated in order to provide for greater 'Quality-of-Service' (QoS). This umbrella term covers a great number of network and service management mechanisms, such as the allocation of computing (i.e. processing) resources to match the demands of incoming service requests. Another application of QoS is the provision of additional service extensions, for example to provide security and transactional support (for when business partners request said facilities).

It is meaningless to define the quality of a service in arbitrary terms, and just as meaningless to ignore the expectations of service participants when doing so. The QoS definition of a particular network service must be meaningful and serve a purpose. The QoS associated with a service typically defines quantifiable characteristics that represent the desired performance attributes of the service. These may include qualities such as request completion latency (how long service requests take to go from their point of origin, to the target server, and back again as response

messages), server availability etc. Such characteristics may be defined by interacting parties (in those cases where a direct business relationship has been agreed between them). Alternatively they may be dictated by a service developer before deployment to provide an idealised, predictable, and consistent model of behaviour that it is believed the service must adhere to. Service providers can then use QoS definitions to manage their computational resources effectively.

1.3 Monitoring

To be able to regulate the quality of a service in a large and unpredictable environment such as the Internet, there is a need to know how well it is actually performing in relation to its ideal (i.e. expected) performance. This knowledge allows for informed action to be taken when there is a need to compensate for shortcomings in service provision. For example additional server machines may be added to a server cluster to meet increased service demand. As another example, without knowing specifically what is wrong with a service that is serving requests slower than expected, a service provider may believe that processing resources are expended. They may then choose to add a single extra server at a time to their server cluster to provide additional processing power and alleviate request loads. However, it could be the case that service monitoring processes indicate that two servers must be added to the machine cluster in order to compensate for the lack of processing capabilities. This simple example illustrates that without an accurate view of how the end-to-end system is performing any compensatory actions are essentially reliant on guesswork, and can potentially confuse or compound already inconsistent service behaviour.

Service monitoring in a large networked environment such as the Internet cannot rely on human perception alone. So that a measured course of action can be prescribed service monitoring must be carried out, in such a way that it quantifiably represents quantifiable aspects of the service behaviour (with respect to both the provider and the service consumer, and even the network that connects them).

The performance of participants within a service environment is monitored to determine the levels of QoS provision evident within its bounds. In the simplest cases low-level characteristics may be monitored, such as network latency and jitter (the variance in latency over time). As services and service monitoring become more complex, the same is required of the processes that observe them. In some cases

monitoring logic must be capable of providing a composite view of the system environment and the performance of interacting parties in a way that identifies the actions of individual entities in the network. This is particularly important with respect to accountability. For instance, a situation could arise wherein a service provider is found to be processing client requests particularly slowly, but here the blame may lie with another client that is overloading the server with requests and taking up more computational resources than they were allocated.

If monitoring data is to be gathered from a service environment, it must be useful. It is ineffectual to try to determine the inner workings of an application server by directly observing a client machine. Specialised monitoring techniques need to be employed across the network, potentially alongside or even within the machines that are interacting in an active service environment. Use of such techniques allows monitoring mechanisms to gather data directly from within network entities (for example by inserting monitoring agents within an application server), or to infer information through other less direct methods (such as sending fabricated requests to a server and observing their behaviour across the network).

1.4 Evaluation

When monitoring data has been collected it must be evaluated in a meaningful way to determine if service performance is as expected. This could be carried out local to each monitoring component, for example within an application server or at a router node. Alternatively data could be gathered at a centralised point for post-processing, perhaps if a number of distributed monitoring components are actively collecting monitoring data within an observed service environment. Processing such as this would typically involve collation of monitoring data so that it represents the behaviour of the participating network entities in relation to each other. This would more accurately reflect overall system performance. Evaluation processes can potentially be automated, so that measurement data is automatically analysed to determine whether any aspects of proposed service QoS are not as they should be.

Some services incorporate tightly-coupled interactions between participating organisations. This can be seen in E-Commerce services, for instance between an electronic bookstore provider and the operators of an online publications warehouse. In these cases the QoS expectations pertaining to each party are typically agreed upon

by both parties, thereby constituting their performance obligations. These obligations can be recorded in an electronic contract referred to as a Service Level Agreement (SLA). An SLA provides a machine-readable electronic representation of the QoS attributes for a particular service, as well as details of the participant organisations and how QoS obligations relate to them. For instance, a warehouse provider may agree to provide 99.999% uptime on their servers, while a bookstore operator might agree to only forward requests for stock information between the hours of 9 a.m. and 5 p.m. Machine-readable representations of these obligations would be recorded in the associated SLA. Identifying information about the warehouse providers and bookstore operators would also be included, such as their company names and website addresses.

1.5 The Heterogeneous Nature of the Internet

Testament to the open standards of the Internet, not every service provided over the Internet is the same. Services may rely upon application logic written in a particular language, delivered to consumers using a particular communication protocol. Even the way entities interact with each may differ across individual services (for instance if participants are providing services to each other at the same time). Interacting parties may want specific measurements to be made at specific locations within the service environment. However just as QoS requirements can change dramatically from service to service, similarities may also be found. Monitoring and evaluation mechanisms cannot be effective without first taking into account the characteristics of the individual systems they are applied to.

Monitoring and evaluation logic may be written on a per-service basis, but this would require a great deal of time. It would prove more efficient to re-use existing logic, especially considering that many services share commonalities. The capacity to apply monitoring and evaluation components over different enabling technologies would also limit the need to re-write code on a case-by-case basis. Care must be taken however to ensure that monitoring constructs are deployed in such a way that they do not interfere with the services they are observing. Otherwise the measurement data being collected would potentially be inaccurate.

1.6 Contribution of Thesis

Existing QoS monitoring and evaluation software for Internet applications suffers from a number of limiting factors. Each monitoring system tends to be domain-specific, applicable only to certain types of Internet applications. Monitoring frameworks also require manual effort to generate monitoring logic to measure service characteristics, and further manual effort to deploy monitoring components within an observed service environment.

Monitoring software for Internet applications should ideally be suitable for all service domains. It should also be capable of automatically generating monitoring logic as required by an observed service. Deployment of monitoring components should also be automated, to alleviate further effort on the part of interacting parties and monitoring agents.

This thesis attempts to make progress on the journey from "what monitoring software is now" to "what monitoring software should be". The work presented in this thesis progresses a monitoring infrastructure applicable to various types of distributed applications, particularly E-Commerce services. This infrastructure aims to accommodate a variety of monitoring requirements and SLA evaluation engines, thereby providing applicability to a wider range of service environments.

The monitoring infrastructure also aims to reduce deployment effort. The capacity to re-use existing monitoring and evaluation logic removes the need to re-write proven code on a per-service basis. Furthermore, by considering the respective needs of interacting parties, the monitoring infrastructure aims to minimise any disruption that may be causes within an observed service environment during its deployment.

This thesis develops a logical distinction between monitoring and evaluation components, allowing them to be considered and managed effectively in isolation. Based upon the monitoring and evaluation model implemented within the monitoring infrastructure, further work is conducted to exemplify this distinction. Work is carried out to illustrate how evaluation processes can be developed and extended for application to different service domains. A means of evaluating Distributed Virtual Environments (or DVEs), most commonly encountered within Massively Multiplayer Online Games (MMOGs), is developed. Further work describes a suite of DVE simulation components capable of automatically configuring DVEs for optimum performance.

The content of this thesis is arranged as follows: Chapter 2 describes services within large networked environments such as the Internet, while also describing the enabling technologies and the paradigms that control inter-organisational interactions and behaviour patterns. The monitoring and evaluation requirements of E-Commerce applications and Distributed Virtual Environments are outlined prior to detailed work in these two domains. Examination of associated research is also presented to clarify the objectives of the applications developed herein.

Chapter 3 describes the implementation of the Metric Collector (MeCo) Framework [MorganIfip05]. This is a distributed, modular monitoring & evaluation infrastructure applicable to E-Commerce applications such as Web services. Chapter 4 applies the evaluation paradigm developed in Chapter 3 to the domain of Distributed Virtual Environments (DVEs), utilising these constructs to develop processes for the evaluation of DVE performance. This then acts as the basis for a suite of DVE simulation tools comprising the DVE Simulator [Parkin06] and Evolutionary Optimisation Simulator [Parkin07].

Chapter 5 presents performance results for experiments applied to both the monitoring infrastructure and the DVE simulation suite. These tests assess the capabilities of the respective applications, and are ultimately used to illustrate how well they achieve their respective goals. Chapter 6 concludes this work by providing a synopsis of the thesis, while discussing potential avenues for future research. There are additional Appendices, one of which details how to deploy and manage the Metric Collector Framework in practice.

1.7 Summary

- Networked services are becoming more popular, both commercially and for recreational purposes.

- To guarantee that distributed services reach users in their intended form over unpredictable networks, there is a need to compensate for fluctuations in the availability of underlying resources. Defining the Quality of Service (QoS) for a given application provides a means to describe quantifiable service characteristics that should be adhered to in order to guarantee meaningful service usage and provision.

- For QoS provision mechanisms to react appropriately to fluctuations in resource availability there is a need to monitor service provision, primarily to detect inconsistencies in performance. To guarantee accuracy, monitoring processes may need to collect and collate information from service participants, and perhaps from other components across the breadth of network that connects them.

- In complex scenarios such as E-Commerce service environments there is a need to define electronic contracts detailing per-participant service obligations. These electronic contracts are typically referred to as Service Level Agreements (SLAs).

- The Internet provides numerous challenges to QoS provision and service monitoring. Monitoring and evaluation processes must be scalable, transparent, and heterogeneous in order to be successful. There is also a need to provide simple deployment and logic reuse in light of the growing demand for distributed services.

- Portions of the work described in this thesis have been published in internationally-recognised conference proceedings [MorganIfip05, Parkin06, Parkin07].

2. Background

There is a need for a cross-platform, application-agnostic monitoring and evaluation framework for distributed Internet services. Within this it is necessary to identify the environments within which such a system may be deployed. This includes examination of how elements within an observed service environment can interact, and the nature of the technologies that enable these elements to participate in meaningful interactions. These investigations will be furthered with an examination of the monitoring and evaluation requirements that a service environment may have, supported by an accompanying discussion of related work (both commercial and academic).

2.1 Service Provision and Consumption over the Internet

Before approaching the monitoring of Internet services it is first necessary to examine how service users and providers behave within a service environment, and with this how elements within a network achieve meaningful communication.

2.1.1 Overview of Service Provisioning

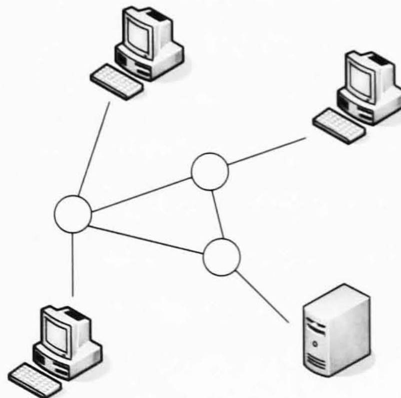


Figure 1 The basic elements of a computer network (computer users, network elements such as routers, and application servers)

The Internet (or any other large network or network-of-networks) essentially acts as a means to connect potentially disparate computer users via their machines so that they may communicate with each other. People may explicitly communicate with a specific person (as in the plethora of personal messenger applications [Yahoomsg, Microsoftmsg]). Alternatively they may choose to present information as web pages, allowing those users with access to a web page the choice of viewing it.

There are users who turn to the Internet in search of specific services. These people are in essence looking for a web page or network-accessible computer program containing a functional component capable of carrying out a particular operation, or solving a particular problem, for which they have a need. Examples of Internet services include up-to-the-minute currency-converter websites and daily weather-reports. These essentially constitute a piece of logic, accessible over a network, which can take input from users with access to the same network, perform some amount of processing on that input data, and produce a result relevant to its context (for instance telling a user how much a specific amount of money is worth in another currency).

In this sense, some of the communication patterns that come to exist across the Internet can be regarded as provider/consumer relationships. Herein a person (or an organisation) makes available a service created to solve a specific problem or address a need for a specific piece of processing logic. Other users are then able to issue a request for the service, with the request being directed towards the target service. The request is subsequently processed by the service, and a suitable response is returned to the user that sent the original request. As an example, a user wanting to carry out a currency conversion would query a specialised website with a request describing a specific amount of money and specific currency types for the original and converted currencies. The server hosting the website and its associated logic would carry out the currency conversion while the user waits for the response. Once the conversion has been completed, the result of the conversion (in this case, how much the amount of the original currency is worth in the conversion currency specified) will be returned to the user over the network.

It is possible for services to form chains. One such permutation is when a service must query another service for the additional information it needs to complete a user's request (for example when a currency-conversion service needs to query a financial markets service to determine what the current exchange-rate for a particular currency is).

2.1.2 Realising the Provider/Consumer Relationship

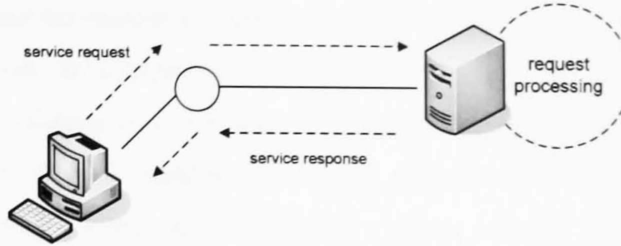


Figure 2 Provider/consumer relationship

Just as there are different communication patterns that users can enact when interacting over large networks, there are also different levels of complexity at which they can communicate. These range from the physical network itself (by directly controlling the data that is sent across the network) up to levels of abstraction where the details of how data is transmitted are delegated to underlying communication protocols, completely hidden from the user (and potentially even the end-user application). Here communication processes are driven by high-level concepts more directly understood by the user (such as "find a book about QoS provision which is for sale and can be delivered by tomorrow").

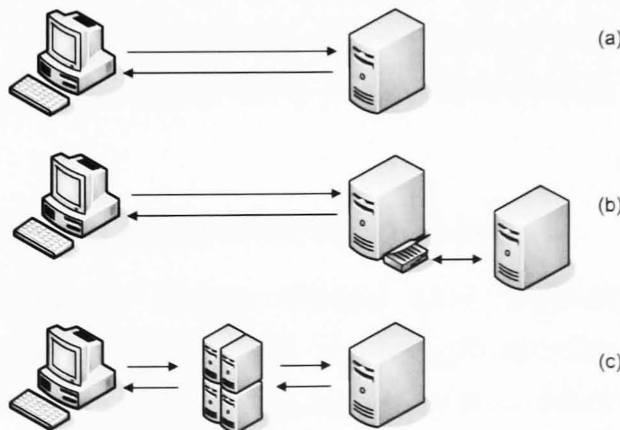


Figure 3 Communication patterns: (a) direct addressing; (b) service endpoints; (c) indirect messaging

At the lowest level (Figure 3a), communicating machines send data across a network (in what are called 'packets') to a specific port on a target machine. This data traverses other network elements such as routers, which direct data depending on where it is in relation to its intended destination. When the packets reach their destination they are recombined to mirror the original form of the data.

Networked services can alternatively be made available at 'service endpoints' (Figure 3b) where the translation to a specific machine port is hidden from the user behind a service descriptor (which translates a service address to a network address). This abstracted approach helps to encapsulate many of the complexities of the communication process within the enabling technology, while actively acknowledging the dynamic nature of the Internet.

Another option is to negate direct communication and have machines send and receive messages to and from abstract endpoints (analogous to a communal mailbox). These endpoints are then distinguished by the types of messages that they are intended to hold. For instance an endpoint may receive and store messages relating to a specific subject such as a service name or data type. These endpoints then correlate to specific message groups held on dedicated messaging servers (Figure 3c). Specialised messaging logic running underneath applications at communicating machines negotiates communication between a user's machine and the messaging server, and in turn between the messaging server and other machines (whether they represent services or other users). In this way, the end-user application need not even be directly aware of how data is transmitted or what it is that the user is communicating with. Such communication is described as 'loosely-coupled' as communicating parties essentially have no direct ties to each other.

2.2 Observing the Quality of Service Interactions

There are any number of different machines active within most large networks (especially the Internet), and they often have different capacities for processing and relaying information, in accordance with any number of communication patterns. This variance in how different parts of a network are represented creates comparable differences in how they uphold communications between a user and a service. There are some services which require certain guarantees as to how the environment (and more importantly how those entities both using and providing the service) will

behave. Disparity (and inconsistency) between different sections of a network can have a detrimental effect on these services. As such there needs to be a means of compensating for these differences if such services are to be useful.

2.2.1 Quality of Service

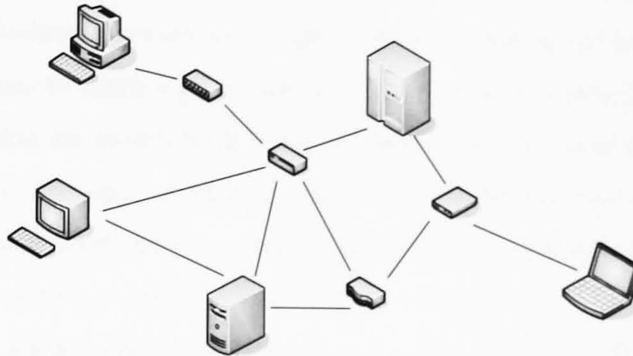


Figure 4 How different machines make up a network

Large networks or networks-of-networks (such as the Internet) are typically made up of different computer elements, each with their own capabilities with respect to the transmission of data (such as transmission speed, message buffer size etc.). Data packets may be sporadically dropped as they travel between pairs of network nodes or may take an unexpectedly long time to traverse the route between their origin and destination. Such behaviour is not uncommon on the Internet, especially as any actions to compensate for this behaviour are typically initiated at the end-system devices [Qosforum99] to afford simplicity and scalability. Such behaviour does not however adversely affect most communications across the Internet.

There are many services which cannot tolerate unpredictable network behaviour. Where e-mail and file-transfer applications have an inherent capacity to tolerate infrequent errors at the communication level, real-time or critical services and heavily data-oriented services do not [Kakadia01]. As such, there are cases where there is a need to compensate for the shortcomings of large networks in order to achieve a more consistent level of performance for a given service. In this context the performance and behaviour of a service is known as the 'quality' of the service. Any steps made towards achieving consistent service quality are then referred to generally as Quality-

of-Service (QoS) management [Colouris01]. QoS management mechanisms do not necessarily provide additional resources to alleviate shortcomings in the communication process, but instead provide a means to better manage those resources that are already available. Simple examples of compensatory actions include the provision of message-ordering capabilities enabled underneath applications at end-system devices, or message-buffering procedures at network routers.

There are different ways to quantify the Quality-of-Service at different functional layers. At the low-level network layer, QoS may be measured by the time it takes for a consumer request to reach a given service (the transmission latency) or the quotient of data packets that are inadvertently dropped due to physical load factors (e.g. buffer overflow in router devices), in essence the basic technical measurements of a service. At the application layer service quality could be represented by how long the application logic within a server takes to process a single client request, or perhaps the window of time within which a user is able to access a service. The latter measures embody abstract concepts that are more in keeping with a user's perception of service behaviour.

Many organisations charge users to utilise their services. This suggests that there must be some level of distinguishing quality to their services which justifies the charges they place on using them, and the use of their services above those of their competitors. For these and more critical services, the rigid and unambiguous definition of service performance attributes is of particular importance. Any bounds on quantifiable QoS characteristics form the QoS requirements of that service [Tanenbaum02]. These then act as quality guarantees, that service providers must make efforts to adhere to in order to ensure that their services perform as expected. Failure to do so could risk penalties associated with underperformance, not least of which would be a fall in confidence amongst their users.

QoS requirements may also apply to service users. If a user misbehaves, by for instance overloading a service with requests, or initiating requests outside of an allotted usage period, they may risk incurring losses or other penalties depending upon their agreement with the provider.

Some of the difficulties in defining performance guarantees come in trying to maintain them. If QoS management procedures have been put in place to manage resources to compensate for changes in the working environment, these procedures must be alert to fluctuations of quality in both the end-system devices and the

underlying network. Delivering QoS over the Internet (with its dynamic and unpredictable nature) is a significant challenge [Mani02].

In their simplest form, efforts to maintain particular levels of quality across a network can be directed in two ways. Firstly, network traffic can be assigned priorities [Qosforum99], to allow the network elements themselves to shape the flow of data or apply resource management techniques (either on a per-application basis or by grouping data from similar applications into an aggregated application-flow). Alternatively, resources can be allocated on a per-application basis, thereby guaranteeing a certain level of quality for specific services. The drawback here is that without adequate resources some services will experience diminished QoS levels in relation to others with a higher priority. Resource reservation such as this can be done statically (essentially off-line) between end-system devices (so that a fixed collection of resources is allocated prior to application-specific traffic entering the system). Reservation of resources can also be conducted dynamically, using a number of techniques such as flow-shaping and flow-control (i.e. the regulation of network traffic in accordance with QoS policies) or flow-policing (the observation of how users adhere to the QoS policies). Dynamic approaches further require monitoring and maintenance of QoS provision levels, so that there is some capacity to react to changes in the performance of a particular service.

Examples of QoS control solutions include both the Reservation Protocol (RSVP) [Rsvp] and the Differentiated Services (DiffServ) protocol [Diffserv], which allow communicating entities to manage resource usage between them at the network level.

2.2.2 Monitoring of Service Quality

To ensure that QoS provisioning mechanisms effectively adapt the resources available to a service to match changes in a service environment, there needs to be some measure of what the QoS characteristics are at any time (i.e. statistical data that describes the performance and behaviour of the service). Quantifiable characteristics that indicate the quality of a service at a particular time may include (among others) server availability, request latency, request processing time, or server usage. The characteristics that are used to assess service quality depend upon the criteria dictated by those parties interested in the performance of the service. Quantifiable service

properties can also be requested in the event of a service provider or consumer wishing to know how well a particular service is performing.

To determine how QoS is experienced between a service provider and a service consumer it may be necessary to install monitoring mechanisms at the end-system devices. There may even be a need to monitor network traffic between them, most certainly within or adjacent to the intermediary network router devices.

A monitoring infrastructure can be used to gather meaningful data to describe how a particular part of the network is performing. This data can in turn be used to provide a view of QoS that properly reflects reality and which can (if necessary) be further processed and presented to a human user in a comprehensible manner.

2.2.3 Monitoring Concerns

When monitoring a service environment care should be taken in deploying the monitoring infrastructure so as to minimise disruption to the workings of the existing system. Interacting parties may otherwise experience undue detrimental effects during use of the system, perhaps then feeling less inclined to have monitoring mechanisms in place. QoS monitoring mechanisms that operate in a non-disruptive manner such as this are said to be capable of 'in-service monitoring' [Chen98].

There are two distinguishable types of non-intrusive QoS monitoring [Dilman01]: 'Statistical monitoring' relies on the examination of network traffic in an attempt to discern predictable trends in observed system behaviour, which are then used to inform how resources are allocated. On the other hand 'reactive monitoring' involves the deployment of a central management platform. When provided with information pertaining to the state of the network the platform is capable of building a global view of the service environment and identifying alarm conditions when they arise (where such alarm conditions usually pertain to a fault or other anomalous event which has occurred within the observed system). This allows for more dynamic QoS management (albeit at the cost of more complex monitoring), and is therefore more suitable for use in unpredictable service environments such as the Internet.

Using an end-to-end view of a service environment a monitoring platform can determine how a service provider and its corresponding consumers are behaving in relation to one another [Jiang00]. As an example, a provider's server platform may be found to be serving requests at an unacceptably slow rate, but this may be attributable

to improper overuse of the service by another client. In this case, although the provider initially appears to be underperforming, it is in fact a specific client that is responsible for the loss of service quality. Without correlating consumer requests with request processing jobs within the server this would not become apparent.

A more detailed system view can be obtained by combining end-to-end monitoring with data from all of the network elements between a sender and receiver, through 'distributed monitoring' techniques. Such an approach affords greater accuracy, especially in determining which network elements are responsible for a particular pattern of events. This does however come at the cost of increased monitoring infrastructure and with this increased interference with the observed network. Most QoS monitoring systems are of the simpler end-to-end variety, but may in some cases assume that there is access to more detailed monitoring data from interconnecting network elements, should they require it.

2.2.4 Monitoring Service Performance in the Internet

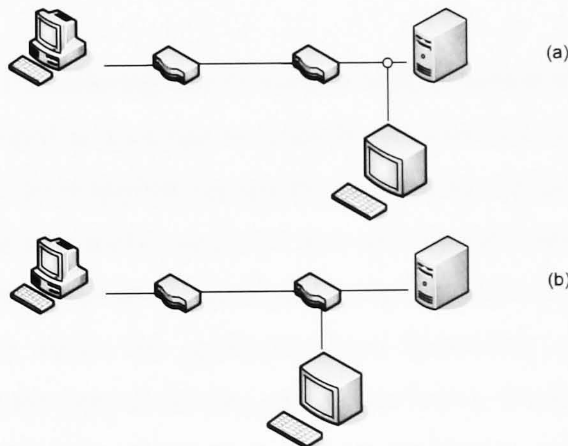


Figure 5 Low-level monitoring techniques: (a) local to an end-system device; and (b) local to a router

When monitoring a dynamic and unpredictable operating environment such as the Internet, monitoring mechanisms must be adapted to differing systems that behave differently over time. There are numerous methods for dynamically collecting monitoring data: at the network-level 'packet sniffing' techniques can be employed, where the data packets that form the most basic network traffic are directly observed

and (albeit simple) performance metrics inferred from them (such as data packet throughput). Observation of low-level network traffic in this way may be achieved by housing specialised software on a machine close to a router or end-system machine (Figure 5a), or directly on a network router (Figure 5b). In this way the behaviour at and around specific points in a network can be observed (e.g. at a specific router, or at the point connecting the underlying network and a service platform). With this basic QoS characteristics such as packet throughput and bandwidth usage can be inferred from the perspective of a specific entity in the network.

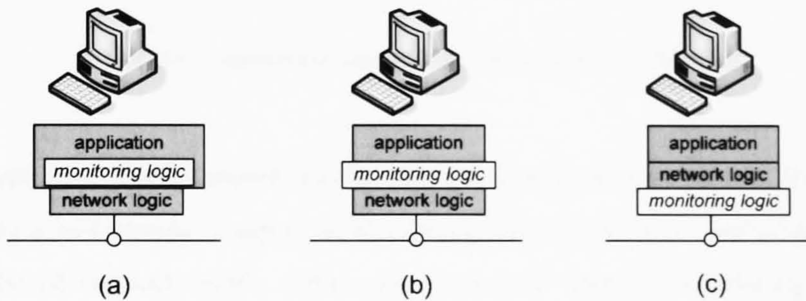


Figure 6 Application-level monitoring techniques

Although low-level monitoring can be used to build a picture of how the underlying network is being used it does not necessarily describe how a particular service is performing. Higher-level application-specific monitoring mechanisms may be used to compose a view of how traffic attributed to a specific application is behaving. Such mechanisms typically need to be installed directly into software (Figure 6), either (a) through integration within the application-layer (providing access to application-specific metrics at the cost of altering application logic); (b) directly underneath the application-layer (allowing observation of inter-application communication without application-specific monitoring logic), or; (c) between the application-layer and the underlying network (a simple alternative, albeit offering a reduced depth of observation data).

2.2.5 Monitoring Service Traffic

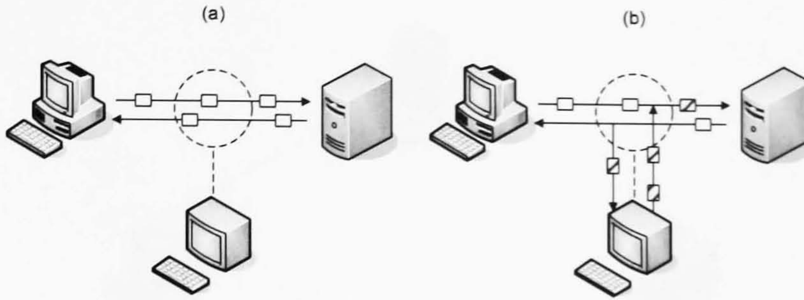


Figure 7 (a) passive and (b) active monitoring techniques

Metrics obtained from network traffic can be based on real user traffic or traffic injected by a monitoring component simulating the actions of a user (Figure 7). The observation of network traffic (otherwise known as 'passive monitoring') involves inferring QoS metrics from the behaviour patterns of existing user-generated traffic within the system (Figure 7a). Care must be taken in how processes treat network traffic so as to avoid adversely affecting the performance of the system. For example if additional information is encoded into user messages to track service interactions, monitoring objects could contribute to a reduction in message transmission speed around the system as they process message contents.

The use of injected traffic for monitoring purposes (generally referred to as 'active monitoring', as illustrated in Figure 7b) involves the periodic creation of 'probe' messages (of a format relevant to the service or application family being monitored). These messages are tracked from the monitoring object in order to determine service provision metrics (such as request round-trip-time). This form of monitoring is non-intrusive but creates additional network traffic. As such care must be taken when scheduling the insertion of probe messages into the system so as not to overload the underlying network.

2.2.6 Creating an Overview of System Performance – Gathering Monitoring Information

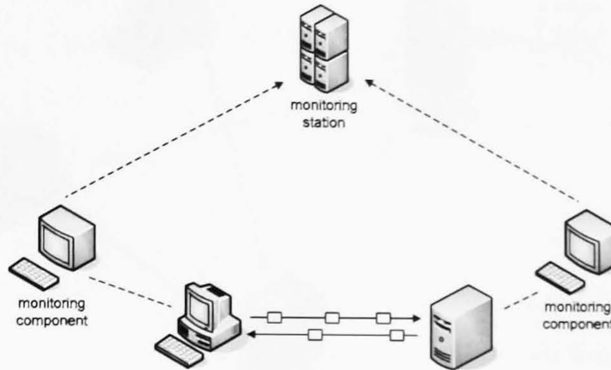


Figure 8 Collection of monitoring data at a centralised location

To compose an overview of how an observed system is performing, metric data from disparate monitoring components must be gathered at a central location (Figure 8). This information can either be sent as part of a coordinated system-wide 'event reporting' procedure [Dilman01] (where data is periodically sent from all monitoring components to the central monitoring station) or it can be collected by the monitoring platform through explicit 'polling' requests (i.e. requests for monitoring data sent to each monitoring component in the network). With either method a balance must be met between the duration of the reporting interval and the amount of monitoring data that is generated. For instance, short reporting intervals may provide finer granularity (and more detailed information) but would place the underlying network under greater strain due to the increase in monitoring traffic [Jiang00].

The amount of data obtained by each monitoring component may need careful consideration. Data collection could be comprehensive (thereby providing a more detailed view), but at the cost of greatly increased transmission overhead. Alternatively data collection could be made scalable by having only a subset of the measurement data transmitted to the central monitoring station. This would however require more processing to select the desired subset of metric data before transmission from each monitoring component [Chan00].

2.2.7 Making Monitoring Information Meaningful – Electronic Contracts

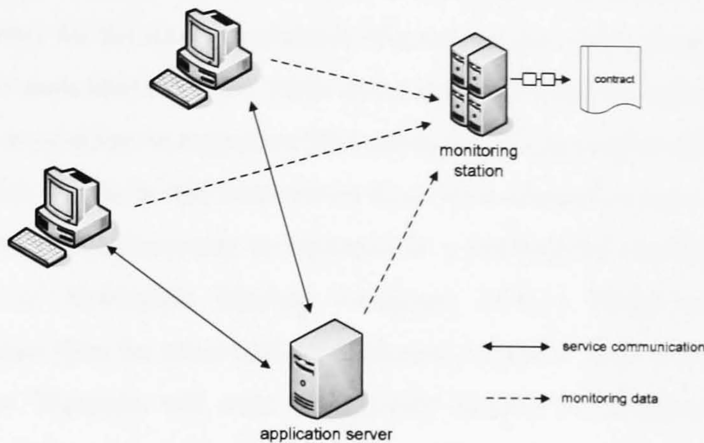


Figure 9 Gathering service-oriented data

Once metric data has been collected at a central location, it must be collated and processed in such a way that it provides a system-wide view of machine performance and service behaviour. In this way different elements and organisations acting within the service environment can be made accountable for their actions.

The quantifiable QoS requirements of a service can have performance bounds placed upon them, which service participants should adhere to. It is a matter of computation to compare measurement data to these bounds to determine if any performance expectations are not being met (i.e. that one or more terms relating to the service quality definition for a service have been violated). Automated evaluation of measurement data allows for uninterrupted monitoring without the need for human intervention, thereby avoiding any disruption to the monitoring process. It also provides greater transparency between an observed system and the monitoring framework.

Parties interacting in a provider/consumer relationship can define electronic contracts which formally record the QoS requirements for a particular service within one self-describing, transferable and machine-readable electronic document. Electronic contracts also provide descriptive information about the participants in a service relationship. These contracts are commonly referred to as Service Level Agreements (SLAs) and are becoming increasingly important in enabling enterprise networks to

operate effectively [Muller99]. SLAs provide unambiguous definitions of expected service behaviour and can accommodate loosely-coupled interactions between organisations [Debusmann03]. Building QoS evaluation procedures around an SLA document allows for the direct evaluation of a service provider's compliance with the QoS that their consumers expect, while conversely allowing the provider to observe the actions of consumers in respect to SLA terms describing permissible behaviour.

A range of SLA standards and taxonomies have been adopted across industry. Within a typical SLA, information may be arranged in a hierarchical structure, for example through use of Extensible Markup Language (XML) [Xml] constructs. This information may then be processed by violation-detection logic associated with the SLA language. Elements will exist to explicitly identify the supplier of the service, and the service client or clients expected to use the service. These elements may also describe participant names and other identifying information such as website addresses.

There may be a list of service obligation elements defined in an SLA, linked to the appropriate participant element, which each define a measurable expectation of behaviour within the scope of the service. For example an obligation may be defined that refers to 'threshold for request processing time within the server'. This obligation would be linked to the service provider description, thereby associating expectations regarding request processing with the provider. An obligation description may include further details such as a numerical threshold for request processing time within the server. Combined with this number there may be an element to indicate the unit of measurement to associate with it (e.g. milliseconds, days). This then provides machine-readable terms that can be used as a measure of adherence to the specific service obligation.

An SLA may include elements detailing the lifetime of the contract (e.g. the contract commencement and expiry dates, or the duration for which the SLA is applicable), with measures and units for the associated dates or times provided as required. The hours within each day during which service provision is applicable may also be stated within the SLA, again within their own distinct, machine-readable elements. These definitions then allow for SLA adherence logic to be applied only when required (i.e. when the governance of a given SLA is in effect).

Despite the aforementioned commonalities, different SLA languages define differing groups of elements to associate with participant bodies, service obligations, and the

contract itself. Even in those cases where descriptions are similar, the arrangement and format of SLA contents may differ. For example in one language a generalised 'party' element may exist to describe an organisation, with a sub-element defining whether it is the 'client' or the 'server'. However in another language there may be distinct elements identifying the 'client' and 'server' parties in the service relationship. The logic used to process SLA contents may then differ between SLA languages, and so each SLA language may have associated with it a distinct SLA engine for interpreting documents. An SLA engine would be used to determine, for example, which of the document elements to examine upon receipt of monitoring data stating that the 'request processing time within the server' was 10 milliseconds.

2.2.8 QoS Evaluation – Accountability

Where SLAs provide binding contractual obligations there are also matters of accountability to negotiate. The demand for accurate and verifiable performance guarantees stands in contrast to the great reluctance of service providers to accept accountability for network elements that exist outside of their perceived domain of control [Overton02]. For instance, different Internet Service Providers (ISPs) provide differing levels of QoS over their own networks [Jimenez04], which an organisation may not believe to be their responsibility when negotiating an SLA with a prospective service customer. Furthermore, if any of the communicating parties governed by an SLA offer to host or maintain the components necessary for QoS monitoring, evaluation and reporting (or otherwise place themselves in an influential position with respect to the monitoring process) they face a conflict of interest. As contrasting cases, they could choose to either report QoS metrics with honesty and precision, or alternatively use their position to manipulate the metrics that are reported. In the latter case this could allow an organisation to avoid any penalties that they would otherwise incur through violation of an associated SLA, or misrepresent the actions of their business partners for their own gain (for instance to justify the nullification of an unprofitable business partnership). In light of such intractable issues, interacting organisations frequently choose to delegate QoS monitoring tasks to trusted third parties that specialise in the provision of monitoring infrastructures.

2.3 Message Dissemination Mechanisms

There are various forms of communication that can be used to support the distributed systems, and which dictate how messages are disseminated from one entity in a network and delivered to another. The choice of communication subsystem also influences how QoS monitoring mechanisms can be integrated into a distributed system.

2.3.1 The Client/Server Model & Socket Layer

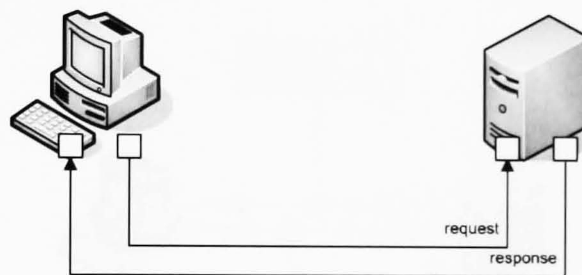


Figure 10 Socket communication

The simplest and most conventional realisation of a provider/consumer relationship is characterised within the client/server model [Ince02]. This model is typically implemented so that a server program (the logic of the application) operates continuously on a computer at a location remote to that of the client entity or entities wanting to communicate with it (e.g. a web server that delivers web pages to Internet users). Clients are realised as programs (e.g. a web browser) on end-user machines, which facilitate communication with the server on behalf of the user. The server program listens for client requests directed towards it over the network (Figure 10), processing requests when it receives them, and responding accordingly. At the lowest level, communication between a client and a server is achieved using the TCP/IP protocol combination [Forouzan03]. A 'socket' is an abstraction that allows a program or programmer to direct data to a specific channel in the TCP/IP communication subsystem of a particular machine, by identifying the IP address of the machine and the numbered 'port' to send the data to. For example, port 110 identifies mail sent with POP3 (Post Office Protocol version 3). A server program can

monitor a number of assigned communication sockets on its own machine. Users then communicate with ports directly or through an associated client-side program. The server process takes input data from these ports and translates it into program requests. When processing of a request is complete the results are transmitted back to the client machine. Since a great deal of the communication is delegated to underlying network protocols, communication between interacting parties is kept simple, but with this is relatively limited in its capabilities.

2.3.2 Remote Procedure Calls

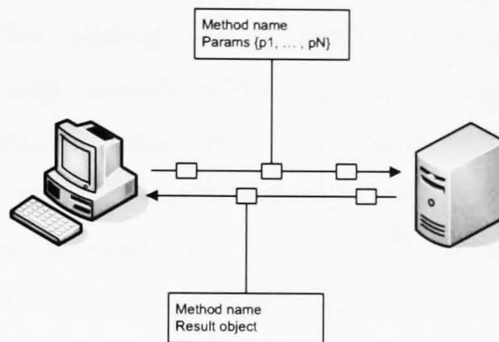


Figure 11 Remote Procedure Calls (RPCs)

In a distributed system functional components are separated between processes that are not necessarily under the control of the same computer structure. These processes have to communicate with each other over a network to complete process calls. Such a system could be implemented with Remote Procedure Calls (RPCs) [Srinivasan95], specialised calls that directly target applications or methods located elsewhere in relation to the origin of the call. All of the information relating to the specific method being invoked by an RPC (as well as information pertaining to the input parameters required by the method) is wrapped within a client's RPC message (as shown in Figure 11). This differs from simple socket communication, which does not encapsulate information in self-describing service requests and responses as RPCs do. A client can send an RPC to a server to invoke a method with a given set of parameters, and receive the results of invoking the method in a response message. The transparency between system components is preserved, as the middleware (i.e. the

underlying logic enabling communication) translates function calls into inter-process messages [Colouris01].

To enable access to RPC-based services from numerous platforms, standardised service descriptors defined in an Interface Description Language (or IDL) are typically employed. Language-specific tools (referred to as stub compilers) can then be used to generate code for use by clients and servers to facilitate communication. These pieces of code are known as 'stubs', where the client and server processes are linked to a client-stub and a server-stub respectively. Stubs hide the details of message passing from higher-level applications. They are responsible for packing (marshalling) parameters into the respective calls and responses, and unpacking (un-marshalling) the parameters into a form that can be understood by the intended recipient program. This packing and unpacking of parameters allows for the standardisation of message contents within inter-process calls, so that rules can be created mapping the data encodings native to the client and the server process to the object representations used in the RPC messages. This has the benefit of enabling communication between processes that do not necessarily run in the same program language. Examples of RPC systems include the Common Object Request Broker Architecture (CORBA) [Corba] and the Sun or Open Network Computing (ONC) RPC system [Srinivasan95].

2.3.3 Web Services

RPCs can be utilised to afford interoperability between different systems, and in this context they are typically referred to as Web services. However a Web service is technically any network-accessible interface to application functionality built using standard Internet technologies [Alonso04]. More and more applications originating within differing platforms and programming environments need to communicate with each other to achieve their respective goals. In this context, Web services act as an interface between application code and any user of that code. They provide an abstraction layer, separating the platform- and programming language- specific details of how application code is invoked. The provision of a standardised Web application layer such as this essentially means that any program written in any language can access application functionality as long as it is able to access the Web service associated with that application.

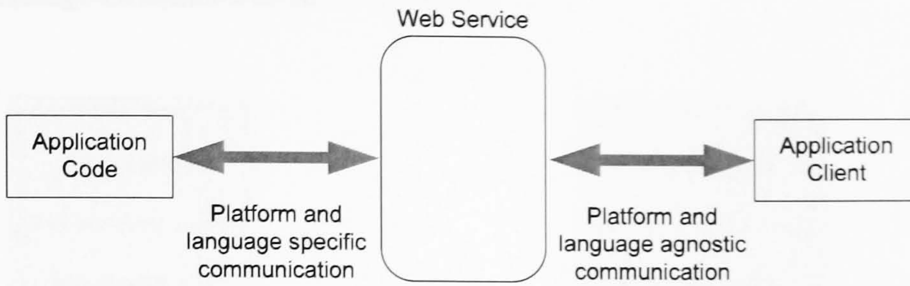


Figure 12 Web services provide an abstraction layer between the application client and the application code [Snell02]

Figure 12 shows how Web services are typically employed. The application code holds all of the logic comprising the system's core functionality. With abstraction behind a Web service interface, there is scope for cross-platform interoperability in a way that makes the platform-specific details of the application logic (on both sides of the interaction) irrelevant. Another way to view Web services is as a messaging framework – using a platform-agnostic approach the only requirement of a Web service is that it must be capable of sending and receiving messages using a combination of standard Internet protocols (which also serves to increase its potential user-base).

Examples of Web Service technologies are SOAP (formerly Simple Object Access Protocol) [Soap] and the Web Services Description Language [Wsdl]. The former acts as a communication format, while the latter describes services and associated protocol bindings, defining how entities can communicate.

2.3.4 Message-Oriented-Middleware (MOM)

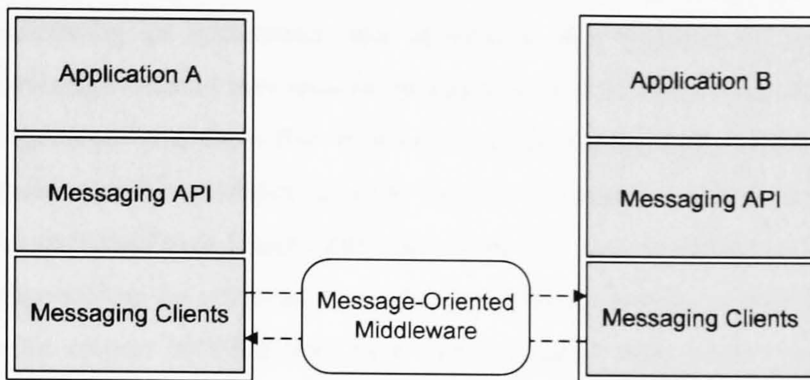


Figure 13 How Message-Oriented Middleware (MOM) may be deployed [Haefal01]

Another form of inter-process communication that allows even greater degrees of abstraction and interoperability is Message-Oriented-Middleware (MOM). MOM is the generic term for messaging systems that enable application-to-application communication through message channels, without the need for direct communication with each other (as shown in Figure 13) [Ince02]. Enterprise messaging systems such as these allow two or more applications to exchange information in the form of messages to inform other applications of particular events or occurrences in other parts of the same system. These messages do not necessarily suggest the instigation of methods or specific processing tasks, with more focus on the content of the messages. This approach is particularly useful in situations where processes are driven by raw information. Open-source examples of MOM systems include the Java Message Service (JMS) API [Jms], and the JBossMQ service [JbossMQ] (soon to be replaced with JBossMessaging [JbossMessaging]). Commercial MOM systems include ArjunaMS [Arjuna], IBM WebsphereMQ [WebsphereMQ] and Microsoft MSMQ [Msmq].

Using MOM, messages are transmitted from one messaging client to another across a network through messaging middleware. These middleware platforms typically afford reliable distribution of messages to their recipients, while also providing configurable levels of transactional support.

Interacting applications exchange messages through virtual message channels or queues. When a message is sent, it is transmitted through dedicated middleware which can be used to indirectly transmit the message to a specific application. As opposed to directly addressing an application, any application that registers an interest in a particular message channel may receive messages from it. In this way the applications that send messages and those that receive them can be decoupled. This pattern of communication can be extended to allow parties to interact without needing to be available at the same time [Eugster00]. Any messages that are found at a particular message channel can be retrieved at a later date without having to wait for them to arrive. In this respect MOM differs from tightly-coupled technologies such as RPC systems, which require an application to know the methods exposed by a remote application and require applications to be available when requested.

A further advantage of asynchronous messaging is that a failure in one logical component does not necessarily impede the operation of other entities in the system. Transmitted messages are treated as complete autonomous units within themselves, with each message existing as a self-contained, self-describing encapsulation of all of the data and state information needed by any application logic that processes it.

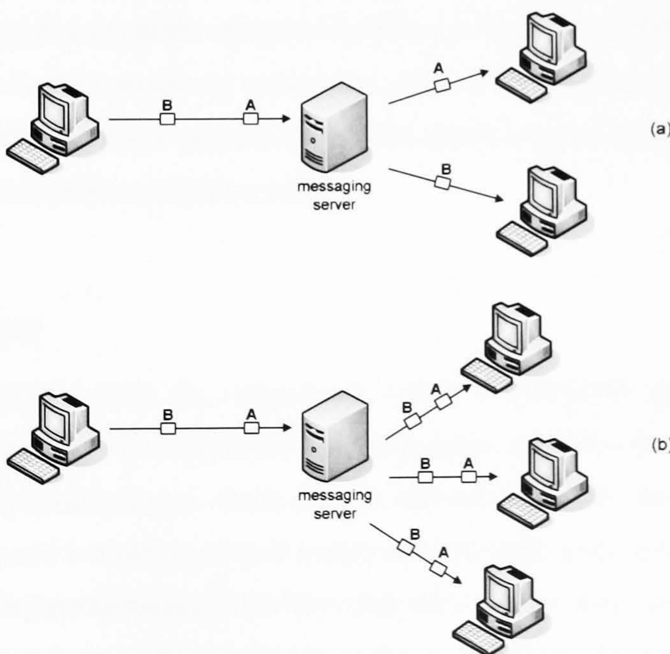


Figure 14 (a) one-to-one and (b) publish/subscribe MOM communication

A number of messaging paradigms have grown from the basic queuing architecture of MOM (illustrated in Figure 14). One-to-one (or 'point-to-point') message delivery allows entities to send and receive messages to each other both synchronously and asynchronously. Only one receiver can consume each message sent to a given channel, although there are instances when a channel can have multiple receivers. In such cases, only one registered recipient will receive each message, as determined by the middleware [Alonso04].

In contrast to the point-to-point approach, the publish/subscribe paradigm (Figure 14b) is intended for situations that demand a one-to-many broadcast of messages. One producer can send a message to a number of consumers with the only requirement for receiving the message being that the consumers must be 'subscribed' to the relevant message channel (i.e. they have registered an interest in the message channel with the messaging server). In this way, producers and consumers are anonymous, and may dynamically publish or receive content to and from different channels. Event-notification systems based on the publish/subscribe messaging model can be used to build loosely-coupled, autonomous components, as required in large-scale heterogeneous distributed systems [Carzaniga00]. There are different schemas that applications can use to register to receive messages [Eugster03]. Applications can declare an interest in a specific channel identifier or 'topic' (as in TIBCO Rendezvous [Tibco]) or in content produced across the channel (e.g. Siena [Carzaniga01] and Elvin [Segall00]). In some systems interest in specific types of messages (such as message object classes) can also be stated.

2.3.5 Summary

There are numerous ways for components within a distributed system to interact. These approaches vary in complexity and abstraction and have their own respective advantages and disadvantages. Such differences must be taken into consideration if the monitoring and evaluation of QoS within an observed system is to be successful.

When monitoring performance at the low-level socket layer, there is a requirement for specialised alterations at the hardware level to make the process of monitoring efficient and non-intrusive. These alterations are usually implemented in network routers (thereby allowing statistical data to be inferred from the passage of data around a network). Although accurate, this does not immediately present a picture of

the end-to-end system performance. Although this is useful for managing QoS compensation actions (by observing changes in network performance) it does not provide a higher-level view of the system and the services active within it, and with this any evidence of how interacting parties are behaving in relation to each other.

Some RPC-enabled languages allow transparent insertion of additional logic at the application-layer, which is then capable of accommodating monitoring functionality. Where this feature is available it allows for monitoring based on the observation of messages and their contents as they pass between an application and the underlying communication sub-system along the request-response chain (e.g. in CORBA [Kim01, Narasimhan99]). Web Service technologies such as SOAP provide features for the transparent insertion of additional information into a designated message header without altering the message payload, which allows for more detailed monitoring. RPC-level monitoring allows inference of application-layer metrics (e.g. observation of the methods being called by a specific client, and the amount of time a request takes to be processed within a server application). Evaluation of contractual obligations (i.e. quantifiable user-perceivable metrics) can then be realised.

With respect to MOM technologies, messages can only be observed directly within the messaging middleware, and it is not guaranteed that functionality to do so is exposed. Without direct access to the messaging medium it is difficult to monitor MOM applications for higher-level QoS attributes without instead falling back on network-layer monitoring techniques. Transparent message interception is however illustrated in the Chameleon framework [Curry04], which was built upon the Java Message Service (JMS), so there is scope for developments in the future. Matters are not helped by the nature of MOM communication, which obscures the interactions between parties. There is no focus on specific services or reciprocal relationships, and so it is difficult to gather data relating to a system in such a way that individual parties can be made accountable for their actions.

2.4 A Different Service Domain – Distributed Virtual Environments

The previous discussions regarding communication patterns and enabling technologies have focused on their use within the context of a provider/consumer relationship. Though form of communication is widely seen across the Internet, another form of interaction that is gaining prevalence through distributed systems is

Distributed Virtual Environments (DVEs). An examination of DVEs is therefore worthwhile within the context of this work, to highlight the associated deployment issues alongside discussion of how the aforementioned enabling technologies can be used to realise a DVE.

2.4.1 Introduction

A Distributed Virtual Environment (DVE) is a virtual world inhabited by geographically dispersed computer users. These users interact with each other and the environment (e.g. terrain) within an instance of the same virtual world, and there may be hundreds or perhaps thousands of users inhabiting a single world (especially in the case of Massively Multiplayer Online Games, or MMOGs). Users of virtual worlds are typically presented with a rich interactive experience, both in terms of graphics and sound elements, in addition to information regarding the actions of other users. Ensuring complex interactions are represented in a consistent and timely fashion for an arbitrarily large number of world participants is a non-trivial problem, which requires that DVEs are supported by design and deployment techniques that are inherently scalable. As message exchange is the only way to propagate events to geographically-dispersed users, care must be taken to prevent exhaustion of the network bandwidth and available processing resources that support a DVE.

One approach to managing resource and network demands in a DVE application is to use Interest Management (e.g. [Greenhalgh95, Zyda91]). This approach frees bandwidth and processing resources through targeted message-passing techniques (eschewing broadcast-based approaches) to ensure that messages are only sent to recipients that may be interested in them. Defined areas within a virtual world are used to restrict message passing, and in this way a message and its receivers are associated with a specific, bounded area of virtual space. Areas used for restricting message passing are commonly termed 'areas-of-influence', and an object is said to exert influence upon (i.e. send messages to) all other objects in its area-of-influence.

When modelling restricted message-passing based on object localities in a virtual world, there is the possibility that 'missed interactions' may occur. Missed interactions occur when objects that should be seen to interact with each other do not due to a lack of message exchange between them. Missed interactions are related to the consistency-throughput trade-off typified by Singhal and Zyda [Singhal99]: "It is

impossible to allow dynamic shared state to change frequently and guarantee that all hosts simultaneously access identical versions of the state".

Considering the aforementioned trade-off, missed interactions occur because the degree of inconsistency present in a DVE is sufficient to allow an object's traversal of an area-of-influence to go undetected by the associated Interest Management scheme.

2.4.2 Interest Management

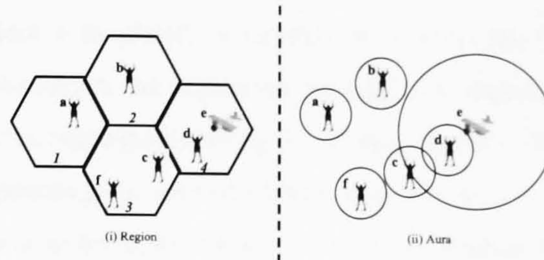


Figure 15 Areas-of-influence

As part of Interest Management objects within a DVE are associated with an area-of-influence within the virtual world, which acts as the criteria for determining whether other objects may be eligible to receive messages from a particular object. This happens aside from the objectives of the DVE, and is purely a mechanism within the processes that control inter-object communication. The use of areas-of-influence allows the communication subsystem to reduce the subset of objects that are to be informed of the position (and perhaps also the actions) of the object or objects within a specific area-of-influence within the virtual world.

Interest Management techniques may be classified into two categories: (i) region-based; or (ii) aura-based (Figure 15). In the region-based approach (e.g. [Zyda91]) the virtual world is divided into well-defined and uniformly-sized static regions (with their boundaries defined during creation of the virtual world). The recipient of a message resides within the same region as the sender, or one adjacent to it. Because regions are fixed within the virtual world, objects that move around will pass between regions, and with this the set of objects with which they communicate will also change during the lifetime of a DVE. In the NPSNET-IV virtual environment model [Macedonia94] for example, the world terrain is divided into hexagonal cells. In the

aura-based approach (as used in the MASSIVE virtual world system [Greenhalgh95]) each object has associated with it an 'aura', which then defines an area of the virtual world over which an object may exert influence. An object then communicates its actions to those objects that are seen to enter its own aura.

The aura-based approach to Interest Management provides a more accurate model of object interaction on which to base message exchange, especially with respect to online games (where it is more naturally representative of the activities of the virtual world participants). Figure 15 illustrates a virtual world scene using both region- and aura-based Interest Management techniques as described. Using auras it can be determined that object *e* (a plane) is capable of influencing objects *c* and *d* (both player avatars). However, in the region-based approach object *e* may only influence object *d* (as object *c* is regarded as being in another region). This shortcoming could be addressed by expanding the area-of-influence of object *e* to encompass additional regions and so allow it to influence object *c*, but if this change were permeated across the DVE it would inadvertently allow object *e* to influence object *f* (so that messages are sent unnecessarily from object *e* to object *f*). An alternative to the exertion of influence over any objects within an aura is to only regard influences between objects whose personal auras overlap. With this, in the aforementioned model, objects *c* and *d* would be capable of influencing the plane (object *e*). This particular approach is commonplace when most objects in the virtual world are similar in nature (e.g. when they are all human-like avatars). Multiplayer game environments tend to favour this type of scenario, as the alternative of allowing a player the capacity to enact influence over another without providing them with the ability to react only serves to detract from game play (as it is not necessarily fair for all involved players) [Sweeney99].

2.4.3 Message Exchange

To facilitate scalable message exchange between users of a virtual world, a communication subsystem must exist that is both capable of identifying message recipients and which can be integrated with an Interest Management scheme. In the region-based approach message exchange may be achieved by associating each region with an identifier, which may then be used to send or receive messages to and from a user. As an example, each region may be identified by an associated IP-multicast address.

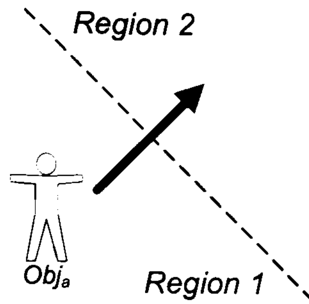


Figure 16 An object crossing region boundaries

Using this example (as illustrated in Figure 16), if an object Obj_a traverses the boundary of a region by travelling from region 1 to region 2, Obj_a will then subscribe as a sender/receiver for region 2's IP-multicast address, and duly unsubscribe from region 1's IP-multicast address as it leaves region 1. In this way, objects need only be aware of which region they are currently in to enable effective message passing to occur – there is no need for individual objects to contact each other directly.

Due to the lack of static regions in aura-based Interest Management, the identification of message recipients is achieved either by the objects themselves (in a peer-to-peer fashion) or via a centralised management server. In the peer-to-peer approach all objects within the DVE must register with the communication middleware used by the application, and must exchange messages periodically notifying other objects of their position within the virtual world, so as to be aware of when aura influence is being exerted over them. These messages are commonly referred to as 'heartbeat' messages, and serve to indicate the location of the sending object. Using positional updates from all other objects within the DVE, a particular object (or a centralised server acting on behalf of all DVE objects) can determine if the area-of-influence of any other object overlaps with its own aura.

Once aura influences have been determined, a phase of high-frequency message exchange is enacted between interacting objects (so as to provide more detailed and responsive behavioural information, improving both consistency and timeliness of updates during object interactions). In the peer-to-peer approach this is achieved independently between each pair of interacting objects. For example, if object Obj_a receives a heartbeat message from Obj_b and determines that Obj_b is influencing Obj_a,

Obj_a will declare an interest in receiving Obj_b's high-frequency messages. In the server-based approach interacting objects send high-frequency messages to a centralised server instead of each other. The server assumes responsibility for discovering interactions between objects by considering all aura influences within the virtual world. Once a server determines the existence of object influences, it relays high-frequency messages between objects.

2.4.4 Missed Interactions

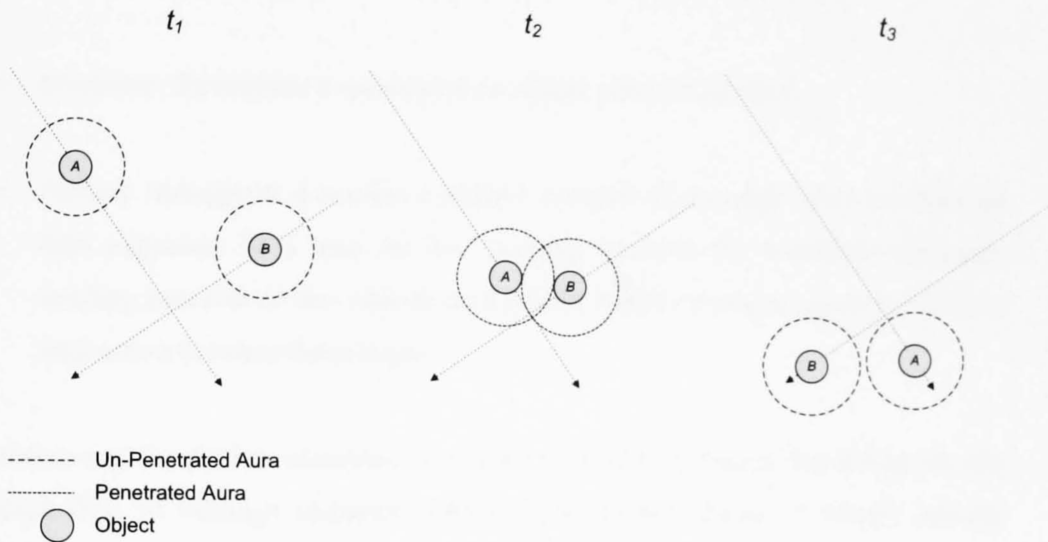


Figure 17 Example of a missed interaction

As an example, Figure 17 illustrates a missed interaction occurring between two objects under an aura-based Interest Management scheme within a peer-to-peer DVE. At time t_1 two objects A and B are travelling on intersecting paths. Objects A and B have heartbeat message-sending intervals of i_A and i_B respectively, which may not be identical. Object A last sent a heartbeat message at t_A, and object B last sent a heartbeat message at t_B, where $t_A < t_1$ & $t_B < t_1$. However, both objects have sent heartbeat messages to each other recently enough so as to be aware that there is no interaction occurring between them. At time t_2 objects A and B have progressed along their paths, and their auras have intersected. At this time $t_2 - t_A < i_A$ and $t_2 - t_B < i_B$. As such neither i_A nor i_B has elapsed, and so neither object has received a positional

update from the other. Because of this neither object is aware of the interaction taking place. At time t_3 both objects have progressed further along their paths, to such a degree that their auras are no longer overlapping. Both i_A and i_B elapse after the point in time at which the interaction between objects A and B ceases. As such, both objects send heartbeat messages to each other, but are unaware of the interaction between them having occurred. This constitutes a missed interaction, contributable to inappropriately large heartbeat message-sending intervals.

To characterise missed interactions the notion of a 'session' is introduced. A session is regarded as an unbroken period of influence exerted by one object over another. Using a session it is possible to describe two types of missed interactions:

- *Complete:* throughout a session no messages were exchanged.
- *Partial:* throughout a session a smaller number of messages were exchanged than expected. This may be for example because the heartbeat message-sending intervals of two objects do not both happen to elapse exactly when an interaction between them begins.

A session may be further classified as unary or binary in nature depending on the expected flow of message exchange between interacting objects. A binary session occurs when both interacting objects are exchanging messages.

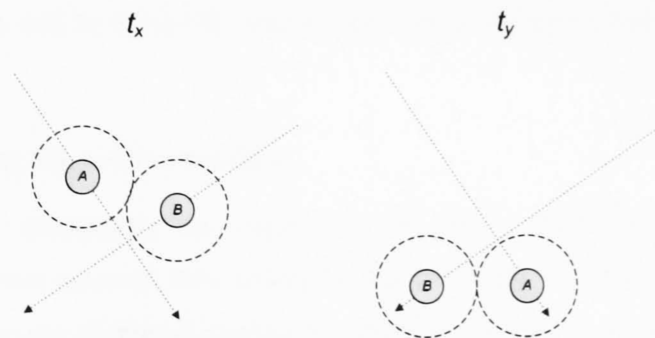


Figure 18 Illustration of a binary session involving two objects with auras

An example of a binary session in an aura-based system is illustrated in Figure 18, wherein objects *A* and *B* interact as their paths intersect. Time t_x represents the time at

which the auras of the two objects first intersect, and t_y is the time at which the auras of both objects cease to overlap. Between t_x and t_y both objects will send high-frequency messages to each other. A unary session occurs when only one object should be sending messages. In the region-based approach all sessions are binary (as objects that share a region influence each other). In the aura-based approach unary sessions are possible if an object must be within another object's aura in order to be influenced (assuming different objects can have different aura sizes as in Figure 15). Missed interactions may manifest themselves as a unary session when a binary session should have occurred e.g. when two objects should be exchanging messages during a session but only one of the objects is aware of it. A situation such as this may occur if objects are sending heartbeat messages at different intervals, and interacting objects are moving rapidly enough for only one object to receive a heartbeat message during the session and instigate high-frequency message-sending.

The implementation of a DVE also influences the occurrence of missed interactions. If a centralised server is used, it must inform objects of sessions taking place before missed interactions occur. Since messages are sent between all DVE objects and the server, the occurrence of missed interactions relates directly to the speed (i.e. scalability) of the server in determining the occurrence of interactions. If the server does not have the resources to process each interaction in a timely manner, missed interactions will be seen to occur. In the peer-to-peer approach the occurrence of missed interactions is regulated by the process of heartbeat message exchange. The less frequently heartbeat messages are exchanged between objects the more likely it is that interactions will be missed by one or both objects during a session.

2.4.5 Avoiding Missed Interactions

Approaches to minimising the occurrence of missed interactions within virtual environments have received little interest in the literature. No attempt has been made to describe the types of missed interactions that may occur in a virtual world and how these may relate to different Interest Management schemes given the use of client/server or peer-to-peer implementation models.

Assuming that networking and processing resources may not be easily altered to alleviate the problem of missed interactions, current efforts to minimise their occurrence are managed in an ad-hoc manner. These include reducing the maximum

achievable velocities of objects (making it easier to observe their actions) or increasing the size of the area-of-influence for each object (so there is a greater chance that an influence may be resolved before auras are completely traversed by other objects). When considering peer-to-peer implementations, a developer may also choose to increase the regularity of heartbeat message exchanges to provide more detailed positional data (albeit at the cost of greater message production across the system).

Alterations to the aforementioned parameters for a given DVE will manifest themselves as a change in the Quality of Service (QoS) experienced by users. Decreasing the achievable velocities for world objects may reduce the responsiveness of a virtual world, whereas increasing areas-of-influence or reducing the interval between heartbeat message exchanges may result in unnecessary message passing (thereby potentially slowing the underlying network and the detection of interactions). The core issue to be addressed in this context is the derivation of optimal values for those parameters that govern influence and message exchange frequencies between virtual world objects, in such a way that the associated DVE remains scalable and responsive.

2.5 QoS Provision – Case Studies

Distributed systems come in many forms, with the potential for differing relationship dynamics between interacting parties and the utilisation of various means of communication. Interactions can be monitored for quality (with respect to both performance and behaviour) using a variety of methods, depending upon the enabling technologies that underpin the system. Distributed systems can differ with respect to how they are structured, and their QoS monitoring requirements.

Case studies are presented examining two contrasting systems: E-Commerce services and Distributed Virtual Environments (DVEs). In examining these two different forms of distributed systems, comparisons can be made and conclusions drawn as to which aspects make each individual system unique, while also providing a view as to the similarities that may exist between them. This examination will clarify how a generic monitoring framework will have to adapt to be useful in a variety of contexts.

2.5.1 E-Commerce Systems

E-Commerce systems include web-based applications that have to provide some level of structure and interoperability e.g. Web services and similar data-processing services. Such services may not necessarily be used at a constant or regular rate, depending on the changing demands of service consumers. Maintaining valid and correct working behaviour within an E-Commerce service is of great importance. Such services are heavily data-oriented, and as such the prevention of loss or corruption of message data is often a high priority. It is important for these services to remain available to serve the needs of consumers as and when required by them.

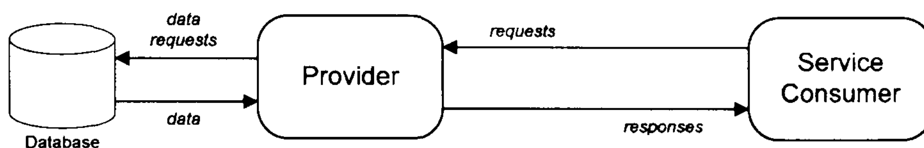


Figure 19 Communication pattern in an E-Commerce environment

When processing service requests it is not uncommon for a service provider to access some form of back-end legacy system (e.g. a database server) to retrieve information necessary for the successful completion of client requests (Figure 19). This in itself can influence the performance of a particular service, depending on the capabilities of the back-end system and the connection with the service platform.

Communication in an E-Commerce system is typically bidirectional between service providers and service consumers, and both parties may be capable of participating in complex multi-part request-chains. This adds further to the demand for reliable communications between interacting parties.

There are several QoS performance characteristics that are of particular importance within an E-Commerce environment, and which can influence the success of a given service:

- *Availability:* whether the service is present for immediate use at a particular time. If consumers cannot use a service that they have paid for, or a service is unavailable when prospective customers happen upon it, the service provider only stands to lose revenue and existing or potential members of its user-base.

- *Throughput:* the rate at which requests are successfully processed by a service, which can be influenced both by the processing resources available to the service and the request load produced by service consumers.

- *Packet Delay (Latency):* any inconsistencies in the timeliness with which requests are processed may inadvertently affect the quality of the service, producing outdated or invalid results. One example which would require reduced latency would be if a provider offers a service that purports to accurately inform a user of the time in any time-zone across the globe.

- *Reliability:* refers specifically to the regularity with which consumer requests are met with correct (i.e. meaningful) responses.

Other factors that can contribute to the perceived quality of an E-Commerce system include the notion of accessibility (the degree to which a service can serve a request, focusing on how it allows cross-platform communication between different operating environments), and any provisions for secure communications if they are requested (such as confidentiality and non-repudiation). There may also be stringent requirements with regards to how a service conforms to the law, as well as compliance with any business practices associated with the service environment. When such quantifiable needs materialise the QoS requirements and obligations associated with a service can be encapsulated in a Service Level Agreement (SLA).
Modern E-Commerce systems employ component architectures such as Java Remote Method Invocation (RMI) [Javarmi] or Web Service technologies such as SOAP. Combinations of technologies such as these are particularly suited to E-Commerce as they can be configured to work with an organisation's existing communications infrastructures (as with Web services), while offering extensions that address transactional and security demands.
In whichever way QoS is managed in an E-Commerce service environment, it is desirable that applications active within the system are shielded from the complexity of the associated QoS provisions [Aurrecoechea96]. Employing a principle of transparency acts to reduce the additional functionality incorporated at each

participating machine. Transparency also hides the details of QoS specification from the application, delegating QoS management activities to the supporting framework. There are a number of ways in which QoS management functionality can be realised [He01], as shown in Figure 20:

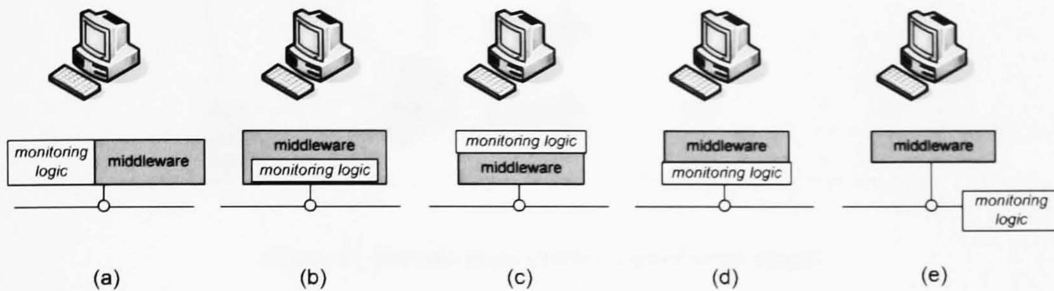


Figure 20 Various approaches to QoS integration in middleware

a) The *Service Approach* implements QoS enhancements as separate middleware services, providing portability and interoperability.

b) The *Integration Approach* directly modifies the middleware platform to provide enhancements, achieving better performance at the cost of interoperability.

c) The *Interception Approach* relies on intercepting messages from either above the middleware (providing a higher level of abstraction) or;

d) Below the middleware (allowing for the utilisation of efficient transport protocols).

e) The *Gateway Approach* involves the insertion of a QoS-enabled component at the transport layer between the end-system devices, which is then responsible for implementing QoS enhancements [Schantz99].

A combination of these approaches can also be employed, providing a broader range of control and data gathering from different parts of an observed system.

2.5.2 Distributed Virtual Environments

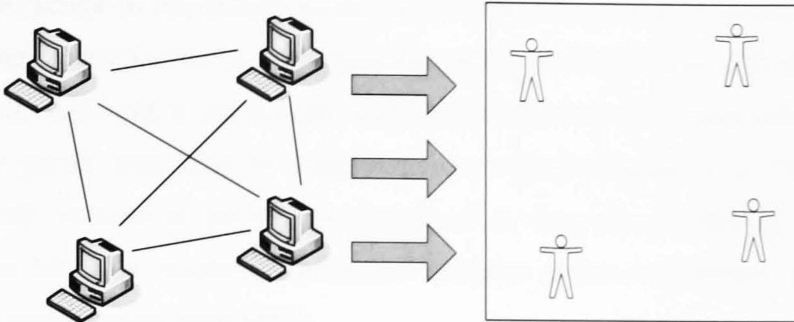


Figure 21 Machine users within a virtual environment

Distributed Virtual Environments (DVEs) may come to exist within Massively Multiplayer Online Games (MMOGs) or strategic military applications (e.g. [Karr97, Mastaglio95]). Within these there are potentially many hundreds or even thousands of objects interacting within the same virtual world, with many directly representing human users (as in Figure 21). Users of an application typically control characters and are able to move their own character within the confines of a virtual environment. Information relating to their position in the world and their current state (e.g. the nature of the actions that they are performing at a given time) must be communicated in real-time to other participant nodes to maintain a consistent world view for all active users of the virtual world. In this sense the primary concern of each DVE is to maintain a reasonable level of quality with regards to the experiences of its users.

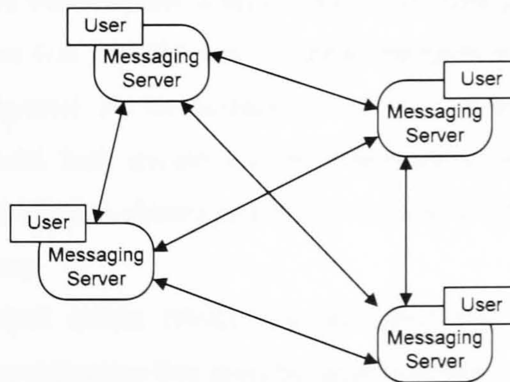


Figure 22 Peer-to-peer (P2P) communication pattern in a DVE

Nodes participating in a DVE application may transmit information directly to all other nodes active in the virtual world (as in Figure 22), or to a centralised server which disseminates the data to the applicable network nodes (Figure 23) [Merabti04]. The latter is employed in small-scale multiplayer games such as Half-Life [Halflife]. For larger games that need to support greater client numbers (e.g. Second Life [Secondlife]), clusters of servers are employed to share the processing load. Most commercial MMOGs employ a back-end database server to maintain player state within the virtual world [Hsiao05].

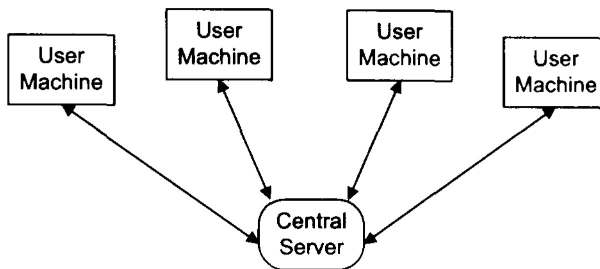


Figure 23 Centralised communication pattern in a DVE

Each node has an obligation to act in a timely and accurate manner, transmitting positional data for the associated entity (typically a human user) in a reliable fashion (either directly to other nodes or to a centralised server). The actions of the entity can then be accurately represented in the virtual world. The low-level QoS requirements for a DVE application would concern real-time responsiveness (directly relating to processing latency and transmission delays in the underlying network) and consistent system-wide behaviour (i.e. reliable and complete transmission of world data to the relevant set of participants). As an example, if one user walks directly into another user in the virtual world, both should see this happening at approximately the same time (i.e. with enough time to absorb and react to the new information and preserve fairness across all users).

The main thrust of QoS within DVEs is a consistent user experience. Users are directly involved in the processes that must be regulated, and it is the consequences of their actions that must be managed. This differs from E-Commerce systems where processes are transparently automated, predictable, and influenced only by the

existence of service data within the system. Also, within an E-Commerce application the details of each process must be precisely defined, and can remain hidden from human users. For example, a person browsing a bookseller website doesn't know that a warehouse is being queried for stock availability, and even less so the nature of the obligations that the warehouse has with regards to provision of their own service(s). In contrast, DVE users want to see precisely what other entities participating in the virtual world are doing, and it is this which must be supported through the application of QoS mechanisms.

To manage the number of users that could conceivably inhabit a virtual environment, the underlying communication medium must be scalable, and able to support users entering and leaving the DVE dynamically (much like they would within Web service interaction). Existing network-level infrastructures are typically used to provide communication sub-systems within DVE applications to promote heterogeneity between differing client platforms. This would mean that network-level monitoring techniques would be the only means of observing QoS performance in a DVE.

Message Oriented Middleware (MOM) has also been identified as suitable for providing the basis for message exchange in distributed multimedia applications [Gore01], with the Mercury message dissemination system [Bharambe02] providing a positive assessment of MOM support for networked games (i.e. MMOGs). DVEs may then employ messaging middleware, such as CORBA (as in [Wilson01]).

Beyond basic QoS parameters such as network delay and data loss there has been little research into what constitutes the QoS requirements of a DVE, and therefore how the underlying communication mechanism affects the realisation of QoS monitoring in DVEs.

2.6 Related Work

2.6.1 E-Commerce & Web Services

In light of the discussion in previous sections relating to E-Commerce systems, there are a number of requirements that must be satisfied if an SLA monitoring framework is to be truly useful in practice:

- The framework should be generic, so as to facilitate the wide range of service configurations that exist in the heterogeneous service environments of the Internet:

    - With respect to the application technologies that it can interact with, both in the collection of data from within existing E-Commerce systems and when integrated with underlying communication technologies.

    - Towards the range of service domains that it can operate within. This would increase the usability of the framework within evolving service environments.

    - The framework should be able to accommodate existing QoS definitions (including SLA languages), along with any contract terminology that organisations may wish to use. This would reduce the need to re-interpret existing business logic.

- The framework should be scalable to match dynamic E-Commerce service environments:

    - It must scale to observe an arbitrary number of clients without adversely affecting the performance of the service environment.

    - It must be capable of monitoring and evaluating a growing number of service contracts without detrimentally affecting the existing system.

- The monitoring framework must be characteristically transparent:

    - In its deployment, so a minimum of effort is required to enable monitoring within service environments.

- In its operation, keeping interference with the workings of the existing system to a minimum, while also requiring as little maintenance on the part of the service participants as is possible.

- The framework should reduce the amount of work required to tailor monitoring and evaluation mechanisms to a particular service environment, and allow reuse of functional components and evaluation logic.

These requirements shall be considered in the appraisal of the following related works.

2.6.2 Commercial Research

2.6.2.1 Pruyne

Distributed applications rely on specialised middleware to enable communication. Pruyne [Pruyne00] highlights the fact that middleware applications in and of themselves do not naturally provide QoS support. Pruyne proposed extending higher-level QoS provision in middleware components through the notion of 'interceptors'. These interceptors are intended to act as additional logic components that allow inspection and manipulation of application-level data as it moves between application components, most notably for the provision of QoS service extensions. This infrastructure of re-usable QoS components was designed for use at both the client- and server-side of the end-to-end system. There is the capacity for interceptor components to be dynamically inserted (potentially in chains) between existing middleware and the application level above. This allows for dynamic adaptation to changing service environments, through the insertion of logic that is able to modify service behaviour to improve performance. The focus is on the need to separate QoS provision from application-level QoS properties, thereby allowing for the reuse of functional interceptor components across separate systems. This inherently separates functionality from any specific application or enabling middleware technology, with QoS extensions acting as a framework to be deployed around existing services.
A standardised request interface is proposed, encapsulating methods for accessing interface and method names, as well as obtaining, altering and adding parameter and

method return values. These methods can then be accessed by interceptor implementations to realise QoS behaviours, as request objects adhering to the latter interface are passed to each interceptor in an interceptor chain.

This work allows for the development of exception catching processes (to manage behaviour in the face of application errors), as well as a capacity for communication between separate interceptor components. Examples of interceptor components to provide high service availability and admission control are discussed in Pruyne's work. There is however little consideration of factors relating to deployment and subsequent use in a platform-agnostic system environment where scalability requirements and ease of placement within an existing system are key factors. Also, there is no discussion of how interceptors can be integrated into larger frameworks for the purpose of enabling end-to-end QoS extensions such as service monitoring.

2.6.2.2 QuO

The QuO project [Schantz99] proposes a QoS-aware object gateway that can be transparently positioned at the transport layer to provide end-to-end QoS manageability within the underlying middleware (and not within the application-layer). It is intended that as well as providing predictable system behaviour, such components support the addition of mechanisms to the system environment to enhance the behaviour of the service participants (without necessarily altering the existing application constructs already in place). The QuO gateway allows placement of dynamically reconfigurable and adaptable QoS enforcement components, although the potential for anything beyond the provided range of available QoS behaviours is ultimately under the control of the project authors.

The QuO framework augments CORBA Interface Definition Language (IDL) interfaces with QuO Quality Description Languages (QDL), providing an encapsulation of QoS service properties. Two QDLs, Contract Description Language (CDL) and Structure Description Language (SDL), describe contract information and selection information respectively. These augmentations allow QuO gateway objects to operate transparently at both the client- and server-side. Insertion of QuO interfaces into a system is achieved through use of the Internet Inter-ORB Protocol (IIOP). Client-side and server-side gateway objects then implement QoS-aware transport protocols and QoS management functions. Examples of existing implementations

include provision of assured bandwidth using RSVP [Zhang93], and service dependability using group communication services as part of the AQuA Project [Cukier98].

With respect to the deployment of gateway objects, although compatibility is offered with any off-the-shelf Object Request Broker (ORB) products (through the IIOP), limitations are imposed in the way in which client-side stubs must be explicitly altered to allow use of the gateway object.

In experimental use the QuO gateway introduced additional latency to the effective operation times of the existing middleware stacks, suggesting scalability issues that hinder effective deployment. Also although a simple QoS contract mechanism is included, it is primarily used to manage the actions of gateway components in response to specific system events, and as such does not offer any scope for explicitly defining how the service participants themselves should behave.

2.6.2.3 WSLA Monitoring & Evaluation Framework

An example of a complete QoS infrastructure is the WSLA Framework [Debusmann03, KellerIbm02, KellerLisa02] developed by IBM to address the need for management of system-wide SLA specification and monitoring processes within a Web services environment. The framework acts as a complete application-level QoS infrastructure, enabling service quality enhancements for stand-alone Web applications that have varying QoS and business requirements. As such the WSLA Framework essentially acts as an off-the-shelf solution for realising SLA-managed service relationships. There is also accommodation for an SLA language that accompanies the work, described in [Ludwig02].

The WSLA Framework is capable of transparently measuring and monitoring QoS parameters, and evaluating monitoring data against the terms of an SLA. Interacting parties can also be notified of SLA violations that occur during the lifetime of the associated service. An in-built SLA 'Compliance Monitor' evaluates and regulates system configurations and run-time metrics relevant to an observed SLA.

Components of the monitoring framework are able to operate in a modular fashion, even across network domains, providing separation of concerns and delegation of monitoring tasks to different network entities. This relates to scenarios where communicating parties wish to delegate monitoring tasks to trusted third parties,

either because an unwillingness to accept the additional workload or due simply to a lack of trust between parties. It is feasible to collect monitoring data from numerous sites and cross-check the data for verification purposes, while at the same time disseminating SLA contents on a 'need-to-know' basis. This shows consideration of data confidentiality and the need to reduce unnecessary replication of data across cooperating and remote monitoring sites.

A 'Measurement Service' component within the Compliance Monitor maintains information about the state of an observed service, either by probing the service or intercepting client invocations within the observed service platform. This component can be replicated at different sites to allow multiple third-parties to share monitoring duties.

There is discussion relating to the classification of QoS parameters within an SLA, such as network-level measurements and complex metrics (composed from basic measurements), and how metrics map to contractual obligations (i.e. SLA parameters) and financial concerns. This shows an approach to generalising the definition of complex service requirements.

Additional discussion concerns deployment of an SLA monitoring framework in an industrial context, relating the financial impact of a monitoring framework and the use of SLAs to govern business practices, as well as resource management and issues of accountability with respect to monitoring entities. This work also validates the need to provide unambiguous SLA terminology and consolidate the different definitions of QoS parameters that may exist across different contract languages (with examples including 'response time' and 'availability').

There is discussion of the need for all interacting parties to be directly involved in the SLA deployment process, highlighting a necessity for extensibility and customisation of contracts. The SLA engine is capable of interacting with various service technologies through specialised middleware plug-ins that connect the measurement components with specific technologies. The SLA engine also provides bindings to different communication protocols (including the SOAP and HTTP protocols) for obtaining measurements.

The WSLA Framework considers the reuse of contract content across similar service relationships. Contracts are defined in XML, and logic for calculating complex metrics is defined in auxiliary files and subsequently referenced in each contract where required. Pre-defined evaluation logic can be re-used and more measurement

capabilities afforded in the future, allowing interacting parties the capacity to tailor data monitoring processes to their own needs.

The SLA language that accompanies the WSLA Framework details a number of elements. A 'Service object' acts as an abstraction of the technical obligations of a service, representing services or individual service operations. Measurements associated with an individual service or operation are represented as 'SLA Parameters' (e.g. 'downtime', 'throughput', 'response time'). An 'SLA Parameter' definition acts to describe a unit of measurement, its name within the contract, its metric class, and the service to which it refers. 'Metric' definitions then describe how a measurement is calculated based upon its metric class. Performance obligations that must be upheld during service operation are defined in 'Service Level Objectives'. The latter include reference to a metric, a threshold value for the metric and a period of validity for the obligation. The violation of a threshold value constitutes a violation of the contract e.g. if server throughput falls below an agreed lower limit. If a specific procedure should be followed upon an obligation being violated, an 'Action Guarantee' can be automatically instigated to perform a pre-defined action within the WSLA Framework. This may include notifying interested parties of service violations using the communication subsystem. These SLA evaluation processes are carried out by the 'Condition Evaluation Service' within the Compliance Monitor.

The WSLA Framework attempts to automate the process of creating and deploying an SLA. It automates use of Web Service Definition Language (WSDL) [Wsdl] descriptors to construct an SLA document for a particular Web service. The SLA can then be coalesced with business requirements (with the consent of the service participants) using the 'SLA Establishment Service' component. This reflects the need to simplify SLA monitoring and deployment while considering per-party requirements. SLA content is then disseminated by the 'Deployment Service' to monitoring entities.

The WSLA Framework does not fully consider issues of deployment, such as the capacity to scale with the number of observed entities in a given system and the range of service domains that can be accommodated. Furthermore, the available contract language set is proprietary. Users essentially have to utilise the language offered by the framework or they cannot benefit from any of its functionality.

The way in which monitoring is carried out is not necessarily transparent or scalable. For instance, there is no consideration of the amount of additional network traffic that

monitoring generates with a growing base of service participants and replicated monitoring components. Little attention is also afforded with regards to the communication protocols that link service participants. Measurement components would need to be capable of adapting to these.

2.6.2.4 Business Management Platform (BMP) Agent Network

Sahai et al [Sahai02] describe an automated SLA monitoring infrastructure for use in Web services environments. The monitoring components of the infrastructure provide application-level accountability amongst service participants and a richer view of high-level service aspects. Proxy components are deployed within the SOAP communications layer to transparently intercept messages as they pass between communicating parties, correlating these with application logs so as to put them in context with the associated business processes. These actions are managed by a Business Management Platform (BMP) local to each participating machine.

Measurements within a platform are instrumented by attaching a proxy object to the SOAP 'router' responsible for re-directing messages between a client and a server. This allows observation of client and service behaviour. It is assumed that there is a tool available to allow observation of process logs relating to the target service process within the server platform. Measurements relating to a particular service are associated with a WSDL or WSFL (Web Services Flow Language [Wsfl]) service specification. Measurements are collected and stored in a dedicated repository, and the 'SLA Violation Engine' records information pertaining to service obligation violations.

One driving factor in this work is the accurate representation of the client experience. Within each service relationship measurements are taken from both the service provider and the service consumer so as to determine whether conditions within the server are reflected at the client-side (with respect to service metrics e.g. response time). This then provides a valid representation of higher-level application performance. The work also considers the chaining of Web services, wherein the satisfaction of a particular request may require communication with additional services. Accommodating such scenarios extends the adaptability of the work to differing service environments.

With regards to the specification of monitoring criteria the work provides unambiguous SLA definitions and customisable monitoring logic. This includes instrumentation in an existing system and the associated evaluation of data. Higher-level abstract concerns are also visited, such as the measurement and evaluation of complex metrics. Examples include the response time of a Web service operation or the resource usage of a particular operation.

The accompanying SLA language is described in [Sahai01]. This language is defined in XML, with elements describing the obligations of each contract arranged hierarchically. An SLA is defined by its period of use and a set of 'Service Level Objectives' (SLOs). Each SLO refers to a set of measurements and a series of conditional operators. These operators dictate when the objective should be evaluated, the function to use to evaluate the measurement, and the action to take upon violation of the function. Each measurement description indicates where the metric is to be sampled (e.g. within the service platform) and the format of the measurement data.

The SLA language can be extended to include additional object definitions and data evaluation logic to potentially meet previously unconsidered requirements. Also although re-use of contract components is possible, it is suggested that only the administrator of a monitoring framework be entrusted to develop new measurement component logic when required (which can then, to the work's credit, be stored for re-use within an SLA language library local to each BMP Agent).

There are a number of prevalent scalability issues. In order to provide an end-to-end view of system performance, monitoring logic is inserted at both the server and client. This means parties dynamically partaking in service relationships have to consider maintenance of additional monitoring logic. Also, either the clients themselves or the providers of the monitoring logic would be ultimately responsible for maintaining the additional monitoring components. This essentially peer-to-peer approach to SLA monitoring is not necessarily suitable for deployment across any and all E-Commerce services (for instance those scenarios wherein a client's SLA obligations are far outweighed by those of the service provider).

There is little mention of scenarios where multiple SLAs exist between a service provider and numerous consumers, or how the maintenance of such a system might be made scalable. The fact that Business Management Platforms are essentially responsible for monitoring and regulating themselves and potentially those entities that interact with them (thereby inviting inefficient overlap of monitoring concerns

between BMP components) also affects scalability. This is however offset by functionality that aggregates monitoring data before it is transmitted between BMPs, thereby reducing the network traffic that is generated.

This work assumes that the SOAP Web services communication protocol is the only protocol in use between interacting parties. This validates the use of middleware proxies to achieve transparency and observation of application-level behaviour (through insertion of proxy objects into the SOAP stack between the sending application and the network level). However, part of the monitoring process involves intercepting messages and either altering them to include an additional service identifier or interrogating them to determine the presence of such an identifier. These procedures may not scale with the number of clients or services active within an observed system.

Server-side processes are augmented with a proxy component that reads the internal server logs and correlates their contents with message identifiers. Although this shows consideration of application-level monitoring through composition of a high-level view of system behaviour, it assumes the existence of a specific process-log consolidation and interrogation application. There is no mention of how BMP Agents may develop a high-level view of application behaviour in a generic manner i.e. in service platforms that do not expose service log contents.

It is assumed that all providers and service consumers have a BMP Agent already installed locally before a service comes into use, and as such there is no mention of how new clients can choose to participate in a service environment in a naturally dynamic manner while also enabling service monitoring.

Although BMP Agents can carry out different actions based on SLA evaluation results, there is no discussion of how to notify interested parties of contract violations. Such information is maintained in a centralised data store within each BMP Agent, which must be manually accessed at the site. Associated with this is the fact that the interface for viewing contract information and evaluation data is only available locally to the machine that is carrying out SLA evaluation. It is the only machine loaded with the SLAs and all of the associated measurement data. Such a machine would act as a potential bottleneck in the system should numerous organisations wish to view the data (a concern which can also be applied to the provided violation log).

2.6.2.5 QoS Monitoring Framework for Traffic Engineering in IP Differentiated Services

Asgari et al [Asgari03] developed an intra-domain monitoring system to augment their own previous work in the field of traffic-engineered Differentiated Services. This allows for reactive adaptation of service quality for different active application flows within a service environment. One of the core goals of the work was to provide a scalable means of monitoring service flows i.e. one that does not inhibit the operation of observed services. As such, before constructing the monitoring framework, scalability principles were formulated relating to the data-collection framework, such as minimisation of measurement traffic through event notification mechanisms, and regulation of the traffic generated by the framework. Qualities relating to the scalability of the monitoring framework in general were also considered. These included the ability to scale with an arbitrary number of service subscribers, and the number of contractual obligations that a monitoring framework could be capable of evaluating at a given time.

The monitoring framework itself consists of a number of components. Each router between the interacting service endpoints has a monitor object associated with it. This monitor is capable of carrying out passive measurements on the associated router, as well as active measurements of the traffic around both the router and other routers in the chain. The monitor component is also capable of performing limited data evaluation and threshold violation detection (augmented with additional violation notification procedures). With router monitors distributed across the end-to-end chain, a centralised network monitor object performs network-wide post-processing of measurement data collected from all of the router monitors, supported by a library of statistical functions used to process incoming data. This post-processing component uses the distributed monitoring data to build a logical view of the structure of the network between service participants. In this sense, the monitoring framework attempts to gain awareness of changes in network state, by retaining a dynamic view of the service environment. Because this component does not strictly perform real-time processing of data, it also avoids any responsibility regarding issues of scalability and processing bottlenecks within the observed network.

An additional component collects data from the router monitor objects and the centralised monitor to perform service-level monitoring and auditing, taking further responsibility for creating and deploying the basic router monitoring objects. Cross-

examination of system performance against Service-Level Specifications is also performed in order to detect QoS threshold violations etc. This process is augmented by a repository for monitoring and configuration data and an interface for presenting statistical data to end-users. Configuration data includes information about the nature of the metrics being measured, and measurement operations are scripted in XML schemas by clients and used to structure statistical evaluation processes. All of these aspects contribute to how the system can be tailored to suit differing service requirements.

The components of the monitoring framework are assembled in a modular fashion. Monitor objects use CORBA interfaces and event notification channels to communicate with each other. This approach enhances the scalability of the framework in respect to the number of network routers being monitored, and validates the use of Message-Oriented Middleware in providing scalability within a distributed monitoring environment.

The framework incorporates other novel approaches to achieving scalability, primarily with respect to both the size and speed of the network and the number of active participants within a service environment. These include minimising cross-component notification transmission overheads and reduction of the synthetic traffic injected into the system for active measurement purposes. The authors also consider the scalability of a monitoring framework in relation to the provision of different classes of distributed service (e.g. real-time services, best-effort services etc.), while addressing the monitoring requirements of these different service types. However, the framework only accommodates these scenarios by collecting all available monitoring information and assuming that in doing so it provides the necessary data for every service. As part of the data collection process, reports are created based on QoS measurement data for direct comparison with Service Level Specifications (SLSs).

The authors try to account for the differences in the network elements that may be monitored in a service environment by providing each per-router monitoring object with a generic interface component. This enables communication with the associated router while maintaining a separation with the logical contents of the monitoring components, thereby showing consideration for interoperability across differing network entities.

Despite these developments, the system concentrates primarily on the monitoring of low-level network traffic, and as such does not consider application-level composition

of data. Furthermore high-level E-Commerce concerns beyond those of IP-based services are not considered (such as interoperability with component technologies). There is also no discussion of SLA languages or how contractual information is represented. The authors use their own Service Level Specification (SLS) language and library of evaluation scripts, with no consideration for those parties that may wish to use pre-defined electronic contracts. There is also no mention of how evaluation or monitoring logic may be augmented to allow extensions to the framework's capabilities.

Also, the generic interfaces used to augment network routers with monitoring components cannot be applied to application servers (for measuring application-level performance metrics), and the active measurements performed by the monitoring components cannot accommodate anything beyond the complexity of low-level network performance.

2.6.3 Academic Research

2.6.3.1 Nahrstedt et al

Nahrstedt et al [Nahrstedt01] describe a middleware approach to QoS provision and monitoring in heterogeneous environments. Service-specific monitoring, control and resource allocation mechanisms are managed within component-based 'QoS-Aware Middleware' inserted below the application level. The middleware identifies separate aspects of the QoS life-cycle, such as QoS specification, configuration and run-time adaptation. The system is composed of different functional components divided into resource-management and service-management groups, capable of operating transparently in conjunction with the managed application. Service specifications are generated with the provided QoS language, and compiled into QoS profiles to be used by the QoS management mechanisms in controlling the service behaviour as viewed by service users. The QoS-aware middleware is pre-installed at every relevant machine prior to a service relationship becoming active, with all relevant proxy objects synchronising when a client initiates a service interaction. This perhaps assumes that relationships between entities will be long-lived – dynamic business relationships would otherwise prompt numerous initialisations, contributing to the processing load placed on proxies and the underlying communication medium alike.

The QoS proxies are capable of dynamically adapting to updated user requirements during the lifetime of an active service instance. Also since the QoS proxies act solely within the context of resource management, they do not necessarily inhibit the performance of active service participants, and are capable of operating in a transparent manner that does not influence the behaviour of the service environment.

The middleware proxy objects provide a generalised QoS provision mechanism with a very simple notion of what constitutes an electronic contract (referred to as the QoS profile), being as it is generated from the QoS requirements of application developers (not necessarily with the inclusion of service clients). Also although the middleware provides for a range of QoS configurations to meet differing service requirements, it does not consider the use of existing QoS description languages, requiring users to employ the authors' own language.

With respect to monitoring capabilities the resource-management components perform some monitoring duties, but there is an assumption that pre-processed application-level QoS metrics are readily available from the relevant server or client interface. Since the system is primarily concerned with low-level resource allocation (albeit with some adaptation of application-level behaviour properties) end-users are not able to define or observe aspects of higher-level system performance.

There is no practical discussion of how the system would be deployed in a heterogeneous environment composed of various operating platforms, and there is no recognition of inter-organisational or business-oriented issues such as ease of system deployment, being as there is only a rudimentary consideration of what constitutes a provider/consumer relationship.

2.6.3.2 Smart Proxies

Another project that utilises proxies is described in [Koster00]. Here 'smart proxies' allow service-specific QoS logic to be added to a client-side application to afford management of QoS properties in situations where service providers offer different application-level communication formats over the same middleware base. These smart proxies allow clients to adapt to different service modes without having to change the application code that they use.

This work concerns itself with relatively low-level service aspects such as the effectiveness of the underlying communication protocol, for instance multimedia

services that provide different file encodings or discernable levels of stream quality. As a result application-level QoS properties are not considered.

The developers have made a number of assumptions in their design. They assume that by providing different QoS logic within the framework of the smart proxies that similar services can then be treated as identical from the view of the client (which is not necessarily applicable to E-Commerce services). In addition, there is an assumption that service- and client-side application logic is developed in isolation. In trying to make development of client-side logic easier further effort is inadvertently required of the service developer, who is expected to provide the adaptive logic of the smart proxies.

A number of factors also work against the effective deployment of these proxies in a wider context. Client applications are expected to communicate directly with the pre-installed proxy components within client machines, which must be explicitly inserted into the communication stack as a first step. Service developers are also expected to develop proxy code on a per-service basis, without scope for automatic reuse of QoS logic across similar services (although the focus in this work is on providing adaptability and re-use of logic at the client-side). Such factors limit the deployment capabilities and potential for scalability during active use, especially with respect to service-oriented environments such as Web services and E-Commerce services.

2.6.3.3 CQoS

The 'CQoS' project [He01] describes platform-specific interceptors used to provide transparent QoS functionality in applications driven by distributed middleware (e.g. CORBA, DCOM and Java RMI).

The project provides a framework for the development of complex QoS control mechanisms which can be deployed in accordance with a standardised interface. This enables reuse of existing QoS logic to meet the specific needs of a growing range of distributed applications. The interceptor components house generic QoS 'Service Component' objects which maintain QoS attributes within the service environment, suggesting the reuse of QoS functionality across similar services. Portability across various distributed middleware platforms is also visited. Existing interfaces are described for use with CORBA and Java RMI, with suggestion that more are feasible.

The range of QoS functionality available in CQoS is limited to fault-tolerance, security and timeliness properties, although there is scope and discussion of providing more in the future. The work also aims to enable different combinations of QoS support based on these attributes, thereby satisfying the varying QoS requirements of different services.

Interceptors are capable of dynamically reconfiguring their QoS capabilities at runtime. This is further facilitated by the ability to obtain additional QoS logic from a designated code repository. This reflects the dynamic nature of service environments within the Internet.

The CQoS interceptors act more as proxies by directly circumventing existing application stubs. Although this does not require modification of existing applications, it does nonetheless require explicit adjustments to the middleware platform. Component reuse and scalability is made possible through use of the Cactus protocol, a modular development framework allowing inter-component communication through event-subscription mechanisms.

2.6.3.4 SLAng

The SLAng project [Skene03, Skene04] attempts to meet the need for a rigorously-defined SLA engine, providing both a means to create electronic contracts and an engine for evaluating performance data against contracts.

SLAng addresses the need to separate the different concerns typically found in an SLA. By providing separable definitions of the service participants, contract-specific information (e.g. lifespan of a contract), and the machine-readable logic governing service behaviour, it is hoped that organisations will be encouraged to approach the negotiation phase of the SLA lifecycle more readily. This then minimises the presence of contradictory terms in an SLA, separating and declaring contract content to aid in the maintenance of contracts. It also affords a measure of reusability with respect to the constituent parts of a contract e.g. for organisations negotiating similar contracts with multiple service customers.

The separation of contract components allows semantic definition data to be used as a reference for service construction, while accommodating various forms of distributed systems (e.g. Web services, application outsourcing, storage-hosting). There is also an

acknowledgement of the basic contractual requirements within industry, as such providing applicability in real-world scenarios.

To establish easier deployment of SLAs a choice was made to embed the SLAng language in XML. As well as enhancing the rigidity of contract definitions this was a conscious acknowledgement of the widespread use of XML within distributed systems. The semantic representations within SLAng are modelled in the Unified Modelling Language (UML), and behavioural constraints defined using the Object Constraint Language (OCL). As an aside, this combination potentially allows for the development of new contractual languages and associated evaluation criteria other than those found in the SLAng project. When combined with the Model Driven Architecture embraced by the developers of SLAng these choices act to separate business models from technical implementations, thereby enabling reusability of contract components across various systems.

In consciously addressing the need to make electronic contract development easier for all involved, the developers of SLAng have provided a robust SLA engine which reduces the ambiguities in both negotiating and realising a service contract. However, SLAng does not immediately offer compatibility with existing SLA languages.

There are further issues relating to integration with existing monitoring frameworks; there is a need to develop 'hooks' to allow established processes to interact with the SLAng evaluation engine. With regards to scalability there is no discussion of the capabilities of the evaluation engine with respect to an arbitrary number of contracts within the same service environment. Considering that the SLAng engine carries out its own contract evaluation, there is no evaluation of its performance in processing monitoring data or how scalable it is with respect to an arbitrary number of (potentially simultaneous) requests for measurement evaluation.

2.6.4 SLA Monitoring Requirements

The examinations documented in Sections 2.6.2 and 2.6.3 regarding current SLA monitoring solutions served to highlight a range of issues that need to be addressed.

2.6.4.1 Contractual Heterogeneity

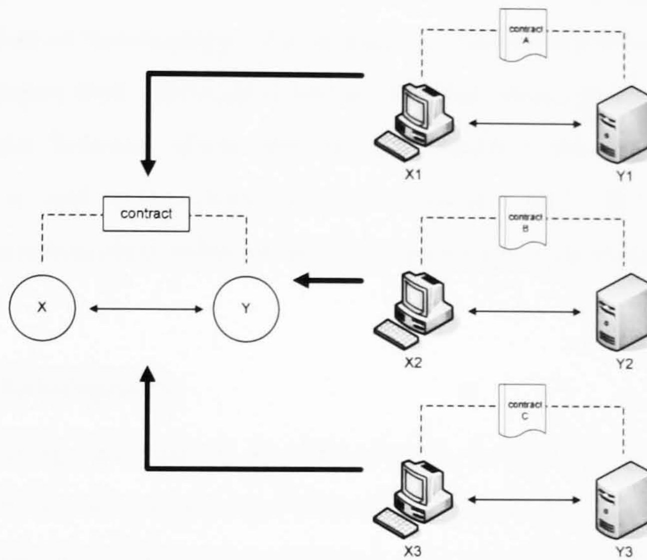


Figure 24 Applying a generic contract representation to dissimilar business relationships

Existing research into SLA monitoring provision highlighted the need to accommodate both new and existing requirements for QoS measurement within the terminology of an electronic contract [KellerIbm02, Sahai02]. Coupled with this is the consideration of how technical contract content can be measured and evaluated. In essence this implies a requirement to accommodate the QoS requirements of any E-Commerce scenario.

In practice this can be achieved by creating a contract engine capable of accommodating any dynamic service environment through updated measurement capabilities. Alternatively provisions can be made to allow for any existing contract specification language to be integrated into the monitoring framework, presumably through a generic interface. The former approach, although essentially simpler to envisage, does not consider that organisations may choose to use a variety of products

to fulfil their different business requirements. As such, they or their business partners may already be using a particular contract-specification engine before choosing to integrate it into a monitoring infrastructure. The latter approach has the potential to allow organisations to use existing contract specifications without the need to re-interpret contract content for use in a different engine. By its nature this would also separate contract definition and performance monitoring duties, making maintenance of the QoS infrastructure more manageable.

Another advantage of generic SLA monitoring is that it would inherently include some standardisation of terminology. If a monitoring framework is to monitor say, a metric called 'response time', it needs to know what that means in no uncertain terms (e.g. response time between client and server, response time between requests entering the server and being processed and returned, etc.). In this sense, the framework must accommodate different monitoring requirements without ambiguity.

2.6.4.2 Domain Heterogeneity

Many SLA monitoring solutions are targeted towards specific service dynamics e.g. Web service-based one-to-one provider/consumer relationships (e.g. [Sahai02]). As E-Commerce services and Web services evolve it is entirely feasible that more diverse service relationships will be realised. If a monitoring framework exists that separates monitoring functionality from service structure, it can be applied to different service types and different service domains (such as media applications and multiplayer online games).

A domain-neutral monitoring infrastructure can conceivably change to match the dynamics of the service that it is monitoring, thereby accommodating the various service permutations that can be seen in the Internet and other large networks. As an example, this would negate the need to re-deploy a monitoring infrastructure when existing service participants engage in interactions using different service dynamics.

2.6.4.3 Accommodation of Enabling Technologies

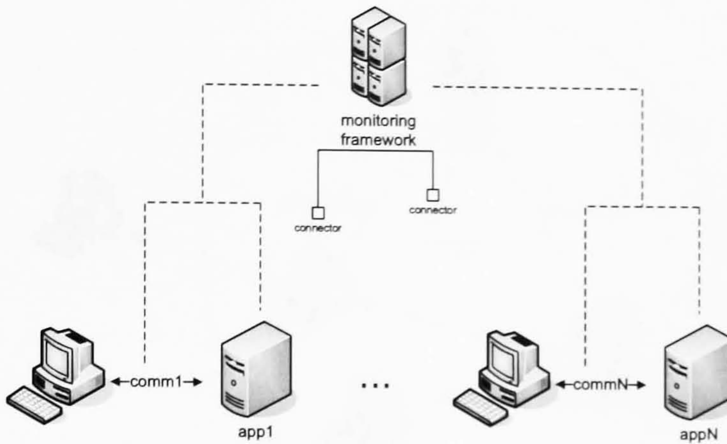


Figure 25 Applying a monitoring framework to different communication and application technologies

Many monitoring solutions are targeted towards integration with specific application software and communication technologies e.g. [Sahai02, Asgari03]. Although this eases monitoring on a per-system basis it does not naturally mirror the heterogeneous nature of service environments in large open networks such as the Internet. Tying a monitoring framework to specific technologies without scope for adaptation also reduces the capacity for reuse of monitoring components.

Another advantage of a monitoring framework that can operate across any combination of communication protocol and application software configurations is that it is likely to be capable of adapting to serve new and developing technologies as they become available. This is especially important in light of the continuing evolution of the Internet. By creating a monitoring framework that can be applied to different service technologies there is also an inherent capacity to serve a wider range of services, thereby potentially increasing adoption of the technology. It also reduces the need to modify service hosting platforms to accommodate monitoring capabilities.

2.6.4.4 Scalability towards Participant Entities and Service Contracts

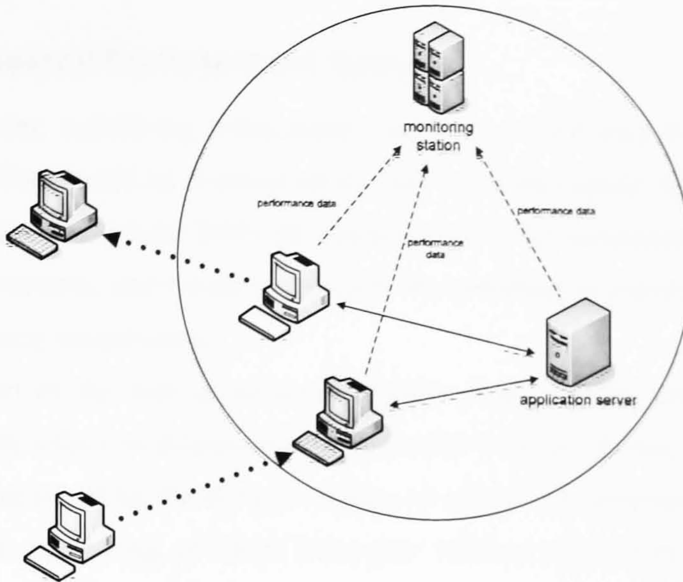


Figure 26 Service clients dynamically entering and leaving a service environment

It has been proposed in the related texts [Asgari03] that if a monitoring infrastructure is to be successful within E-Commerce services (and more notably Web services) it must be able to observe an arbitrary number of service clients. It must also be able to operate without adversely affecting the performance of the observed service environment.

When an organisation chooses to offer a service, the desirable outcome is that the user base for that service grows over time. If service monitoring is being carried out, it is logical to expect the monitoring framework to observe the behaviour of a growing number of service clients, while at the same time following how services behaves in relation to each individual client. Internet-based services have the potential to include a boundless number of service participants. It is also advantageous when developing monitoring components to account for clients entering and leaving service relationships whenever they choose.

Consideration must be given to the activity of collecting measurement data so that it does not detrimentally affect the observed system. That is to say that there is potential to manage contracts in a scalable way, particularly if management is conducted from a

centralised location or if for instance overlapping sets of measurement data are being produced from identical sources.

2.6.4.5 Transparent Deployment and Operation

When positioning monitoring components around a service and its participants, a minimum of effort should be required on the part of the participant organisations. The service participants will most likely be concerned with the maintenance of their own service infrastructures, and would presumably be unwilling to apply further effort to deploy monitoring components.

Minimum effort on the part of service providers when enabling monitoring would ensure a smooth rollout of monitoring infrastructure without slowing the core service deployment process. With the dynamic nature of service relationships there is also a need to enable monitoring of client behaviour without interrupting the actions of individual service clients (for example by requiring them to install monitoring software etc). This is especially important considering that clients are typically able to enter and leave service relationships at will. By reducing the amount of monitoring infrastructure required at both the server- and client-side the process of enabling service monitoring can be expedited.

Once a monitoring infrastructure is deployed it should require as little maintenance on the part of the service participants as is realistically possible. Faults do undoubtedly occur, and it cannot be guaranteed that the service participants will have the knowledge required to fix them. Reducing the number of locations at which monitoring components must be installed (while isolating their internal logic from the observed service) will ultimately reduce the influence that the monitoring infrastructure has on the behaviour of a service environment and its participants.

2.6.4.6 Ease of Deployment and Modularity

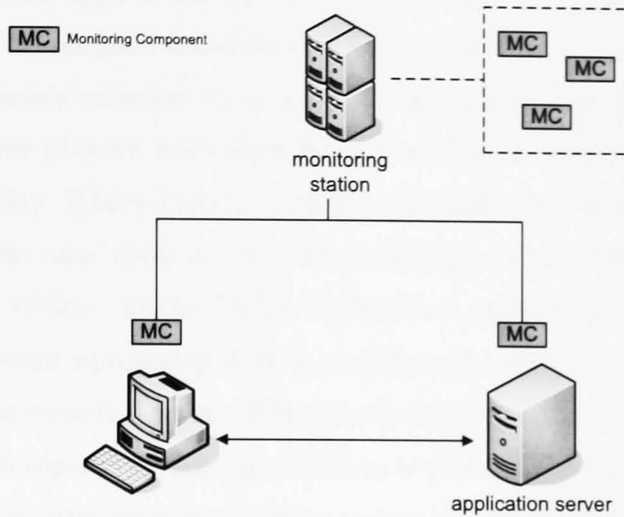


Figure 27 The system components that must be considered when deploying a monitoring infrastructure

A monitoring framework should require as little effort as possible to tailor monitoring and evaluation mechanisms to the intricacies of a particular service environment. Many services use similar technologies but differ in matters of detail (e.g. named service participants, specific specialised monitoring requirements). As such it would also be advantageous to allow reuse of functional components and evaluation logic. These steps would make deployment simple and efficient, while reducing the need to rewrite or hand-code component logic repeatedly for each monitored service.

Modular monitoring infrastructures have already been exhibited by previous solutions (e.g. [KellerIbm02]) as a sound means of enabling per-service alterations to a generalised monitoring framework. A modular approach allows reuse of monitoring logic, reducing the need to recreate large portions of logic for each observed service. It also negates alteration of either the service environment that it is being applied to or the components of the monitoring framework itself, thereby affording simpler dispatch of monitoring capabilities.

2.6.5 Distributed Virtual Environments

As Massively Multiplayer Online Games (MMOGs) and other Distributed Virtual Environment (DVE) applications grow in popularity, developers are increasingly choosing to offer subscription-based provision of online realms (e.g. [Wow]). There is a need to continuously maintain the underlying support infrastructure, meet the needs of the growing base of users, and ensure that the service that users are paying for is of a reasonable quality. Where there is a need to determine the quality of the service provision as in this case, there is scope for monitoring of network- and application-level QoS characteristics within DVEs. Furthermore, there is a need to be able to evaluate the gathered monitoring data in a meaningful way (i.e. at the application level) so as to determine the quality of the service as perceived by its users.

There is as yet no concrete definition of how to evaluate the performance of a DVE. There has also been little discussion of how system properties relating to performance and behaviour can be actively gathered in a meaningful way to demonstrate higher-level quality aspects within a DVE.

2.6.6 Summary

Through study of both E-Commerce services and Distributed Virtual Environments a number of similarities have been found. The participation of users is inherently dynamic in both cases. Entry into a service relationship or a virtual world can be initiated and terminated in a loosely-coupled manner where users are essentially able to join and leave at will. Such behaviour requires a scalable service environment that can accommodate any number of end-users interacting with the system at any one time, which in both domains could run into the hundreds or thousands. Any attempt to monitor the behaviour of service participants and providers must be able to scale in a similar fashion.

Another similarity between E-Commerce services and DVEs is that they both have application-level QoS requirements to observe. In either domain the preservation of service quality is not simply a case of monitoring network-level performance characteristics. There is a shared need to gather metric data from various parts of the observed system, collate this information, and process it in order to illustrate some semblance of how the system is behaving at a higher level that relates to the end-user

experience. Centralised post-processing such as this would need to be automated in such a way that it does not interfere with the workings of the observed system, while providing adaptability to the monitoring needs of individual service environments.

A further consideration in both service domains is that service environments have the potential to encapsulate functionality behind cross-platform messaging mechanisms (be they Web service protocols or messaging middleware). This requires that a monitoring framework be able to adapt to the respective characteristics of the application protocols used within a service environment.

There are a number of differences between E-Commerce systems and Distributed Virtual Environments which must also be taken into account. The typical enabling technologies in both domains are different. E-Commerce systems employ Web service technologies to achieve interoperability between participants, whereas DVEs rely mostly on existing network-level protocols, and in some cases Message-Oriented Middleware (MOM). There are contrasting differences between these communication mediums which must be considered.

Another aspect to consider is where quality is perceived in each domain. With respect to E-Commerce services, the focus is on providing consistent and reliable processing of service data, regardless of how it affects the actions of end users. In contrast DVEs are directly driven by the quality of the end-user experience, which encompasses timeliness and consistency.

The monitoring of electronic contracts in an E-Commerce environment composed of different SLA languages and dissimilar middleware platforms is not possible using existing approaches. Furthermore, the automated generation of code specifically for metric data gathering, although desirable and progressed by [Asgari03] [Sahai02], is not realised as yet.

With respect to DVEs, it has been shown that the quality of a virtual environment is measured by how it reflects changes to the state of the virtual world in real-time, and the consistency with which these changes are presented to users. However there is at present no means of defining how timeliness and consistency translate to the end-user experience in either their constitution or their evaluation.

2.7 Outline of Goals

A number of issues have been raised with respect to the provision of QoS monitoring and evaluation within scalable, heterogeneous Internet service environments. The following goals aim to address these concerns:

1) Develop a low-cost approach to SLA monitoring that requires a minimum of tailoring to match the needs of individual systems, and which maintains a minimal level of intrusion with respect to the observed service environment.

2) When considering the various permutations of distributed service environments, it would be advantageous to develop a general-purpose monitoring infrastructure capable of gathering metric data in a platform-agnostic, scalable manner, with scope for application across different service domains (i.e. E-Commerce and DVE applications). This would provide an alternative to the development of per-service QoS monitoring solutions.

3) There is a distinct lack of application-level QoS definitions and evaluation criteria in the domain of Distributed Virtual Environments. This encompasses how missed interactions can be measured and evaluated, and what constitutes consistency between users of a virtual world.

4) If DVEs are to become better regulated with regards to user-perceived quality, techniques for evaluating associated QoS parameters (including their higher-level constituents) must be developed to enable automated evaluation of service performance.

2.8 Summary

- Distributed services are typically realised as provider/consumer relationships where the provider creates, deploys and maintains a service that is offered for consumption to other entities with access to the same network.

- The Quality-of-Service (QoS) associated with a service is a declaration of how certain characteristics should be maintained to guarantee that the service performs as intended. QoS definitions are especially important with Internet applications, where combinations of divergent network elements can contribute to potentially unpredictable operating environments. QoS definitions can refer to basic network-layer properties, or application-layer service attributes (in the case of service environments that embody more human-perceivable qualities, such as Web services).

- QoS monitoring provides a view of system performance, so that QoS compensation processes can be enacted in an informed manner. Monitoring usually requires software or hardware components to be inserted in any number of locations across the provider/consumer communication path. This may include the server or client platform or elements of the underlying network, depending on the performance metrics being observed. Monitoring of existing or injected service traffic helps to determine the performance of different components within the service environment. Careful consideration must be taken to ensure that QoS monitoring does not adversely affect the performance of the service environment under observation.

- QoS evaluation involves gathering measurement data from the various monitoring components within an observed service environment and collating the associated performance data into a form that reflects the behaviour of the complete service environment. This informs decisions regarding QoS provision, and allows for composite metrics to be derived for use in high-level evaluation processes, such as Service Level Agreement (SLA) evaluation.

- Distributed services can be utilised in a number of ways, including socket-layer port addressing and cross-platform Remote Procedure Calls (RPCs). Web services focus on interoperability between different service technologies, with notable technologies including the SOAP communication protocol and Enterprise JavaBeans (EJBs). Communication between participating parties can also be enacted using Message-Oriented Middleware (MOM), which provides for scalable, loosely-coupled communication. Group-based and peer-to-peer communication can both be achieved using MOM.

- Distributed Virtual Environments (DVEs) are virtual worlds inhabited by geographically-dispersed users who are able to interact with each other. As DVEs support greater numbers of users and become more popular, there is a need to make them scalable. One means of achieving this is with Interest Management techniques, which act to reduce the number of messages sent between participating machines by only informing relevant parties of localised activities. The use of Interest Management techniques does however have the potential to introduce missed interactions into the DVE, wherein insufficient messages are transmitted between interacting entities for their interactions to be properly recorded. It is conceivable that QoS monitoring could be of use in controlling the effects of missed interactions upon DVE performance.

- E-Commerce services and DVEs share a number of basic QoS requirements, but also differ in a number of ways. Their similarities and differences highlight the challenges that exist for any QoS monitoring and evaluation framework intended for use within heterogeneous service environments. Through study of these two service domains and the related work, conclusions can be reached as to the qualities required of the monitoring components developed within this thesis.

3. E-Commerce

Many organisations conduct business with each other over computer networks (most notably the Internet). With strict requirements for order and process, there is a growing need for these interactions to adhere to predictable behaviour patterns. This should be accompanied by clear identification of the respective obligations of each participating party. Monitoring of participant behaviour and performance informs these processes.

3.1 Introduction

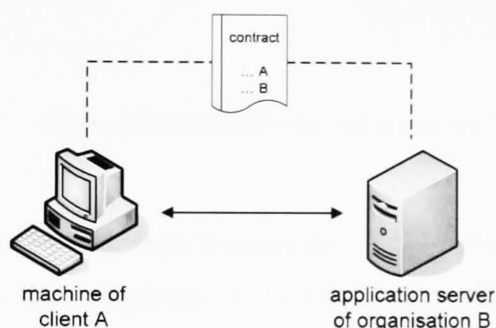


Figure 28 How contracts bind service participants

The contractual obligations of interacting organisations and the expectations of their interactions are increasingly being described within electronic contracts referred to as Service Level Agreements (SLAs). SLAs specify the Quality-of-Service (QoS) attributes associated with the interactions between a service provider and service consumer.

SLAs can be used to precisely define the contractual obligations of each party involved in a service relationship. For example, it can be stated that a client can send a limited number of requests for service in a given timeframe, or a server application must process each incoming request within a specified amount of time. Another advantage of SLAs is that the evaluation of service-related obligations has the potential to be automated. Processes can be developed to gather measurements from

the service environment and directly compare measurement data to the applicable contract content. This then reduces the maintenance demands of enforcing an SLA.

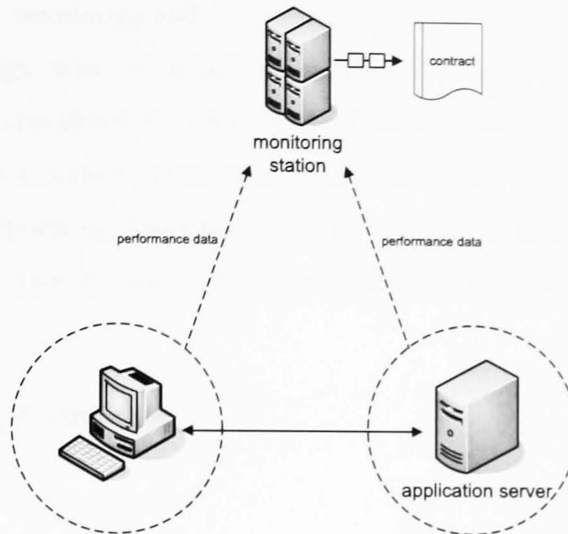


Figure 29 Enforcing an electronic contract

Monitoring of a service environment is necessary for collecting metric data, which is then used to evaluate the compliance of interacting parties in accordance with an associated SLA (Figure 29). This typically involves the insertion of specialised monitoring components across the same stretch of network as the service participants, to allow observation of the interactions between them. Monitoring logic can also be inserted at either end-system to provide a greater perception of how each service participant is behaving. Monitoring data is periodically collected from monitoring components and gathered at a centralised monitoring station for automated processing and comparison against the service-specific terms of the associated SLA. Any violations of service terms can be discovered as part of the latter process and (if required) appropriate compensatory action taken to rectify problems or alter the business relationship.

3.2 SLA Monitoring Architecture

Previous research at Newcastle University by Jimenez et al [Jimenez04] describes a number of the SLA monitoring and evaluation requirements identified in Section 2.6. A framework design was proposed that covers the fundamental issues of SLA monitoring: SLA specification, separation of computational and communication infrastructure, service-related points of presence, and approaches to metric collection. There is also description of components that are capable of gathering and processing system-wide metric data for evaluation against the terms of an SLA.

3.2.1 Monitoring Architecture

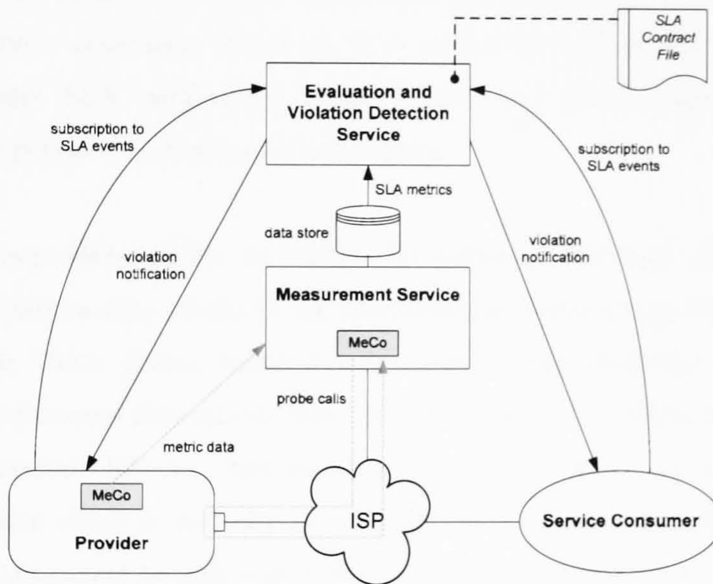


Figure 30 Architecture for the unilateral monitoring and enforcement of inter-organisational SLAs

The proposed SLA monitoring architecture is shown in Figure 30. Only unilateral service provision is considered, as opposed to a bilateral service environment wherein interacting parties would provide services to each other simultaneously. With unilateral service provisioning there is a need to observe two distinct sets of contractual obligations; the QoS obligations of the service provider to the consumer, and the service consumer's obligations to the provider (which dictate how the consumer is expected to use the service).

The components shown in Figure 30 assume responsibility for SLA monitoring and evaluation:

- *Metric Collectors (MeCos):* these components gather metric data associated with the performance and usage of the observed system.

- *Measurement Service:* receives metric data updates from the MeCo components and performs limited post-processing. Data is then stored in a repository for access by the Evaluation & Violation Detection Service.

- *Evaluation & Violation Detection (EVD) Service:* inspects gathered metric data to determine if any SLA violations have occurred, and informs interested parties of violations. In Figure 30 the interested parties are the service provider and service consumer. When an SLA enters the system the EVD service propagates SLA parameters to the MeCos and Measurement Service to provide per-service monitoring capabilities.

The MeCo components shown in Figure 30 gather metric data relating to the provider's obligations (the MeCo in the Measurement Service) and the consumer's obligations (the MeCo placed within the provider's server platform) based on the monitoring requirements they receive from the EVD Service. This data is then relayed to the Measurement Service. MeCos may be realised as a set of distributed components based either in software or hardware. In the system described in Figure 30 the MeCo component housed within the Measurement Service acts to impartially determine the performance of the service provider. Conversely, to infer the behaviour characteristics of the service consumer the MeCo component deployed within the server platform gathers metric measurements pertaining to request processing. Together these two MeCo components provide a composite view of how interacting parties acting within a shared service agreement are performing in relation to their respective obligations.

The Measurement Service MeCo utilises active monitoring. Synthetic load is generated by a simulated client to determine if the provider is satisfying the conditions of the associated SLA contract(s). An alternative to probing of this kind

would be to have a MeCo co-located with the consumer to gather metric data associated with genuine consumer requests. However for various reasons it cannot be assumed that monitoring components can be deployed at the client-side. For instance, clients may be opposed to maintaining monitoring infrastructure.

The SLA monitoring infrastructure is designed to minimise the interference caused to the existing service environment while at the same time providing meaningful monitoring data. The monitoring components are realised as discrete sub-systems that can be deployed on relevant platforms while addressing issues of effective component placement and deployment discretion.

There follows discussion of the concerns that the proposed monitoring framework needs to address (as detailed in Sections 2.6.1 & 2.6.4).

3.2.2 Scalability Considerations

A generic SLA monitoring infrastructure must accommodate various service environment permutations with respect to how the number of constituent entities within the environment changes. This includes factors such as the number of service clients, active services, monitored contracts, and monitoring components within the service environment.

In the simplest case, within an infrastructure such as the one described in Figure 30, there will be one service provider and one service consumer under observation. It may be necessary to define an approach to accommodating situations wherein a single service provider has agreed to process requests from a single service consumer across numerous services. Service operators may negotiate a number of SLAs with a client across the same portion of network space, providing numerous different services.

There is scope for aggregation of measurement updates from monitoring components and consolidation of active measurement messages for those services that offer identical or extremely similar functionality. These steps could reduce the synthetic network traffic and server load generated by the monitoring framework.

Another permutation includes a single provider serving multiple service consumer entities. A single SLA is then defined for each (potentially identical) service negotiated between the service provider and each service consumer. If all of the provider/consumer relationships refer to services with identical internal logic, active monitoring could be carried out on behalf of all of the consumer bodies (essentially

sharing a probe component between all of them). It is assumed with this however that the Measurement Service MeCo component is directly connected to the same network as all of the clients, and it is then able to approximate an experience similar to that of each client [Jimenez04]. There is then a stipulation that the Measurement Service MeCo can only conduct measurements on behalf of multiple clients if they are all connected to the same network. To make use of this opportunity there would however need to be an agreement between service consumers to use the same MeCo unit within the same Measurement Service instance.

It is possible that in this scenario each consumer would want its own view of service provider performance concerning satisfaction of their own requests. This would require scheduling of active measurements on behalf of each consumer, potentially from distinct MeCo units within separate Measurement Service components. There is also the issue of how to deploy a MeCo (or set of MeCo units) within the server platform to enable monitoring of a set of consumers that can potentially grow or shrink arbitrarily. The core issue here is whether each service consumer should be afforded its own MeCo inside the service provider platform, or whether it could be argued that the functionality required within a MeCo will be identical for all services with identical observation criteria. This also again depends on whether the internal logic of the service(s) associated with each client is identical. Sharing a MeCo between identical services would save monitoring resources and make the system more manageable. Service participants may ultimately demand an individual MeCo on principle if a competitor is using an identical service on the same server and does not wish for performance metrics to be indirectly exposed (since the performance of one service would then be indicative of the performance of the other).

Rudimentary replication of Measurement Service components and Evaluation & Violation Detection (EVD) Service components associates a single Measurement Service with each Evaluation & Violation Detection (EVD) Service. Such infrastructures may exist if parties delegate monitoring to trusted third parties capable of both metric measurement and SLA evaluation. These third parties may be entrusted with monitoring of separate sets of QoS metrics or may act to cross-check performance data.

A specialised version of the latter arrangement is where there are multiple Measurement Service instances and a single EVD Service. In this case each Measurement Service should only have access to the contractual information it needs

to operate, separating concerns and reducing the existence of extraneous contract information in the system (as discussed in [KellerIbm02]). Also, extra strain may be placed upon the MeCo component within the service provider platform if it is required to identify and transmit metric data intended for a number of Measurement Service instances.

Each one of the Measurement Service nodes may also need to probe the service provider at intervals (with their combined timetables potentially intersecting). The network activity attributable to probe calls should be managed responsibly as it will increase with the number of Measurement Service nodes. Otherwise performance within the service environment may be degraded.

Another specialised system permutation has one Measurement Service collecting information on behalf of a number of Evaluation & Violation Detection (EVD) Service components (which may be evaluating different portions of the same SLA). Parties requesting SLA violation notifications will require a scalable subscription management platform to cope with any number of EVD Service notification events. For instance an organisation may be partaking in a large number of service relationships with different business partners.

3.2.3 Deployment Considerations

There may be numerous parties interested in SLA violations associated with a single service, such as different departments of the same company, shareholders etc. These parties should be able to dynamically receive violation notifications and process them in whichever way they choose. SLA violation data may be used to alter resource allocations or may simply be logged for future reference in business negotiations. Interested parties should not be tied to specific application logic in this context, and as such a Message-Oriented Middleware (MOM) notification mechanism seems most appropriate (not forgetting also that MOM technologies are naturally scalable).

There is the issue of how the monitoring framework can facilitate the changing characteristics of an E-Commerce service environment. As participants enter and leave a (potentially evolving) service relationship different monitoring capabilities may be required to monitor a service. If the monitoring framework is required to stay online at all times it must be capable of dynamically updating the set of contracts and metrics it is monitoring.

The proposed monitoring framework is modular in nature. This provides potential for monitoring and evaluation logic to be reused across different services (e.g. the same metrics can be monitored in different server software using the same previously-written logic components).

3.2.4 Heterogeneity Considerations

Monitoring components should be able to operate transparently over any application technology within a service platform. With respect to the MeCo within the Measurement Service it should also be adaptable to any E-Commerce communication protocol. This would allow it to emulate service client calls to a service provider in any given E-Commerce service environment.

Once metric data has been collected by the Measurement Service it must be organised into a suitable format for handling by the EVD Service. However organisations may use different SLA languages, so the EVD Service must be capable of interfacing with any variety of SLA languages (e.g. [Ludwig02], [Sahai01], SLAng [Skene04]). This should be accomplished in a manner that does not require changes to the core logic of the Measurement Service. An appropriate approach would be to construct the Measurement Service to work with arbitrary SLA languages with minimum tailoring. Each SLA identifies the types of metric data to be evaluated within the associated service. Therefore, automatic generation of code to translate metric data to a format suitable for processing by an SLA-language dependent evaluation tool is required. Processes for the translation of metric measurements to SLA obligation terminology can be embodied in a specialised interface class. Such an interface class can then be loaded into the Measurement Service to form a bridge with the appropriate SLA language or associated contract engine (through exposed data retrieval and update methods). Characteristics such as obligation definition and measurement criteria differ between SLA languages, and would need to be considered on an individual basis.

There is an additional need to consolidate the different monitoring requirements of disparate service environments i.e. a mechanism should be created to map different obligation terminology to monitoring component logic.

3.3 Implementation

I implemented and developed an SLA monitoring infrastructure [MorganIfip05] based upon the framework design outlined in [Jimenez04] and described in Section 3.2.1. In this section the implementation is described in detail, with reasoning given for the structure of the monitoring framework and its components.

3.3.1 Overview

A monitoring framework was developed as illustrated in Figure 31:

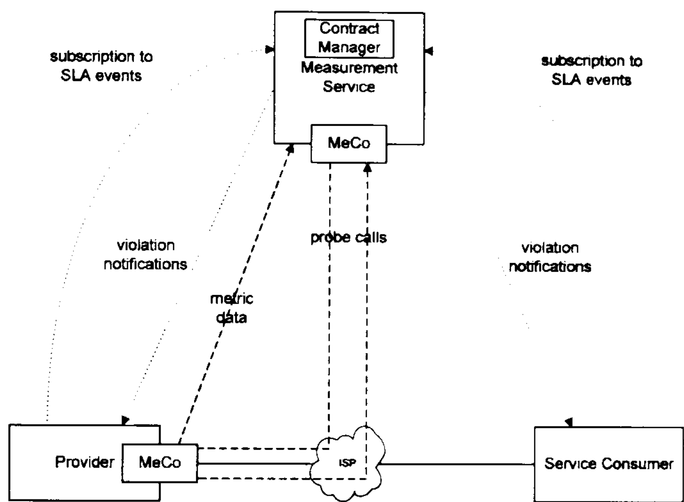


Figure 31 SLA monitoring architecture

The focus of this implementation is the monitoring of unidirectional Web service provision and the provider/consumer relationship encapsulated therein. It is assumed that the provider and consumer are linked via a single Internet Service Provider (ISP). Multiple ISPs are not considered as there are no guarantees that inter-ISP service quality can be maintained or reliably monitored and attributed to any one participant.

The SLA monitoring framework illustrated in Figure 31 is a set of separate, modular components which when combined allow per-system tailoring of SLA monitoring processes. A modular approach to implementation allows for component modification, to meet various monitoring requirements without the need to alter and redeploy the monitoring framework as a whole.

Each of the components in this architecture has a specific purpose:

- *Provider-side Metric Collector (MeCo):* intercepts consumer requests (and associated service responses) and records measurements relating to a service consumer's usage of the provider platform. These measurements are sent to the Measurement Service component for post-processing and evaluation. See Section 3.3.3.

- *Measurement Service Metric Collector (MeCo) / MeCo Probe:* observes the performance of a service provider by assuming the role of a simulated service consumer. Using active measurement this MeCo component emulates an actual service client, periodically generating synthetic client load. This allows monitoring of the service provider's obligations (i.e. QoS provision) by observing the conditions inherent in the client experience. With this approach there is no need for clients to maintain monitoring logic, which in itself would be impractical with an arbitrary number of service consumers with the potential to dynamically enter service relationships. See Section 3.3.4.

- *Messaging Service:* the underlying messaging infrastructure across which metric data and SLA violation notifications are propagated. See Section 3.3.5.

- *Measurement Service:* collates metric data gathered from the Provider-side MeCo and MeCo Probe then interprets and evaluates it against the relevant SLAs (and if need be informs service participants of any associated SLA violations). Returning to the conceptual system described in Section 3.2.1 and illustrated in Figure 30, the implemented Measurement Service incorporates the Evaluation & Violation Detection Service and the 'data store'. See Section 3.3.6.

- *Contract Manager:* a sub-system of the Measurement Service that reads contract-specific configuration data into the framework from monitored SLAs. This data is used to calibrate the various components towards monitoring of specific metrics. The Contract Manager is also capable of detecting SLA

violations given metric data supplied by the Measurement Service. See Section 3.3.6.

In the following sub-sections there are descriptions of the components of the implemented framework, with explanations of how different components collaborate to provide SLA monitoring, evaluation and violation notification capabilities.

3.3.2 Implementation Assumptions

A number of assumptions are made regarding the environment within which the MeCo Framework is deployed, and the application technologies, communication protocols and services it interacts with:

- It is assumed that the MeCo Probe is connected to portions of network maintained by the same Internet Service Provider (ISP), and is free to probe the service provider from any location as long as it does not leave the domain of this ISP [Jimenez04]. It is assumed that probe calls instigated from the same ISP domain would be indicative of the client experience. It is assumed with this that the QoS properties of the service participants are all that is required to evaluate the terms of an SLA: the performance of individual network nodes or routers along the communication path between the provider and consumer is not of concern.

- It is assumed that in every observed service environment the Measurement Service and all its internal components are maintained and operated by a Trusted Third Party (TTP) agreed upon by all service participants. This ensures that the service participants will respect the SLA evaluation results generated by the framework. The TTP may also take responsibility for keeping the capabilities of the Provider-side MeCo up-to-date.

- It is assumed that each SLA engine exposes methods to allow other technologies to interface with it. This includes permitting other programs to obtain SLA details and to input monitoring data into the engine for evaluation.

- The actions of service clients can be correlated with their SLA obligations through identification of their IP address (including specific port address if required).

- The service platform is assumed to be maintained by and under the control of a service provider or affiliate. It is also assumed that the monitoring data collected by the Provider-side Metric Collector will not be manipulated by the operator of the service platform.

- Service providers can be identified from within their respective server platforms, and that text-string identifiers can be assigned to each server platform. It is also assumed that the latter identifiers are unique across all provider platform instances under observation.

- SLA contract files can be uniquely identified amongst all other contracts. It is assumed that this is achieved through a globally unique entry in each contract file, which can be obtained using the associated SLA contract engine.

- Any Provider-side MeCo wrapper implementations exploit message interception features built into the enabling technology. It is assumed that the communication between service participants is enacted over middleware technologies that support transparent message interception. This is a valid assumption as all major middleware technologies used in E-Commerce applications provide a mechanism for message interception e.g. interceptors in CORBA [CorbaNS], handlers in SOAP [Axis], and interceptors in Enterprise JavaBeans (EJB) containers [Ejb]). It is also assumed that wrapper implementations can infer any QoS metrics required by service contracts.

- It is assumed that the Measurement Service MeCo Probe is able to simulate and thereby observe typical client behaviour accurately through the use of active measurement techniques. However only single-part requests are replicated by the MeCo Probe.

- Any communication protocols being observed through the MeCo Probe can be accessed through Web Services Invocation Framework (WSIF) [Wsif] interfaces. See Section 3.3.4.

- Performance measures relating to server-side QoS can be inferred through active measurements conducted by the MeCo Probe.

- If multiple services are being monitored by a single probe instance, the same probing interval is required by all of the associated SLAs.

- It is assumed that no two consumer requests are processed at the same time within the observed server platform, and with this that the Provider-side MeCo does not need to accommodate concurrent monitoring of multiple requests.

3.3.3 Metric Collector (MeCo) Interceptors (Provider-Side)

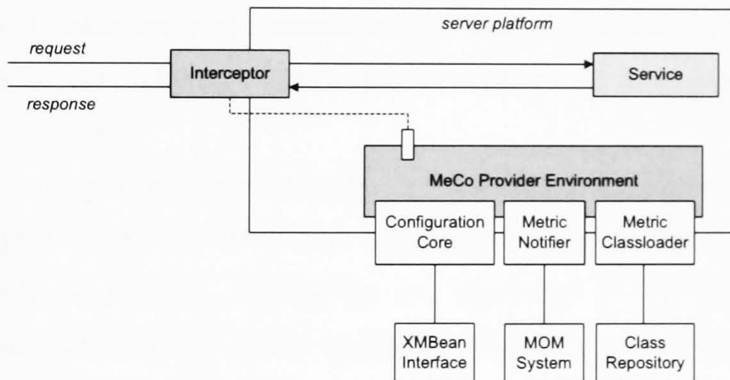


Figure 32 Provider-side MeCo placement

There are aspects of SLA evaluation that require a view of server-side performance, and of how client behaviour affects this performance. To provide this view, localised measurements can be taken from within the server platform hosting the observed services. These measurements can then be gathered for further analysis against SLAs. The Provider-side MeCo (Figure 32) gathers metric data from within an observed server platform based on service usage, and propagates it to the Measurement Service

for evaluation against the relevant SLA(s). A MeCo unit is deployed within the service platform of the service provider (as identified in the SLA), to monitor usage of the associated service(s) by service consumers.

Arbitrary application technologies are supported with the use of MeCo 'hooks'. It is assumed that 'interceptor' mechanisms (as discussed at various points in Section 2.6) are available to allow transparent and non-intrusive examination of request and response messages for each observed service. With this it is assumed that a specialised interface implementation can be derived for any application technology being used (in Figure 32 this refers to the 'Interceptor' at the entry point to the internal server logic).

Once deployed, additional monitoring logic (contained within the 'Provider Environment' as shown in Figure 32 – see Section 3.3.3.2) determines which QoS metrics to gather. This is achieved through consultation of an internal configuration core initialised remotely from the Measurement Service component (see Section 3.3.3.3). The configuration information governs which metric measurement classes to load from the associated class repository. Tailored monitoring capabilities are provided through the creation of measurement class instances based on SLA contents.

A Provider-side MeCo exists as a combination of platform-specific 'wrappers' and a series of metric data classes, coordinated by centralised processing logic (the 'Provider Environment').

Each MeCo wrapper interface includes a specialised implementation of the 'MecoInterceptor' interface, which is a template for the operations that a MeCo should be able to perform. Interceptors are registered to be included in the request/response stack of their native application language, and are given access to request objects as they pass in and out of the server. Two methods are included in the generic interface:

```
public abstract void setRequestResult(String metricName, Object requestValue);

public abstract Object getRequestResult(String metricName);
```

These methods respectively associate a metric measurement with a specific identifier, and recall a measurement through its identifier. Initialisation methods within the native application technology are exploited to initialise the MeCo Interceptor implementation and the 'Provider Environment' upon server start-up.

3.3.3.1 MeCo Interceptor Implementations

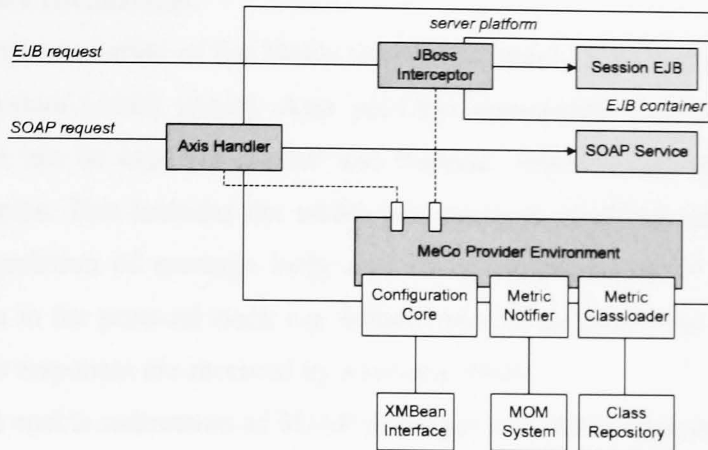


Figure 33 Implementation of Provider-side MeCo's

The use of MeCo hooks has been demonstrated to support Web services using SOAP and Enterprise JavaBeans (EJBs) using Java Remote Method Invocation (Java RMI) [Javarmi], as shown in Figure 33. This preliminary set of MeCo implementations mirrors the combination of these two approaches in many complex E-Commerce applications working within the Java 2 Enterprise Edition (J2EE) architecture [J2ee] (a popular open-source system used for the development of enterprise computing solutions). This combination also provides evidence to illustrate how different measurement capabilities can be enabled both in isolation and in combination to satisfy monitoring requirements (with further details provided in Section 5.1).

The J2EE specification describes a generalised platform for Web-enabled applications using Java Server Pages (JSPs) [Jsp], Servlets [J2eeservlet] and EJBs. Java application servers (referred to as J2EE servers) must cater for all of these technologies. J2EE Web services present services for inter-organisational communications with back-end application logic based in EJBs.

A specific implementation of the J2EE server platform (the JBoss application server [Jboss]) was used to act as the test-bed server. JBoss can be used to combine J2EE-compliant technologies and other Java-based server technologies (e.g. SOAP-based services) in a single server instance.

JBoss Interceptors are used to implement MeCo hooks suitable for interception of Java RMI invocations (the 'EJBMecoInterceptor' implementation used to observe EJB messages). With this approach EJB application logic need not be modified to enable service monitoring. Only small modifications are required within the JBoss Interceptor stack declarations.

The SOAP implementation of the MeCo wrapper is based on the Apache eXtensible Interaction System (Axis) [Axis]. Axis provides interceptors (referred to as Axis Handlers) that can be used for request and response interception, and alteration of message contents. This includes the addition or removal of SOAP message headers and the manipulation of message body content. Interception can be carried out at specific points in the protocol stack e.g. before requests are processed by server-side logic or before responses are received by a service client.

Axis Handlers enable redirection of SOAP messages to a MeCo component for metric data gathering (via the 'SOAPMecoInterceptor' class). The use of Axis Handlers does not require alterations to existing application logic. Only slight modification of the Axis Handler declarations described within the supporting server platform are required. Other SOAP-based MeCo wrapper implementations could potentially be employed, but must provide transparent message interception mechanisms analogous to Axis Handlers. Deployment to a J2EE server platform should also be possible.

3.3.3.2 Provider Environment

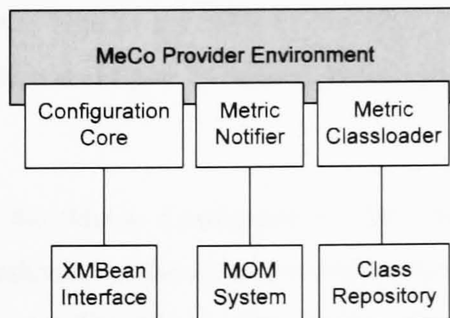


Figure 34 MeCo Provider Environment

The MeCo wrappers integrated into an observed service platform are tied to the same core component, the Provider Environment (shown in Figure 34). This component has knowledge of the QoS metrics to be monitored (through remote configuration from the Measurement Service – see Section 3.3.6.1 for further details). It is also able to use this information to load the required metric-measurement classes from a known class repository (a 'library' of Java class files within the JBoss file system). This approach enables dynamic creation of MeCo functionality on a per-SLA basis, as additional classes can be added to the server file library and retrieved by the Provider Environment on demand.

The MeCo Provider Environment contains a number of sub-components:

- *MeCo Configuration Core:* calibrated through communication with the Measurement Service (see Section 3.3.3.3). The configuration core holds information about the operations being monitored and the measurements to be taken within the Provider-side MeCo. Once initialised the core disseminates relevant calibration information to other parts of the Provider-side MeCo.

- *Metric Notifier:* manages message-passing events and message channels between the Provider-side MeCo and the Measurement Service. The Metric Notifier packages and transmits QoS measurements to the Measurement Service once they have been gathered. See Section 3.3.5 for more information.

- *Metric Classloader:* creates instances of the requisite metric measurement classes as requested by the MeCo Configuration Core. Each class instance represents a metric type as specified by the SLA within the contract engine (e.g. response time measured in milliseconds). This is further described in Section 3.3.6.1.

Upon initialisation of the MeCo Configuration Core the Provider Environment informs the Metric Classloader of the measurements to take regarding incoming and outgoing service messages. The Classloader obtains class definitions with names matching those of the associated metrics, and create instances of these classes dynamically using Java reflection techniques. Each such class is an implementation of the 'DynamicMeasurementInterface', which has the following methods:

```
public Object requestChannel();

public Object responseChannel();

public Object getResult(Object request, Object response);
```

When a request is intercepted by the MeCo the 'requestChannel' method of each measurement class instance is called, and the results of calling this method temporarily stored in the associated MecoInterceptor instance. Responses are processed by the 'responseChannel' method of each measurement class instance, with the final metric measurement obtained using the 'getResult' method. The latter is a single measurement for a completed request that can be delivered to the Measurement Service for analysis. The measurements taken from the request and response channels are given to the 'getResult' method once a request leaves the server, for final measurement calculations to be conducted.

When measurement classes are dynamically loaded into the Provider Environment they are stored upon initialisation and referenced as instances of the interface, negating further use of Java reflection and thereby reducing reference overheads. This approach also allows the monitoring capabilities of the MeCo to be updated while it is operational, thereby maintaining availability of monitoring capabilities.

3.3.3.3 Provider-Side MeCo Deployment and Initialisation

The server-side MeCo component is deployed on the JBoss platform in a specialised manner, exploiting specific features of the JBoss platform to provide greater functionality for the MeCo Framework. The deployment consists of a small set of core files:

- *'meco-core.jar'*: stores the general processing logic of the Provider Environment.
- *'meco-dynamic.jar'*: file library containing measurement class definitions.

- *'measurement-service-mbean.sar'*: a packaged application that enables remote access to the MeCo Configuration Core from the JBoss JMX console [Jboss] from any location with an active Measurement Service instance.

The latter files are 'dropped' into the relevant locations within the JBoss server instance and automatically loaded, persisting until they are explicitly removed. If the server instance is shut down and subsequently restarted they are automatically reloaded. In packaging MeCo functionality in this way it is hoped that deployment is made relatively simple in the eyes of prospective users of the monitoring framework. This approach also allows for simple updating of core files, as files can simply be replaced with newer versions which are automatically loaded by the JBoss server. It is for this reason that the 'meco-dynamic.jar' class library is separated from the core files. It is envisaged that the measurement capabilities of the MeCo will be augmented as more services are supported, even allowing application developers to add their own measurement classes.

Additional configuration steps may be required to enable monitoring of a particular communication medium. For instance, the MeCo Framework identifies clients using their network IP addresses. To achieve identification within EJB services the 'ClientIPInterceptor' class is declared in the JBoss Interceptor stack configuration alongside a reference to the 'EJBMecoInterceptor' class. For SOAP services the 'SOAPMecoInterceptor' is declared as an Axis Handler for observed services within the 'server-config.wsdd' Axis [Axis] deployment file. Client identification in SOAP can be achieved by interrogating request objects as they are passed through the stack.

A Java Management eXtensions (JMX) service [jmx], the 'MecoMBean Control', is deployed within the same JBoss server instance as the MeCo Interceptor(s). This allows limited external control of MeCo Interceptor configuration. The service contains a JBoss XMBean [Jbossxmbean] that facilitates control of some basic MeCo characteristics from a JBoss JMX Console [Jbossjmx]. This enables, for example, service monitoring to be switched on and off from the Measurement Service.

Each service provider must be given an identity in the co-located Provider Environment, to identify it in SLA clauses during SLA evaluation and for verifying the origin of messages that arrive at the Measurement Service. The MeCoMBean provides the means to remotely configure the service provider identifier in the MeCo.

Automated configuration of Provider Environment attributes e.g. client identifier lists and operation-to-SLA bindings (described in Section 3.3.6.1) occurs when the Measurement Service is activated. This maintains the principle of centralised control and dissemination of configuration information, thereby reducing inconsistency within configuration information across the distributed monitoring components of the MeCo Framework. Although this means that there is no permanent record of configuration data at the service-side (in case of server failure), in such circumstances failure would be 'graceful'. The result of a server failure would be a halt in updates to the Measurement Service, thereby indirectly suggesting that the service had ceased operation. Active measurements from the MeCo Probe (see Section 3.3.4) would also detect a non-responsive server, potentially constituting a potential violation of the associated SLA. Also, the lack of dedicated Provider-side MeCo configuration files reduces the resource footprint within the server platform.

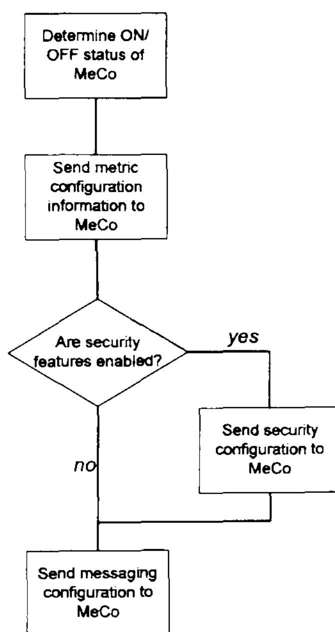


Figure 35 How the Measurement Service configures the Provider Environment

The initialisation of the Provider-side MeCo from the Measurement Service is described in the flowchart of Figure 35, with additional details of how the Provider Environment behaves during initialisation calls shown in Figure 36.

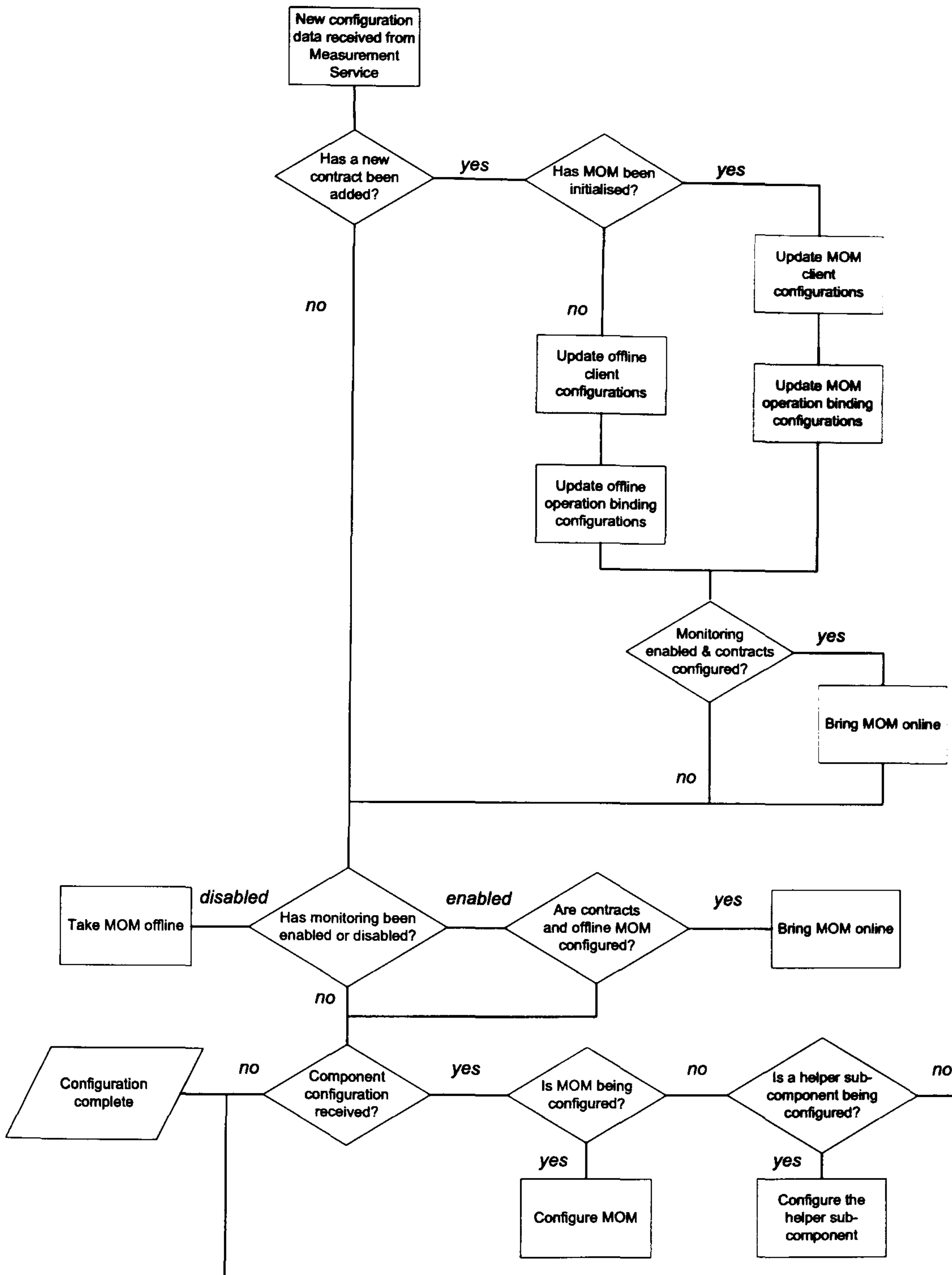


Figure 36 How the Provider Environment is configured based on Measurement Service initialisation actions

If monitoring is disabled the messaging sub-component is taken offline so that undue localised processing within the Provider Environment is avoided (along with unnecessary use of MOM resources). However active probing measurements can still be carried out from the Measurement Service, enabling service monitoring in those instances where internal server performance is not considered (i.e. probe calls alone can be used to infer the QoS measurements described in the associated SLA).

3.3.4 Metric Collector (MeCo) Probe

Active measurements from the MeCo Probe provide a means of assessing the experience of service clients without requiring them to deploy and maintain monitoring components. Simulated service requests are sent to an observed service platform and QoS measurements inferred from the condition of the response. Examples include whether a response is late in arriving or whether the response content contains an HTTP error etc.

The MeCo component in the Measurement Service differs from the MeCo Interceptor located within the service platform in that it periodically sends probe messages to the service provider. This allows collection of metric data related to how a service provider appears to be behaving from the viewpoint of a service consumer, without interfering in the workings of the consumer entity. Alterations to the consumer platform would inhibit dynamic service behaviour and require the service client to maintain part of the monitoring infrastructure. In this respect the Measurement Service MeCo Probe acts much like a synthetic client, but need not be regarded as a real one. MeCo Interceptors are coded so as not to process requests that are identified as having come from the MeCo Probe.

The probing strategy associated with each active probe is located in a Web Service Definition Language (WSDL) file [Wsdl]. This file describes how to enact automated communication with a Web service, and as such can be used to configure the MeCo Probe to send valid messages to the target service. The Java class instances required to enact probing are determined by parsing additional extensibility elements in the given WSDL file. These elements also allow a finite set of input parameters to be configured, which the MeCo Probe can select from at random with every invocation (thereby modelling client behaviour more realistically). Complex request interactions

that use context-sensitive information are not modelled (e.g. if a single request is a sequence of messages that depend on intermediate responses from the service).
Along with service configuration data, the WSDL configuration file for each MeCo Probe contains specialised message-parsing configuration information:

- *Probe Format:* the communication protocol used by the service (e.g. EJB).

- *Probe Method:* the specific method that the probe should target.

- *Return Type:* for when a response is returned by the service provider, so the contents can be properly processed.

- *Probe Arguments:* these include (for each argument) a parameter name, parameter value, and indication of the object type (to ensure correct encoding).

After synthetic requests have been dispatched responses are processed by the Measurement Service to provide additional measurement data for potential use in evaluation of contractual obligations.
As with the platform wrapper in the Provider-side MeCo, a platform-specific wrapper is used for implementing the MeCo Probe (for example EJB/RMI or Web services/SOAP). The MeCo Probe was built using the Web Services Invocation Framework (WSIF) [Wsif]. The WSIF framework uses protocol-specific WSDL extensions to provide heterogeneous access to Web services through WSDL service descriptors. This allows the MeCo Probe to be configured to match any service type that WSIF can support (including scope for support of additional protocols).
An XML-based configuration file associated with each MeCo Probe (the 'probe descriptor file') indicates which service it is monitoring (i.e. which WSDL configuration to use) and which service platform to send service requests to. The contents of a sample probe descriptor file are show in Figure 37.

```
<?xml version="1.0" encoding="UTF-8"?>
<probe-config
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:/eclipse/workspace/MeCo/misc/probe-config.xsd">

<!-- FIBONACCI EJB -->
<wsdlDefinition>file:///C:/eclipse/workspace/MeCo/misc/WSDL/Fibonacci_EJB.wsdl</wsdlDe
finition>
<wsdlNamespace>urn:Fibonacci_EJB</wsdlNamespace>
<service>FibonacciEJBService</service>
<contractIDs>fibonacciSLA_EJB</contractIDs>
<providerID>A</providerID>
<probeMetrics>
   <probeMetric>
      <metricName>ejbResponseTime</metricName>
      <metricTitle>EJBResponseTimeCLIENT</metricTitle>
      <metricUnit>mS</metricUnit>
   </probeMetric>
</probeMetrics>

</probe-config>
```

Figure 37 Sample Probe Descriptor File

To aid scalability in scenarios where multiple clients use identical services with the same service provider, a MeCo Probe can be configured to send messages on behalf of all of the clients. Multiple contract identifiers can be listed in the `<contractIDs>` element of a probe descriptor. This can help to reduce unnecessary duplication of probe calls and results.

3.3.4.1 Probe Descriptors

A probe descriptor file can describe additional measurement data to be gathered. This works much in the same way as the dynamic measurement classes used in the Provider-side MeCo component. For each type of measurement there are descriptions of the action to perform both before a request is sent and once a response arrives, as well as how to correlate the request and response data into a meaningful result. This accommodates dynamic configuration of the measurement capabilities of the MeCo Probe. The probe initialisation stage also ensures that probes only activate once the associated contract(s) have been read into the system.

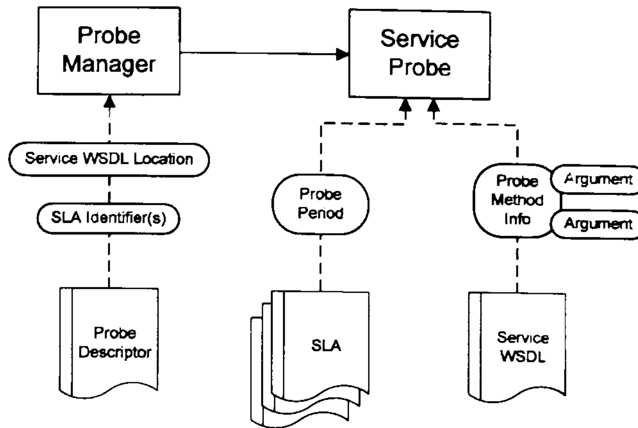


Figure 38 Probe initialisation process

Figure 38 describes the amalgamation of probe configuration information during creation of a new service probe. A top-level Probe Manager inside the MeCo Probe component manages all 'Service Probe' instances. When a new 'Probe Descriptor' file is added to the file system the Probe Manager extracts information from the descriptor. This includes the identifier(s) of the SLA contract(s) that will evaluate the probe measurements, and the location of the WSDL file that describes how to interact with the observed service. The Probe Manager creates a Service Probe configured to contact a specific service method using a particular set of input parameters. These are obtained from additional extensibility elements within the 'Service WSDL' file identified in the probe descriptor. This configuration information is then encapsulated in a 'Probe Method Info' object and associated 'Arguments' inside the Service Probe. In the implementation the interval between Service Probe activations (the 'Probe Period' as in Figure 38) is dictated by the maximum acceptable server down-time defined in the SLAng SLA language (see Section 3.3.6.1).

3.3.5 Messaging Service

Measurements taken from the observed service platform must be propagated to the Measurement Service for analysis. There is also a need to be able to notify interested parties when SLA violations occur. Both of these must be achieved in a scalable and reliable manner. The Messaging Service is provided to enact communication between the distributed components of the MeCo Framework. The Messaging Service passes

metric data from the Provider-side MeCo to the Measurement Service. It also supports the transmission of SLA violation notifications to interested parties.

The MeCo Framework is built upon a Message Oriented Middleware (MOM) design paradigm. The software used to enable communication between the distributed components of the MeCo Framework must support publish/subscribe communication patterns. For the implemented system the Java Message Service (JMS) [Haefal01] was used. This conceptually allows for a great number of Provider-side MeCos and Measurement Service instances to communicate with each other, or for numerous simultaneously operating third-party agents to monitor the same range of services.

3.3.5.1 Event Notification within the Messaging Service

The JMS API supports both point-to-point and publish/subscribe models of interaction, with the latter chosen for use with the Measurement Service. Publish/subscribe topics are used on a per-operation basis. A 'metric' topic is associated with the name of each operation defined in an SLA, to which data related to that operation is transmitted. This approach balances sending data associated with each contract (producing few messages of greater size) and sending each individual piece of metric data (resulting in a greater number of fragmented service usage updates). Since both the SLA language engine and the MOM interface can be tailored to individual systems, it is quite conceivable that a different message-management format can be developed other than the operation-based model described.

The publish/subscribe approach provides the opportunity for multiple SLA engine instances to be integrated into the monitoring framework. Existing SLA engines often lack scalability when required to evaluate increasing numbers of SLAs. The capacity to employ numerous Measurement Service instances connected to message channels offers potential to improve scalability and share processing load. In addition, different SLA language engines can be used at the same time (which for example could ease transitional periods between use of existing and replacement SLA engines).

The decoupled nature of MOM minimises disruption to the monitoring infrastructure. Components simply register as consumers for the metric data topic(s) pertaining to the contracts they are monitoring.

When an SLA evaluation indicates that a contract obligation has been violated, interested parties can be made aware of the event. Scalable message dissemination is

desirable here as it is possible that an arbitrary number of organisations may be interested in the performance of a particular service (e.g. business partners, investors etc.). Propagation of SLA violation notifications again uses message topics. Here, topics are used on a per-SLA basis, uniquely identified by any piece of text which separates one contract from another (it is assumed that the SLA language provides this distinction). Organisations then assume responsibility for subscribing to topics that refer to the SLAs they are actively participating (or simply have an interest) in. SLA violation notifications consist of the name of the metric that was violated and the value that caused the violation. Additional information generated by the SLA engine can also be included if methods to access this information are exposed.

3.3.5.2 Implementing the Messaging Service

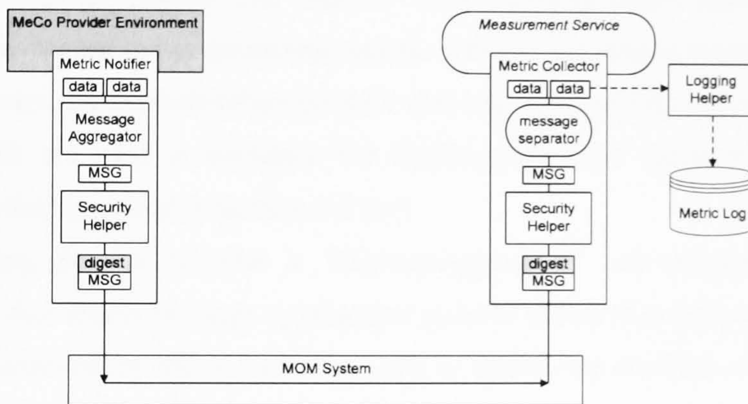


Figure 39 How measurement data is processed between the Provider-side MeCo and the Measurement Service

Different notification middleware technologies (aside from JMS) can be used for the underlying communication mechanism of the MeCo Framework. Integration into the Messaging Service is achieved by providing implementations of the 'MetricNotifier', 'MetricCollector', and 'ViolationDetectorEnvironment' interfaces. These interfaces define inter-component communication methods for the Provider-side MeCo Provider Environment, the Measurement Service evaluation engine, and the violation notification subsystem of the Measurement Service respectively.

The methods of the 'MetricNotifier' interface are:

```
public abstract void updateMeasureService(HashMap metrics, String clientID, String
topicID, String protocol);

public abstract void updateClientConfig(String cID);

public abstract void updateClientConfigurations(HashSet cIDs);

public abstract void updateBindingsConfig(Object slaOpBindings);

public abstract void updateBindingConfigurations(HashMap slaOpBindings);

public abstract void updateProviderID(String provID);

public abstract void initialise(Properties props, HashMap helpers);
```

The 'updateMeasureService' method is called by the Provider Environment to transmit metric measurements. The 'metrics' Java HashMap object stores key/value pairs of metric names and measurement values, while other parameters identify who called the service, the publish/subscribe topic that was used, and the service protocol. Other methods are used to configure the implemented class (these configuration details are further described in Section 3.3.6.1).

The Messaging Service includes a 'MessageAggregator' sub-component at the provider-side that enacts message aggregation policies across relevant metric topics. One method exposed by this interface is used to update the contents of a message before it is transmitted to the Measurement Service:

```
public String updateMessage(String nNotification);
```

The policies used to group messages are defined within dynamically-loaded classes retrieved through Java reflection techniques, thus allowing for new message aggregation policies to be added to the MeCo system over time. The implementation already contains aggregation policies defining sequence-based message grouping (aggregating a pre-specified sequence of messages before transmission) and temporal message grouping (grouping messages over finite periods of time and transmitting them when the time-frame elapses).

Additional logging and security classes can be configured for use in the messaging subsystem. By implementing the 'MecoLoggingHelper' and 'MecoSecurityHelper'

interfaces tailoring of data logging processes and basic verification features is possible. The 'MeCoLoggingHelper' interface declares methods for logging both metric notifications and SLA violation data. These methods are:

```
public abstract void logMetrics(HashMap logProps, HashMap metricMap) throws MecoLoggingException;

public abstract void logViolations(HashMap logProps, Set violations) throws MecoLoggingException;
```

The 'logMetrics' method takes in configuration information for connecting to the log, and a Java HashMap of metric identifiers and measurement values. The 'logViolations' method takes a list of violations and then disseminates them to interested parties. The 'MeCoLoggingHelper' interface has already been implemented with MySQL [Mysql] to allow storage of logging data in a database

The security helper interface has methods for encrypting message text into message digests, and verifying digests using key-pair authentication. Its methods are:

```
public String encryptText(String messageText) throws Exception;

public boolean verifyContent(String messageText, String signedDigest, String senderID) throws Exception;
```

These methods are used to encrypt the content of a message as a digest, and verify the origin of a message respectively. The 'encryptText' method uses the local server identifier within the Provider Environment to identify the security key to use when encrypting a message digest for transmission with monitoring data. The 'verifyContent' method uses the public key of the entity identified by the 'senderID' (the server that sent the message) to verify the 'signedDigest'.

Digests can be used to determine the authenticity of metric notifications (fortifying communication integrity between the observed server and a potentially third-party Measurement Service). An implementation of this interface has been created using the Java Keystore encryption technology [Javakeystore].

3.3.6 Measurement Service

Measurements from within the observed service platform and from service probes must be correlated in a centralised manner to provide a consistent and complete view of service performance. At the same time management of measurement capabilities and SLA evaluation procedures should be centralised, to enable dissemination of consistent configuration state across the monitoring framework. All of this is embodied in the Measurement Service component.

The Measurement Service receives measurement monitoring data updates from both the Provider-side MeCo and its own internal MeCo Probe. Monitoring data is evaluated against the relevant contracts. If during the evaluation process it is found that a contractual obligation has been violated, the Measurement Service can notify any interested parties of the violations (via the relevant SLA topics). The Measurement Service contains a number of specialised sub-components (Figure 40):

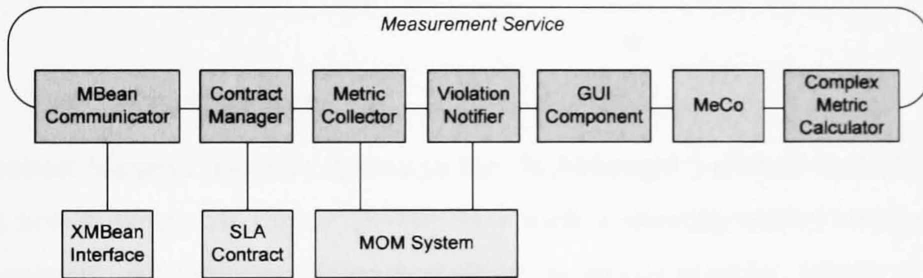


Figure 40 The Measurement Service and its sub-components

- *MBean Communicator:* connects to the Provider-side MeCo within the JBoss service platform, allowing activation/deactivation of server-side measurements and transmission of measurement configurations to the Provider-side MeCo.

- *Contract Manager:* evaluates metric data received from the Metric Collector, MeCo Probe, and the Complex Metric Calculator. The Contract Manager can create SLA violation notification messages for transmission by the Violation Notifier.

- *Metric Collector:* manages subscription to topics associated with observed contracts (as instructed by the Contract Manager). It also translates update messages into a format suitable for processing by the Contract Manager.

- *Violation Notifier:* used by the Contract Manager to format and disseminate SLA violation information to those parties registered to receive them.

- *GUI Component:* displays measurement and violation information in charts and lists that are updated as data is processed by the Measurement Service (see Section 3.3.6.4).

- *Complex Metric Calculator:* creates composite metrics based on measurements from the Provider-side MeCo and the MeCo Probe. See Section 3.3.6.2 for a description of 'complex' metrics.

3.3.6.1 The Contract Manager

The Contract Manager interface (known as the 'SLAManager') provides methods for reading new contracts into the monitoring framework, evaluating metrics based upon those contracts, and obtaining information about the service provider, service clients or observed operations defined in each contract.

Two contract engines have been implemented. These can be interchanged or replaced with future implementations as per the generic SLA language mechanism (as long as new implementations adhere to the 'SLAManager' interface).

Both of the existing implementations use the SLAng modelling language [Skene04] as a base. The contract language defined in SLAng is used to interpret contracts and obtain configuration information for the monitoring framework. One of the implementations relies on the SLA evaluation components of SLAng (the 'SLAngManager' class). The other ('SLAngSimpleEvaluationManager') uses SLAng for contract definition purposes only and augments it with purpose-built evaluation logic (thereby avoiding use of the memory-intensive evaluation components of SLAng). Just as the Contract Manager is generic in nature it is also capable of allowing different features of individual SLA engines to be reused or combined with new processing logic. This enables service-specific tailoring of SLA evaluation

processes in the same way that the MeCo monitoring mechanisms can be customised to meet individual needs.

An example of a SLAng contract, as used in the MeCo Framework, is provided in Appendix B. More information concerning the layout of SLAng contracts can be found in [Skene04]. Contracts such as the one found in Appendix B are read into the SLAng contract engine. An interface with this engine then allows the Measurement Service to obtain details contained within the contracts. SLAng contracts define one-to-one provider/consumer relationships. The SLAng engine exposes operations for evaluating measurement data against contracts stored within an instance of the engine. Further information regarding how contract contents are exposed in SLAng can be found in [SkeneEdoc04].

The SLAng contract language is capable of measuring 'maximum latency', 'maximum time to repair', 'reliability' and 'maximum throughput'. Within the context of the MeCo Framework these metrics are taken to mean 'request processing time within the application server', 'permissible server downtime', 'server availability' and 'maximum permissible client request throughput' respectively. Access to these parameters is exposed by the contract engine through provided methods. The interfaces to the contract engine were written to use these methods to obtain the details of specific metrics. For example, measurements conducted in relation to the 'request processing time within the application server' associated with a contract will be controlled with those methods that expose access to the 'maximum latency' parameter within the SLAng contract engine. Metric units are also declared for each measurement type.

Interfaces that bind the Measurement Service to a specific contract engine must all be created so as to knit the metrics exposed by the contract engine to the capabilities of the MeCo Framework. There must also be an understanding of the contract engine and the terminology of the associated SLA language. This may suggest that only the developers of a particular SLA language or SLA engine are in a position to create interfaces for use in the Measurement Service, although this is not definite.

In a SLAng contract threshold values are stipulated for each parameter. A violation of one of these parameter thresholds constitutes a violation of the SLA obligations as a whole. The 'SLAngManager' interface implementation uses the SLAng contract engine to evaluate measurements against the content of a SLAng contract directly. This happens retrospectively and does not consider allowances for multiple obligation

violations, but is also extremely resource intensive. As an alternative, the 'SLAngSimpleEvaluationManager' interface implementation evaluates measurement data through use of its own internal logic, without the SLAng contract engine. It can also record multiple obligation violations, thereby sharing responsibility for service violations between both the contract engine and the complete monitoring framework.

Where parameter thresholds are defined for specific service operations, operation names are extracted by the Measurement Service. These are then bound to SLA identifiers, and used in the Provider-side MeCo to determine which operations are to be monitored as they are called by clients. Determining the identity of the client associated with the SLA obligation that references each operation allows the Provider-side MeCo to observe specific client/server interactions. When received at the Measurement Service, measurements are correlated with the appropriate SLA content through reference to the aforementioned SLA-to-operation bindings.

Monitoring logic is written to measure each metric described in an SLA engine. Where an SLA engine interface implementation accesses methods exposing SLA metrics, it is able to communicate to the Measurement Service the monitoring classes that are necessary to monitor the SLA. This information is conveyed to the Provider-side MeCo upon configuration as the 'Measurement Types' (described further in Section 3.3.6.2).

Every contract loaded into a Measurement Service instance is interpreted by the same contract engine, and the contract engine is stipulated in the Measurement Service configuration file (Section 3.3.6.3). Other Measurement Service instances with different contract engines can subscribe to the same topics if there is a demand for multi-engine contract verification.

3.3.6.2 Measurement Service Contract Configuration

When a new contract is added to the system the MeCo Framework determines which measurement classes to load within both the Measurement Service and the Provider Environment. Three distinct categories of configuration data are extracted from each contract (with data extraction processes dictated by the logic within the associated SLA engine interface):

- *Client Identifiers:* sent to the Provider-side MeCo so it is aware of the clients that must be monitored from the associated server platform. Client identification allows correlation of server activities and SLA obligations to determine client behaviour.

- *Measurement Types:* describe the metrics to be measured within the Provider-side MeCo and the MeCo Probe. The set of measurement classes loaded into the system is determined by the metrics identified within each contract. For example if an SLA clause includes details of a threshold limit on the processing time for client requests as measured in milliseconds at the server, and this is referred to as the request 'procTime', the Contract Manager can request that an instance of the 'METRICprocTime_mS' Java class be loaded into the Provider Environment. A list of the metrics that are observed within the SLAng contract engine is provided in Section 3.3.6.1.

- *Client-Measurement Bindings:* these mappings allow the Provider Environment to carry out per-client performance measurements on top of the per-protocol behaviour of the MeCo Interceptors. A list of measurement types is associated with each client identifier, which then dictates which metrics to observe with each processed request based upon the originator of the request.

- *SLA Operation Bindings:* it is assumed that each operation defined in a contract is universally unique and with this that each operation (and the associated SLA) can be identified as it is being monitored. A reference to the associated SLA allows the MeCo system to correlate measurement data with contract obligations during the evaluation process. Information regarding operation and contract identifiers is also used to configure metric topics and SLA topics respectively.

The measurements initiated within the MeCo Framework are arranged into subsets for manageable separation of per-component metric-collection concerns:

- *Basic Metrics:* application-level performance measurements taken at the server-side. These measurements are transmitted from the Provider-side MeCo to the Measurement Service for post-processing and evaluation. The SLAng engine interfaces dictate that measurements of 'request processing time' are conducted at the server-side. This metric is then referred to as a 'basic' metric.

- *Probe Metrics:* measurements taken from the MeCo Probe. These are referenced in each probe configuration and linked to specific contracts. The SLAng interfaces refer to 'permissible server downtime' as a 'probe' metric.

- *Complex Metrics:* composed from basic metrics (and potentially probe metrics), these measurements are produced from complex calculations carried out within the Measurement Service. In general 'complex' metrics may be used to provide a more immediately representative indication of system performance within the charts produced by the Measurement Service GUI. SLA parameters may be interpreted differently to indicate more obvious trends that the end-user (i.e. system administrator) finds easier to read in a data plot. Alternatively if the SLA engine implementation incorporates additional evaluation logic (as in the 'SLAngSimpleEvaluationManager' implementation), complex metrics can be used to calculate per-service metrics based on complete service usage. As an example, to assist the 'SLAngSimpleEvaluationManager' request throughput is calculated as a complex metric where otherwise the underlying SLAng engine would have conducted the calculation. This approach can also be used to move complexity away from the Provider-side MeCo.

The Contract Manager is capable of monitoring a designated contract folder and updating both its own configuration and that of the Provider Environment when a new contract has been added to the system. Details of this are found in Section 3.3.3.3.

3.3.6.3 The Measurement Service Configuration File

The 'measurement-service.xml' configuration file is arranged to accommodate the adaptive nature of the MeCo Framework. A global configuration section within the file lists the class-name prefixes of the Java classes used for each sub-component implementation in the Measurement Service. Additional configuration sections (identified by the same class-name prefixes) contain initialisation parameters for each sub-component. For example if the 'messaging' element in the main configuration section has the value "JMS", the Measurement Service will create an instance of the 'JMSMetricCollector' class, and the MeCo Provider Environment will create an instance of the 'JMSMetricNotifier' class. The "JMS" section of the configuration file will be examined for initialisation parameters for these classes, such as which ports to use to connect to the JMS server etc.

There is a set of general configuration headers within the configuration file which point to additional class-specific configuration information:

- *'slaEngine':* describes the SLA contract engine. Within the class-specific section there is a parameter to describe the directory which the Measurement Service should poll for new contracts. Another parameter controls whether the system transmits SLA violation notifications. If this entry is not found it is assumed that violation notification mechanisms are disabled.

- *'messaging':* describes which messaging implementation to use in the Messaging Service.

- *'logging' (optional):* the logging class to use.

- *'security' (optional):* the security and authentication class to use when transmitting and receiving updates.

Configuration information is also included for the MeCo MBean Communicator (i.e. how to contact the JBoss server – see Section 3.3.6), MeCo Probe sub-component (which directory to observe for probe descriptor files – see Section 3.3.4), the set of

complex metric classes available to the system, and the graphical interface sub-component (see Section 3.3.6.4).

3.3.6.4 Measurement Service Visual Component

It is presumed that a single organisation or trusted third party is tasked with managing the monitoring and evaluation processes relating to a service relationship. Appropriate members of the organisation should be able to observe measurement and evaluation events in real-time, so as to be able to react to particular events in a timely manner. For this purpose, a graphical component is attached to the Measurement Service to present data as it is received and processed from across the MeCo Framework.

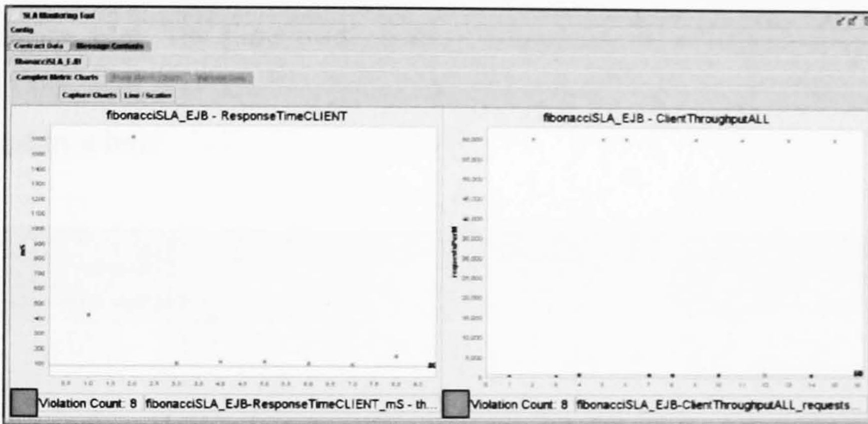


Figure 41 The GUI chart window

Once the Measurement Service is operational an interface is created as shown in Figure 41. This interface is intended for use by an administrator of the monitoring process. If other parties wish to view data appropriate measurement logging features can be enabled accordingly.

For each contract loaded into the system an individual interface tab is created. For each per-contract tab, there is a group of 'Complex Metric Charts' showing graphs for probe metrics and 'complex' metrics measured by the system (see Section 3.3.6.2). There is also a 'Basic Metric Charts' window tab displaying charts of raw measurement data as collected from within the Provider-side MeCo, and a tab for the 'Violation Data' relating to each contract. Within the charts that are displayed, red

horizontal lines on a data plot denote the threshold value for the associated metric (where a threshold is defined by the associated contract).

The x-axis of each of the basic and complex metric graphs is request-based (as opposed to temporal-based). Charts are updated whenever new measurement data is processed by the Measurement Service. The 'Message Contents' tab however provides information relating to exactly when measurements were made. If a violation of a metric threshold occurs the colour of the 'traffic light' associated with the relevant chart turns to red as an indicator. Information relating to the SLA violation appears briefly under the chart (so as to provide instant notification to a system administrator).

The 'Capture Charts' button can be used to create screenshots of all of the charts (the destination folder for these screenshots is defined in the 'measurement-service.xml' configuration file). The Line/Scatter button changes the presentation of data in the charts so that each data point is represented as a dot or all of the points in a chart are connected by a line.

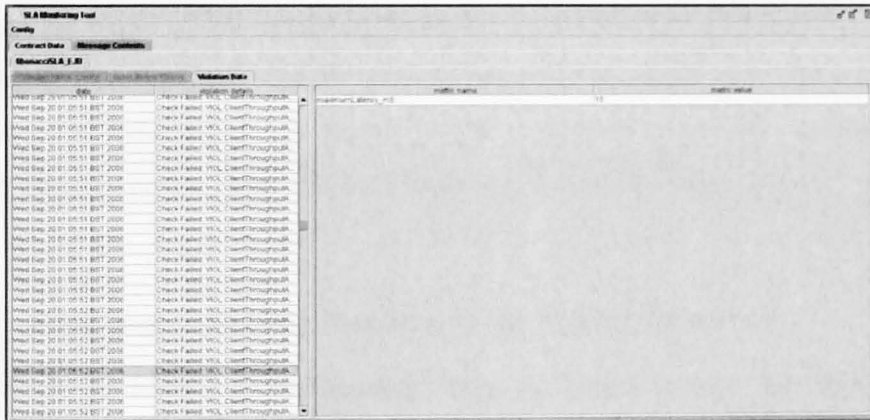


Figure 42 The GUI Violation Data window

The 'Violation Data' tab window (Figure 42) displays information about any SLA contract violations that occur for a contract while the monitored service is being used during the lifetime of the Measurement Service instance. The left-hand side of the window displays information about the time at which each contract violation was detected, along with any additional SLA engine-specific information provided (such as references to violated obligation terminology etc.). When an individual record is

selected from the left-hand pane, the right-hand pane displays metric data associated with the violation (identifying the measurement that caused the violation notice).

Figure 43 The GUI Message Contents window

The 'Message Contents' tab window (Figure 43) displays contract-specific request details in the left-hand pane. This includes the exact time that a measurement occurred, the name of the operation that was called, the provider identifier, and the identifier of the client that sent the original request. The metric data collected during the associated measurement is shown in the right-hand pane upon selection of a specific message (in the form of a list of metric-name/metric-value pairs).

3.3.6.5 Additional Scalability Measures in the MeCo Framework

A small set of additional mechanisms were prototyped within the Measurement Service to provide greater capacity to process incoming data and transfer information between the various internal sub-components.

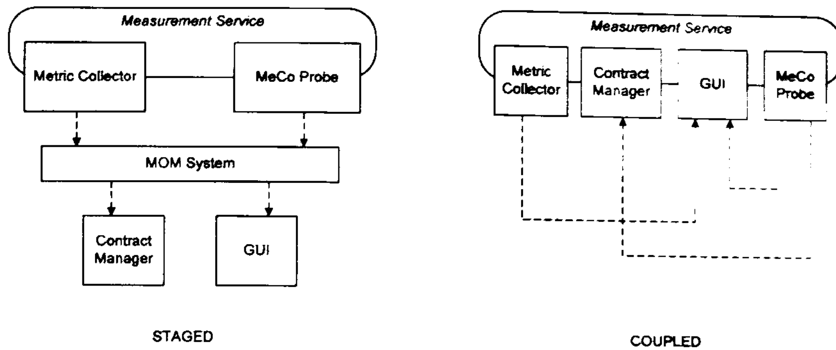


Figure 44 Staged and coupled metric processing

The notion of distinct 'Staged' and 'Coupled' processor-core components was introduced (Figure 44) based on the notion of 'staged components' (as described in [Welsh01]). For example if a 'staged' processing core is used within the Metric Collector, measurement data is delivered immediately to JMS queues to which the Contract Manager and GUI are registered. These sub-components then use the message data to carry out contract evaluations and updates to the end-user interface respectively. In this way the Metric Collector can continue processing messages without waiting for other sub-components to complete actions using measurement data. This differs from the 'coupled' approach wherein the Metric Collector would have to explicitly call the Contract Manager to process metric data, wait for it to complete processing, and then wait for the GUI to draw data to the relevant charts etc. To allow a component to participate in 'staged' communication it must implement the 'StagedComponent' interface and provide suitable logic for the inherited 'processEvent' method:

```
public void processEvent(LinkedList msgParams, JMSStagedComponentNotifier cLnr, String
methName);
```

Message parameters are passed in using this method, and the method to be called within the staged component is identified. A specialised JMS object is also supplied as an argument to allow the object to communicate with other staged components. This allows components to be addressed through JMS queues by their Java class-name. The option is given to enable or disable 'staged' processing in the Measurement Service configuration file (Section 3.3.6.3).

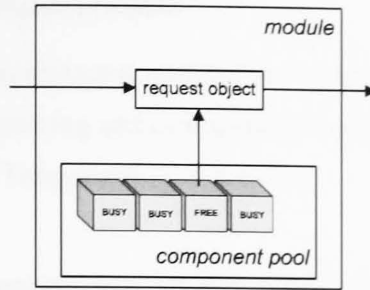


Figure 45 Generic object pooling

'Object pooling' (discussed in [Little99]) was another scalability mechanism visited in the MeCo Framework (Figure 45). With this approach a sub-component becomes in essence a cluster of replicated objects which can each be assigned to process an incoming piece of data. Object pooling was implemented in a limited capacity in the logging 'helper' sub-component, based upon a generic pooling strategy.

These extensions were engineered on a small scale for evaluation purposes, but were not deployed across the system. It was determined that they added undue complexity to the deployment and maintenance of the MeCo Framework (i.e. modifications to the MOM subsystem that hindered heterogeneity and a reliance on inherent knowledge of optimisation techniques on the part of those using the framework).

3.4 Satisfaction of Requirements

It is worth revisiting the requirements outlined in Section 2.6.4 to appraise the MeCo Framework as an SLA monitoring and evaluation infrastructure. A brief outline of the achievements of the MeCo Framework is as follows:

- Any contract engine can be integrated into the Measurement Service provided an interface to Java code can be created (see Section 3.4.1).

- Measurement capabilities can be supported on-demand across the system (see Section 3.4.1).

- The MeCo Probe can support any application technology supported by the WSIF Framework, but only simulates single-part request behaviour (see Section 3.4.2 & 3.4.3).

- The Provider-side MeCo can be integrated with any Java-based middleware that supports transparent message interception in the request/response stack (see Section 3.4.2 & 3.4.3).

- Any Java-based MOM technology can be used to implement the Messaging Service interface provided it accommodates a publish/subscribe notification scheme (see Section 3.4.3).

- Multiple service instances can be monitored from the same Measurement Service instance. However, multiple clients can only be monitored if they are interacting with the same service provider and are connected to the same network (see Section 3.4.4).

- Deployment of MeCo components at the server-side requires only minimal modification to server configuration files (and not to application code). Client-side logic does not require modification (see Section 3.4.5).

- The MeCo Framework is modular – monitoring logic can be re-used across different services and new monitoring logic can be created where necessary (see Section 3.4.6).

3.4.1 Contractual Heterogeneity

The MeCo Framework provides a generic SLA engine that allows framework users to integrate any contract specification language, providing that a contract interface can be provided to join the monitoring components and the contract evaluation engine within the Measurement Service.

The modular component model accommodates additional measurement capabilities on-demand if a contract specification requires it. This provides adaptability to per-service monitoring requirements (as discussed in Section 3.2.3). The MeCo Probe also adapts to monitoring requirements, so provider obligations can be observed within differing service environments (although only single-message requests are modelled). Additional resources are not required to process probe calls, but there is potential for service providers to deliberately provide acceptable service to MeCo Probe calls if the probing interval is known. For example, a provider may ensure that the service is available and working correctly when a probe call is imminent and then allow the service availability to suffer at other times when service clients may require it. This shortcoming could be resolved in future with (for instance) randomised probing intervals, to monitor service provision without the provider having the opportunity to prepare for being probed in advance. This would be akin to a 'random inspection' of the service.

Sections 2.6.4.1 & 3.2.4 discuss formalised standardisation of parameter definitions within SLA obligations. In the MeCo Framework metrics are referenced by 'global' identifiers (i.e. not just the class-name used to obtain the associated measurements). For example, measurements of the response time of request processing within the Enterprise JavaBeans container in the JBoss server application are referenced by the 'ejbResponseTime' identifier. This is a composition of an informal 'ejb' prefix (indicating the monitored application technology) and a 'ResponseTime' measurement descriptor. When 'complex' and 'probe' measurements are read into the system (from the Measurement Service configuration file and individual probe descriptors respectively) they are encapsulated in a combination of 'metric name',

'measurement title' and 'measurement unit'. This combination of attributes allows automatic location of measurement classes. The aforementioned 'metric names' enable indirect reference to measurement types and units through a set of (albeit informal) metric identifiers. Metric identifiers can be referenced in SLA engine interfaces, but are not formally standardised – discussion of standardisation is included in the Future Work described in Section 6.3.

3.4.2 Domain Heterogeneity

The MeCo Probe can simulate any application-level entity based within Web service technologies through the WSIF framework and its extensibility features. This includes SOAP, Enterprise JavaBeans or even JMS.

The Provider-side MeCo can monitor any application technology that provides interceptor mechanisms for transparent message inspection (as is applicable with most major middleware platforms). This adaptability creates potential for the concepts developed within the MeCo Framework to be applied to other service domains. To prove the latter, applied examples must be presented. For details of an application that re-appropriates the logical structure of the MeCo Framework in the domain of Distributed Virtual Environments (DVEs) refer to Chapter 4. Integration of the entire MeCo Framework into a Distributed Virtual Environment is discussed in Section 4.4.

3.4.3 Accommodation of Enabling Technologies

Interfaces to application technologies can be developed for server-side metric collection and active measurement. Interfaces have been created to integrate monitoring features with SOAP and EJB services. The only stipulation is that an interceptor mechanism exists to allow transparent monitoring of high-level applications. There is no dependence on application-specific data-gathering techniques as data is inferred by the Measurement Service and fed directly into the internal contract engine.

Heterogeneous metric collection is used extensively in the MeCo Framework, including probe measurements, the contract engine interface and the MOM-based communications subsystem (which can employ any MOM product with publish/subscribe capabilities).

3.4.4 Scalability towards Participant Entities and Service Contracts

Multiple contracts can be simultaneously monitored from the same Measurement Service instance. In combination with the server-side Provider Environment this allows for multiple service clients to be monitored. The MeCo Probe sub-component can be shared between similar services (albeit where service clients are connected to the same network), potentially reducing the network traffic that it generates.

The processes of transmitting and receiving metric updates are essentially as scalable as the MOM subsystem that connects the Provider-side MeCo and the Measurement Service. The scalability of the Provider-side MeCo is further governed by the session management facilities within the server platform (which determine how Interceptors are deployed).

Relating to the discussion of service environment permutations in Section 3.2.2, there is mixed success. Consolidation of MeCo Probes across services with identical internal logic is possible. Also the Provider-side MeCo is capable of gathering measurements on behalf of numerous services at once, but only with the assumption that the services all use the same application technology and have similar internal logic, and that metric-dissemination is structured accordingly.

Monitoring of multiple contracts from a single Measurement Service instance (specifically when contracts refer to more than one service provider) can only be achieved if other Measurement Service instances have pre-configured referenced provider platforms other than the one that each Measurement Service is in direct contact with. This is because a Measurement Service instance can only subscribe to metric update channels if they have already been configured.

Another point is that only one service provider can be configured from each Measurement Service instance, and this is discussed in the Future Work (Section 6.3). Furthermore, the Measurement Service does not contain functionality for replicating internal logic to facilitate measurement updates from an arbitrary number of Provider-side MeCos.

With reference to support of multiple Measurement Service instances, trusted third party monitoring from numerous sites is achievable (through subscription to metric topics), with potential for rudimentary load-balancing. In this case it is assumed that per-service metrics such as service utilisation are not observed, as this would require consolidation of evaluation data from numerous Measurement Service instances.

3.4.5 Transparent Deployment and Operation

Deployment of MeCo objects to application server platforms requires only minor modifications to server configuration files, without the need to modify server code. Also the MeCo Framework does not interrupt the behaviour of service clients (since no monitoring logic is deployed at the client-side).

MeCo Interceptors are specialised towards message interception and processing. Any potentially time-consuming post-processing of data occurs at the Measurement Service. This avoids any need for additional processing resources within the Provider-side MeCo. The communication channels connecting the Provider-side MeCo and the Measurement Service can be deployed across a stretch of network that does not interfere with the observed service participants.

Transparent fault behaviour within end-system devices is discussed in Section 2.6.4.5. This applies mainly to the MeCo Interceptor implementations. The EJB- and SOAP-based MeCo Interceptors propagate request data through the JBoss Interceptor and Axis stacks respectively regardless of any internal failures. The Interceptors fail only in the case of failure of the supporting server (which would be detected by the MeCo Probe). Within the Measurement Service database logging can be employed to maintain records of service behaviour in case of potential failure.

If required the Complex Metric Calculator can perform per-service measurements so as to shift processing complexity away from the server platform.

3.4.6 Ease of Deployment and Modularity

Deployment of the MeCo Framework is achieved through positioning of MeCo objects within the service platform, configuration of the MOM subsystem, and activation of the Measurement Service.

The MeCo Framework exists as a set of distributed monitoring components. Monitoring capabilities are modularised within specialised per-metric measurement classes, and technology-specific metric-sending, notification and interception classes are governed by generic interfaces. This affords adaptability to communication protocols and measurement capabilities while also providing the capacity to reuse existing monitoring logic where applicable.

Because measurement classes are retrieved from class libraries, it is possible for application developers to create measurement classes for deployment in both the Provider-side MeCo and the Measurement Service. However, development of SLA language interfaces assumes knowledge of the workings of the language or associated contract engine.

3.5 Summary

- A monitoring architecture was previously proposed [Jimenez04] to resolve a number of the SLA monitoring and evaluation issues raised in the related work. This formed the foundation for the MeCo Framework [MorganIfip05].

- The MeCo Framework aims to provide a heterogeneous, distributed SLA monitoring and evaluation infrastructure. It is composed of a number of core components. These components are distributed and modular, and can be adapted to per-service monitoring requirements. The MeCo Framework consists of:

    - A Provider-side MeCo positioned to observe service client behaviour.

    - A MeCo Probe capable of active measurements of service provider behaviour.

    - A Measurement Service component that collects and processes measurement data.

    - A Contract Manager that can integrate an SLA language and processing engine.

    - A Messaging Service that utilises Message-Oriented Middleware (MOM) to facilitate data transmission between the Provider-side MeCo and the Measurement Service.

- The Provider-side MeCo has a number of distinct subsystems:

    - Middleware-specific 'interceptors' allow transparent integration of monitoring logic into E-Commerce middleware stacks. Implementations of MeCo Interceptors exist for the SOAP Web service protocol and Enterprise JavaBeans (EJBs).

    - The Provider Environment manages the various sub-components during configuration and operation.

    - The Metric Classloader can load classes into the Provider Environment to provide service-specific measurement capabilities.

    - The Metric Notifier communicates measurement data to the Measurement Service via the Messaging Service subsystem.

- The Messaging Service is the communication backbone of the MeCo Framework. It has sub-components of its own, namely the Message Aggregator (which can group messages produced by the Provider-side MeCo) and the security and logging helpers that perform message authentication and logging respectively. The Messaging Service can use any publish/subscribe-capable MOM technology, with the only requirement being that there is some concept of event notification topics available.

- The Measurement Service correlates measurement data from the Provider-side MeCo and its own internal MeCo Probe. The sub-components of the Measurement Service can be tailored to meet the needs of individual service environments:

    - The Contract Manager is an SLA language-agnostic metric evaluation component.

- The Metric Collector collects measurement data from the MOM subsystem and extracts metric data for use in contract evaluation.

- The Violation Notifier transmits notifications of SLA violations to interested parties when they occur.

- The GUI component presents measurement data to framework administrators.

- The MeCo Probe adapts active measurements for transmission across any E-Commerce application protocol (through use of the WSIF framework and specialised WSDL service descriptions).

4. Distributed Virtual Environments

Distributed Virtual Environments (DVEs) allow geographically remote computer users to interact in a shared virtual world. Users interact directly with each other and the environment itself. Actions are conveyed to other users as they happen so as to ensure a consistent view of the world.

In this chapter there will be a discussion of how the concepts and techniques developed within the MeCo monitoring & evaluation framework can be applied to resolve QoS monitoring problems in the domain of Distributed Virtual Environments.

4.1 Re-Appropriating the MeCo Framework

Before considering QoS-related problems within Distributed Virtual Environments, it is first worth summarising the features of the MeCo Framework and how they may be re-appropriated to provide consistency guarantees for scalable virtual environments.

4.1.1 Achievements in SLA Monitoring

The MeCo Framework exhibits a number of useful features for deployment across a variety of E-Commerce service environments. Interoperability between differing network entities in heterogeneous environments is possible. The framework also respects the need to adapt to dynamically-changing network characteristics, in terms of service participants, but also the evolution of deployed services.

The MeCo Framework provides modular, distributed monitoring capabilities and performance evaluation of individual service participants. These features can be tailored to meet the SLA monitoring requirements of individual service environments.

The applicability of the MeCo Framework to divergent services can only be assured if it can be shown to provide benefits to services within different domains. It should be feasible to apply the conventions that were refined in developing the MeCo Framework to a different kind of distributed service. It was deemed appropriate to develop a distinct application of centralised monitoring and evaluation of per-participant behaviour and performance metrics for DVEs.

4.1.2 A Different Domain – Distributed Virtual Environments

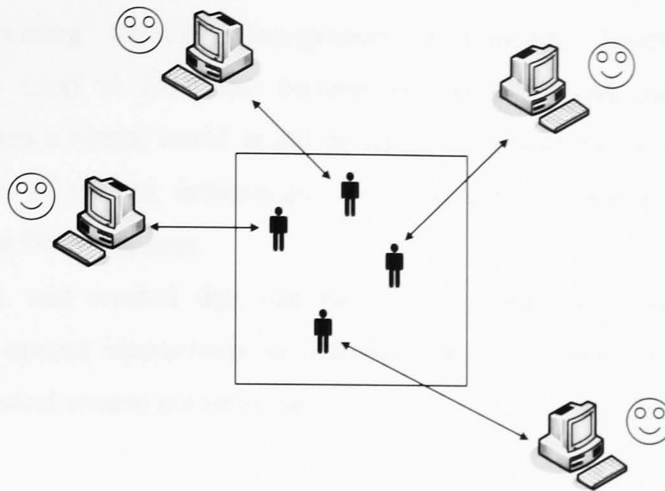


Figure 46 Users interacting within a Distributed Virtual Environment (DVE)

A Distributed Virtual Environment (DVE) is a simulation of a virtual world wherein geographically dispersed users interact with each other and the world according to the rules of the environment. Applied examples include Massively-Multiplayer Online Games (MMOGs) and military simulations. DVEs are typically enriched with multimedia content so end-users can both see and hear the elements they interact with. DVE applications are growing in popularity. It becomes increasingly important to ensure real-time consistency within the virtual world, while also providing guarantees of service quality to an arbitrary number of users which may grow into the hundreds or perhaps thousands.

The need for consistent service quality between a DVE provider and the associated end-users acts as a base of comparison to the provision of E-Commerce services. Both service classes require the service provider to ensure a determinable level of quality, while stipulating that end-users do not intentionally abuse the service. These similarities provide a foundation for monitoring and evaluation constructs within DVEs, with scope for applications specialised towards solving problems specific to DVEs.

4.2 Implementation

In Section 2.4 the principles of Distributed Virtual Environments (DVEs) were discussed, including Interest Management mechanisms. Interest Management techniques are used to limit the number of messages sent between machines interacting within a virtual world to aid in achieving scalability. With this there was an examination of missed interactions and how their occurrence can affect the experiences of a DVE end-user.

An application was created that can be used to configure a DVE so that the occurrence of missed interactions is reduced, and the resource constraints of the underlying physical system are respected.

4.2.1 Implementation Assumptions

A number of assumptions have been made about the DVEs that the application would be to configure:

- The virtual environment is three-dimensional and cube-shaped.
- The virtual environment uses a peer-to-peer (P2P) management approach. This restriction is applied so as to focus the implementation on a specific DVE deployment, while providing a suitable deployment challenge.
- Aura-based Interest Management techniques are used to reduce message production.
- The Interest Management configuration applies to object aura-sizes and inter-object heartbeat message-sending intervals.
- A global view of per-entity messaging and configuration events is achievable.
- All entities within the same object class send heartbeat messages to other entities at the same interval and at the same time. This is also assumed to be the case with high-frequency messages.

- The same bounds of ... object class. Objects move at the same ... falls within the aforementioned ...

4.2.2 The DVE Simulator

If missed interactions occur within a DVE ... virtual world behaviour. As an example ... another entity if they believe an inter... interactions between virtual world ... correctly, an interaction may appear to be ... react to user actions immediately or at ...

The occurrence of missed interactions should ... Object activity information is conveyed ... there is a need to also respect the capabilities ... message production appropriately. Failure to ... flooding the network with messages, degrading ... with it the quality of the DVE experience in ...

A number of steps can be taken to reduce the ... a DVE. These include changing the sizes ... altering the frequency of heartbeat message ... velocities of objects. Normally, it would ... application developer to determine suitable ... own experiences. Such an ad-hoc approach ...

It would be desirable to be able to determine ... heartbeat rate, and object velocity thro... DVE. This can be achieved with a DVE sim... developer could essentially test the Interest Man... prior to deployment. There is then potential ... application and associated DVE-oriented ...

When developing the DVE Simulator ... Management was considered in the first instance ... aura-based Interest Management deployed in ...

auras provide a more accurate model of inter-object influence than the region-based approach. Furthermore, this permutation provides a more challenging test of the DVE Simulator, as the occurrence of missed interactions is dependent upon the exchange of heartbeat messages between participating entities (which in itself would need to be regulated). Alteration of object velocities as an Interest Management mechanism is not considered within the DVE Simulator, as this directly impacts upon the laws of the virtual world itself.

There is a need to be able to assess the merits of a particular Interest Management configuration. The DVE Simulator should present criteria by which to determine the Quality-of-Service (QoS) of an individual DVE. The derivation of performance measures that represent the quality of a DVE would serve as a foundation for automatic evaluation mechanisms and measurement-based assessment.

4.2.3 Object Classes

In a DVE simulation it is desirable to effectively emulate the behaviour of entities in a virtual world. It becomes necessary to simulate object movement in a three-dimensional space (e.g. end-users moving their representative characters around the virtual world). It is assumed that the movement of an object is dependent upon its 'class' (e.g. airplane, person, car) and that instances of more than one object class can co-exist in the virtual world at any time. For instance a plane will behave in a manner that tends to follow a more deterministic flight path at high velocities, whereas a person may exhibit more non-deterministic movement at lower velocities. These two object classes should be able to interact in an appropriate and meaningful way.

The introduction of object classes (and associated styles of movement) is a necessity for realistically modelling DVEs. Achievable velocities between different types of objects may vary to such an extent that one object is capable of passing another before influence between the two can be detected and message passing enacted, depending upon the Interest Management configuration. Such events must be modelled appropriately if they are to be controlled.

The style of object movement is also a factor in determining the occurrence of missed interactions. Inter-object influences involving objects that have a tendency to stop moving for periods of time may be identified more readily in real-time than for those

objects that are constantly on the move. This would all depend however upon the regularity of inter-object positional update messages etc.

Four classes of virtual object (and with this, four distinct styles of movement) were derived for use in the simulation application (as shown in Figure 47):

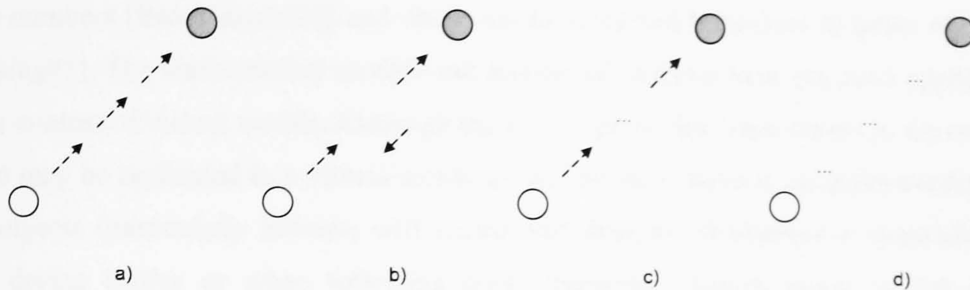


Figure 47 Different forms of simulated object movement

a) *Direct:* objects of this type move – without stopping – along a linear path at a fixed velocity.

b) *Indirect:* objects move along a linear path at a fixed speed but may deviate from this direct path periodically for short periods of time.

c) *Stuttering:* objects move along a linear path at a fixed speed but may pause periodically for short periods of time.

d) *Static:* objects of this type do not move at all (therefore behaving much like stationary, or idle, objects).

The proposed set of object classes and their identifying characteristics provide a range of styles of movement that in combination exhibit an adequate (albeit basic) variety of object behaviours suitable for simulating a DVE. By combining instances of these object classes there is potential for relatively complex interaction scenarios.

4.2.4 Simulating Object Behaviour

In trying to determine how end-users may behave within a DVE, literature regarding human behaviour was consulted. Much of this examined 'crowding' of people within the real world. Examples include studies of crowds entering and leaving stadiums in large numbers [Brocklehurst05] and observations of crowd behaviour in urban areas [Helbing05]. The mathematical models and associated analysis have not been applied in the context of virtual worlds. Although there is no proof that what occurs in the real world may be replicated in a virtual world, in the literature there is an understanding that objects (particularly avatars) will crowd and disperse throughout a simulation (e.g. during battles or when following quest markers). Objects rarely remain in isolation in a virtual world as there is a drive to interact [Singhal99]. As such, the act of crowding amongst simulated users must be emulated.

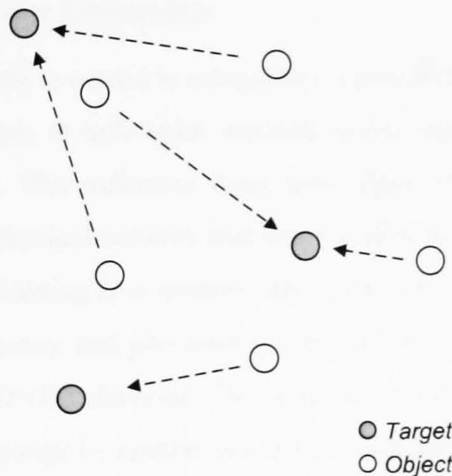


Figure 48 Influencing object movements by assigning targets

An attempt was made to provide realistic crowding of objects within the virtual world, through the positioning of 'targets' (as shown in Figure 48). This would require a number of target objects to be positioned within the virtual world, with objects made to travel towards them. This creates scope for crowding behaviour amongst sets of objects as they travel towards the same target. It would also be conceivable for targets to relocate during the simulation, and for objects to change the target towards which they are travelling. With these characteristics, if the number of targets is strictly less

than the number of objects, objects will effectively cluster and disperse throughout the simulation.

Another consideration is whether a simulated virtual world should include obstacles that restrict the movements of objects as they move around the environment. These include terrain such as mountains or impassable areas such as lakes, and impenetrable objects such as walls. In this case the decision was made to allow objects to roam freely around the virtual world without hindrance. This was rationalised by the need for the preliminary results derived from DVE simulations to be free from any influences contributable to environmental constraints (such as space limitations introduced by the existence of particular obstacles). It would be easier to attribute missed interactions to Interest Management constraints without needing to consider the undeterminable influence of virtual world obstacles.

4.2.5 Modeling Resource Constraints

In simulating a DVE there is a need to adequately represent both the delays associated with processing messages at individual network nodes and the underlying network transmission overheads. The influence upon inter-object message transmission and processing of both the physical network and the available processing resources should be considered when examining how missed interactions occur.

Introducing network latency and processing overhead into a simulation undoubtedly complicates simulated DVE behaviour. The sending of a message by one object and the receiving of that message by another are not guaranteed to occur at the same point in global time. If the receiver of a heartbeat message is Obj_1 and Obj_2 is the sender, Obj_1 will need to determine whether its current aura overlaps with the aura of Obj_2. It may be that when doing this Obj_1 is using positional data that refers to Obj_2's aura as of the time that Obj_2 sent the message. This is not necessarily the same time as when it was received. Obj_1's judgement is then made using potentially out-of-date positional data for Obj_2's aura.

To adequately model the delays associated with network latency and processing overhead two variables D_{lat} and D_{prc} are introduced that respectively describe these two time periods. Together D_{lat} and D_{prc} represent the processing time required at each network node to resolve peer-to-peer Interest Management issues. As an

example, for a DVE with few objects and limited networking resources D_{lat} would be high and D_{prc} would be low.

It is undesirable for a simulation to be influenced by actual processing delays within the execution environment of the DVE Simulator itself (e.g. CPU speed, memory availability, pre-emptive operations in the operating system). As such, the passage of time within each simulation is represented in algorithm iterations. A single iteration then constitutes one unit of time, with each object moving once during each unit.

4.2.6 The DVE Simulator Interface

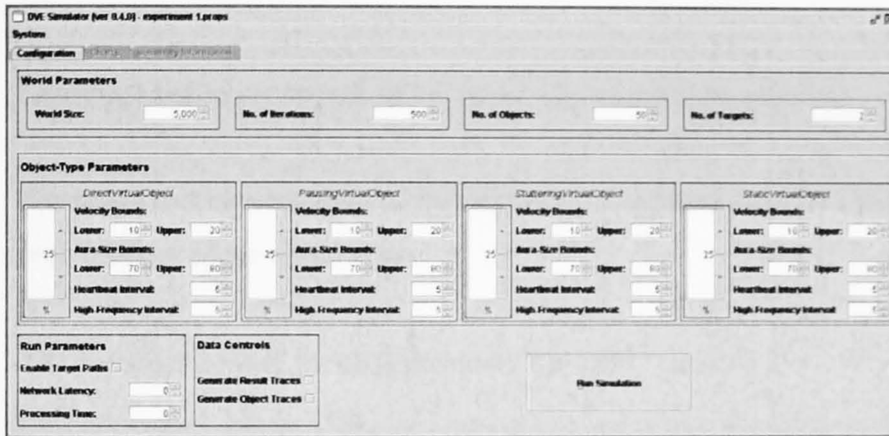


Figure 49 The DVE Simulator interface

The DVE Simulator (Figure 49) incorporates a Java-based Graphical User Interface (GUI). This lets users configure and initiate a DVE simulation, and view the results.

The DVE Simulator interface has a list of menu options to allow DVE configurations to be saved to an external file, loaded from a file, reset or re-run, as well as options to allow numerical results and chart data to be saved to file. There is also an option to record the entire history of each DVE simulation as an XML-based log for external analysis (i.e. data regarding how individual objects behaved during each iteration).

For each simulation, a number of global (or 'world') parameters can be configured:

- The size of the world (the length of the equidistant axes in the cube-shaped virtual environment).

- The duration of the simulation (measured in iterations).
- The number of objects inhabiting the virtual world.
- The number of targets to position in the virtual world.

A number of parameters may also be configured for each object-class:

- The quotient of the total number of objects in the simulation that are represented as instances of the object class (as a percentage).
- The upper and lower bounds of the achievable object class velocity.
- The upper and lower bounds of the radius of the spherical area-of-influence of each instance of the object class.
- The heartbeat interval for all instances of the object class.
- The high-frequency messaging interval for all instances of the object class.

The influence of network and processing constraints (Section 4.2.5) can also be configured through the DVE Simulator interface.

4.2.7 DVE Configuration

Once global, per-object class and network simulation parameters have been configured the DVE Simulator constructs a representation of a virtual world based upon the configuration parameters. The required number of target objects and instances of each object class are created and randomly positioned within the boundaries of the virtual environment. The random dispersal of targets and objects reflects both how goals or meeting places within a virtual environment are

geographically distant, but also how DVE users do not necessarily enter the virtual world at the same location.

To model crowding behaviour (as described in Section 4.2.3) individual objects are assigned targets to move towards during the course of the simulation. This behaviour can be modelled in two ways:

- *Static:* each object is assigned one target to follow throughout the entire simulation. Such scenarios may arise in DVEs if there are a number of integral meeting places within the virtual world that users are encouraged to reach and remain within the vicinity of (e.g. battlegrounds, towns).

- *Dynamic:* each virtual object can be assigned a target to move towards, and upon reaching this target made to pursue another one (assuming there is more than one). The order in which targets are visited is rearranged every time the entire list of targets has been visited. Such behaviour can be seen in DVEs where users are assigned a series of tasks to complete in succession at different locations within the virtual world.

The collision detection algorithm is also configured with a duration value (the number of iterations that must be completed within the simulation) and values for any processing delays that are being modelled.

4.2.8 Interaction Detection

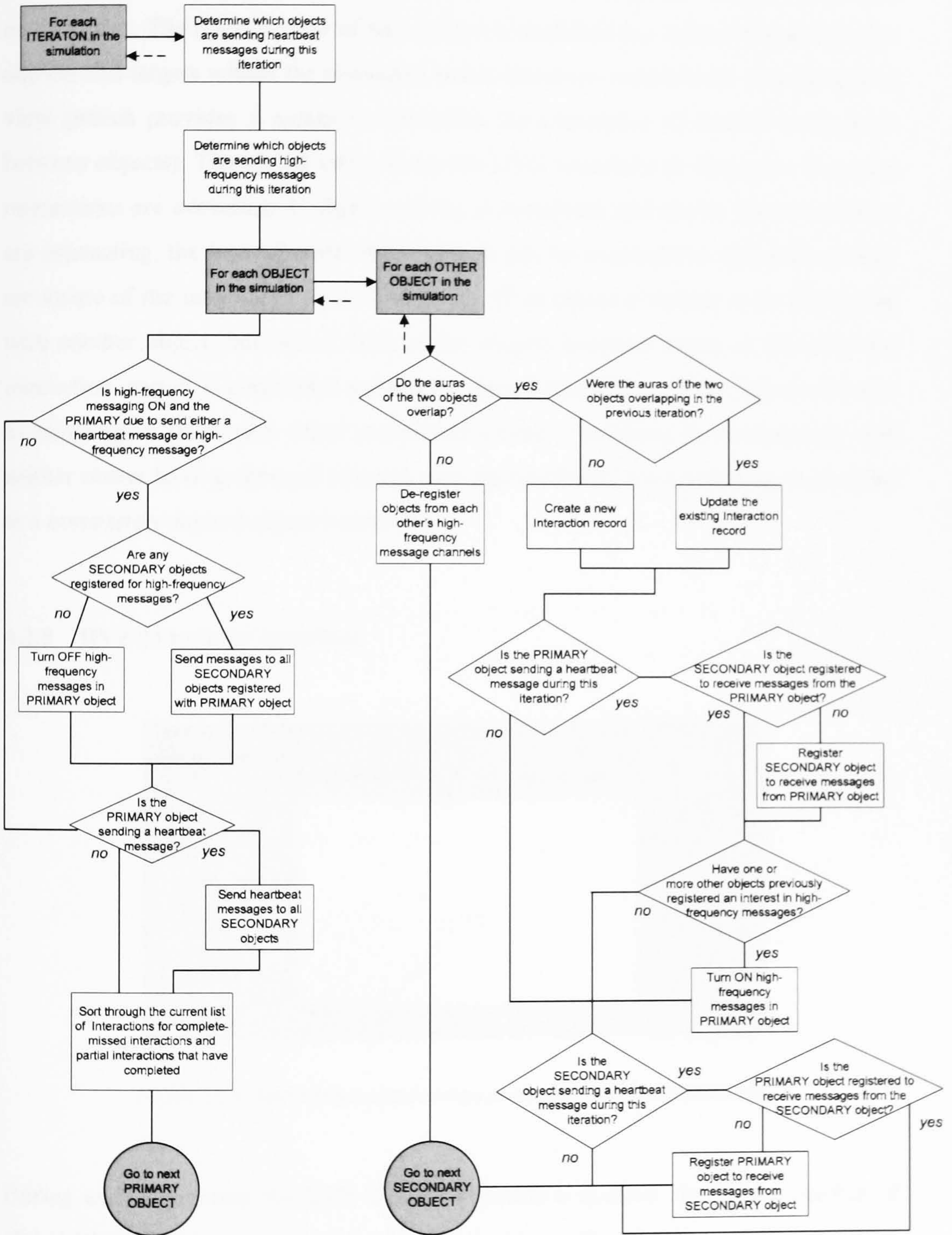


Figure 50 Interaction detection logic

Figure 50 describes the logic for determining object interactions etc. The DVE Simulator models interactions that occur between objects in the simulated environment. The internal view of each object is modelled (i.e. what it knows of other objects and targets within the simulated environment) in conjunction with the global view (which provides a means to determine the occurrence of missed interactions between objects). The global view allows the DVE Simulator to determine if missed interactions are occurring. If object activity is correlated and shows that two objects are interacting, the internal state of the objects can be examined to determine if they are aware of the interaction as it is occurring. If an object is known to be interacting with another object, but one or both of the objects becomes aware of this after the interaction started, it constitutes a partially-missed interaction. If the objects are seen to move away from each other completely (thereby finishing the interaction) with neither object having changed internal state appropriately, the interaction is recorded as a completely-missed object interaction.

4.2.9 DVE Simulator Interface

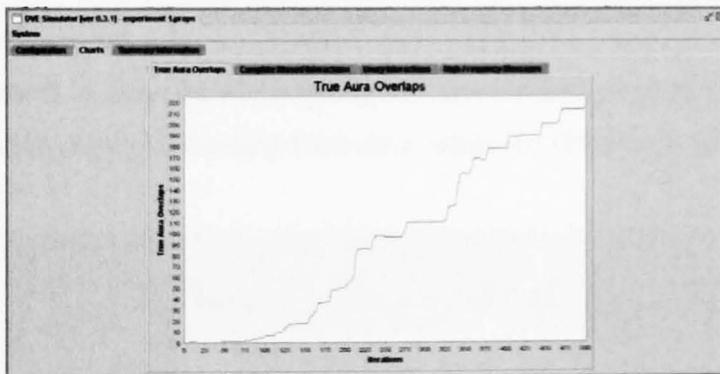


Figure 51 A chart of true aura overlaps produced by the DVE Simulator

During each simulation the DVE Simulator interface updates charts of a number of global DVE characteristics (as shown in Figure 51 and Figure 52):

- *True Interactions:* a measure of the number of interactions that actually occurred during a specific iteration (be they complete or ongoing).

- *Complete Missed Interactions:* how many interactions that occurred between pairs of objects where neither object was aware of the interaction having occurred (due to an inadequate number of messages having been sent between them).

- *Unary Interactions:* the number of interactions occurring at a particular time in the simulation between pairs of objects wherein only one of the objects is aware of the interaction having occurred. Complete missed interactions and unary interactions are determined from the set of interactions that completed during the associated iteration.

- *Number of Messages Sent:* includes both heartbeat and high-frequency messages as sent by all objects during each iteration, acting as a record of the complete number of messages sent within the DVE simulation.

These results allow a DVE application developer to characterise the interactions that occur between objects with a given DVE configuration (as described in Section 2.4.4). They also inform developers of the volume of messages produced with a particular configuration.
Graphs are drawn in real-time for inspection as a simulation progresses, with time (a series of iteration numbers) forming the x-axis values for all of the charts.

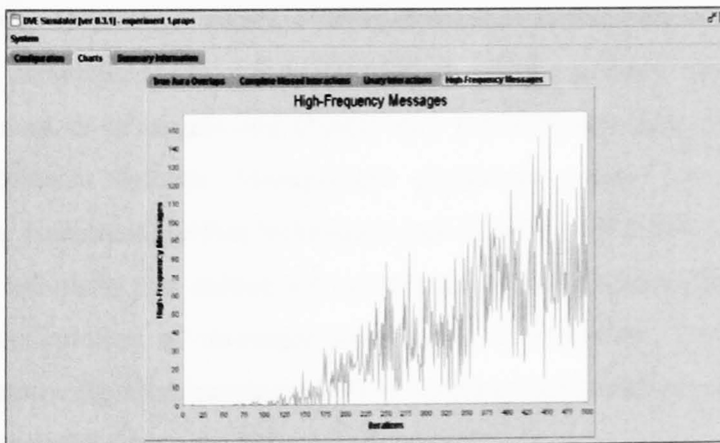


Figure 52 A chart of high-frequency message exchange produced by the DVE Simulator

In addition to charts, an accompanying series of summary measurements are produced once a simulation has reached completion. These include a breakdown of the combined total of heartbeat and high-frequency messages sent by all objects during the simulation, how many complete-missed and unary interactions occurred in total, and how many interactions went unobserved by individual objects (either completely or partially). This data can be saved to a text-based file for further inspection and analysis in other applications (e.g. a spreadsheet package such as Microsoft Excel).

4.3 DVE Simulator – Evolutionary Component

The DVE Simulator allows a DVE application developer to examine Interest Management configurations in terms of potential resource usage and interaction consistency within a virtual world. There is scope to extend this functionality, as the DVE Simulator relies upon the intuition of the application developer in determining the range and combination of parameter values to examine. This is inherently time-consuming when an optimum configuration is not immediately forthcoming.

One solution is to incorporate some form of automated evaluation into the DVE Simulator, so that different parameter configurations can be examined in succession without the need for human intervention. If parameter configurations can be automatically fine-tuned towards an optimum configuration, there would be a reduced reliance upon the intuition of the DVE application developer.

An 'evolutionary optimisation' component was combined with the core DVE simulation logic to create a separate DVE simulation application, the Evolutionary Optimisation Simulator (EOS) [Parkin07]. When given a global DVE configuration (world size, number of targets and objects etc.) the EOS automatically assesses the merits of different Interest Management parameter values for use with the configuration. Genetic algorithm techniques (see Sections 4.3.2 & 4.3.3) are used to pursue those solutions that reduce the occurrence of missed interactions while also minimising the number of messages sent during a simulation. The EOS is then capable of improving upon promising DVE configurations until an optimum set of Interest Management parameter values is discovered.

4.3.1 Evolutionary Optimisation Simulator Overview

The Evolutionary Optimisation Simulator (EOS) is primed for finding optimum solutions through initialisation of a finite set of DVE simulations with the same set of global configuration parameters (e.g. world size, number of objects, object class quotients, and achievable velocity ranges). Distinct values for heartbeat message intervals and aura-size ranges (applied to all object classes) are then assigned to each simulation. The DVE simulations are then identical aside from their Interest Management attributes.

Each unique DVE simulation is run to completion and the associated performance results observed. This includes the number of missed interactions and messages produced over the duration of the simulation. The resulting performance data from each DVE simulation is evaluated in a 'fitness function' [Fogel94] (Section 4.3.2) to determine the effectiveness of the associated Interest Management parameters.

The concept of fitness drives the evolutionary optimisation process. Successive sets (or 'generations') of DVE simulations are created based on the performance results (i.e. 'fitness') of the existing simulation set. Those simulations that exhibit the best Interest Management performance (and thereby have the best fitness) form the foundation for the generation of simulations that follows. The attributes of the 'fitter', more promising simulations are retained and fine-tuned in successive simulations. This then acts to improve the suitability of Interest Management parameter values with each successive set of DVE simulation configurations.

Candidate solutions (i.e. Interest Management configurations) are encoded as 'chromosomes' (i.e. solutions in evolutionary optimisation processes). In this case the chromosomes are value-encoded representations of the heartbeat message interval and aura-size value-pairs associated with each DVE simulation. There is no need to encode other parameters (e.g. world size) into the chromosomes as they are global (i.e. shared by all simulations across all generations within the individual EOS instance).

4.3.2 Evolutionary Optimisation – Fitness Function

To assess each simulation configuration the capability of the associated Interest Management parameters to balance (and ideally reduce) both the occurrence of missed interactions and message production is evaluated using a specialised fitness function. This function and the associated variables are described as follows:

$$F = 1/3(1-C) + 1/3(1-P) + 1/3(A/E)$$

- *F:* overall fitness of the candidate solution.
- *C:* percentage of all interactions that occurred which are missed interactions.
- *P:* percentage of all interactions that occurred which are partially missed interactions.
- *A:* number of messages sent during the simulation.
- *E:* number of messages that it is estimated would have been sent during the simulation if no missed interactions had occurred.

The fitness function attempts to balance the number of missed interactions that occurred with the number of messages exchanged between objects during the simulation. Without seeking a direct balance, it would for instance be feasible to give preference to a simulation that produced no missed interactions at the cost of producing an unacceptably high number of heartbeat messages. A result such as this would undoubtedly have the potential to hinder scalability in practice.

The fitness function is a formula, the use of which can be automated to assess the balance of a defined set of parameter values in one calculation. The closer *F* (the result of the fitness function) gets to a value of 1.00 the better the candidate solution being assessed is at striking the desired balance. A simulation configuration can only achieve a fitness of 1.00 if no missed interactions (complete or partial) occur and the number of messages produced is equal to the predicted number of messages that

would have been exchanged within the system had no missed interactions occurred (a logical expectation).

A DVE application developer can potentially alter the fitness function based upon their own preferences. The fitness function described here balances scalability (number of messages exchanged) against the occurrence of missed interactions (partial and complete). A small alteration to the fitness function could give preference to one of the aforementioned parameters over the other, for instance if there are either plentiful resources or relaxed global consistency requirements. Such alterations to the fitness function would however require direct alterations to the code within the EOS.

4.3.3 Evolutionary Optimisation – Crossover, Mutation and Elitism

When the fitness of each candidate DVE configuration in a solution set has been derived, the 'mating potential' of each chromosome is determined in relation to all other chromosomes in the same generation. For this, the fitness values of the chromosomes are compared in tandem with the standard deviation of all the fitness values in the generation. Once the mating potential for each DVE simulation has been determined, the most promising chromosomes are chosen to 'mate' with other selected chromosomes (i.e. blend their characteristics to form a single hybrid offspring). They may also potentially live on into the next generation themselves, through a copy of the chromosome inserted into the next working set.

Mating is achieved through use of the 'crossover' technique [Fogel94]. Two solutions are chosen based upon their mating potential. Chromosome parameter values (i.e. heartbeat message interval and aura-size parameters) are then randomly selected from one or either of the 'parents' to construct a single composite offspring. The hope is that in mating two good chromosomes a better one will be produced, although this is never guaranteed.

In order to maintain a level of variance in the parameter values being examined throughout the evolutionary process, there is an inherent chance that random mutations of varying but limited magnitude are created in the Interest Management parameters of an offspring chromosome. These mutations may or may not then contribute to the successes of candidate solutions in subsequent generations.

The evolutionary process preserves those candidate solutions that show the most promise i.e. those chromosomes that may not be the ideal solution but could otherwise

be regarded as prospective parents in subsequent generations. All chromosomes within the existing population with a fitness value above the average are allowed to 'live on' into the next generation. This is referred to as 'elitism', wherein the best chromosomes are chosen to outlive the rest of the population.

A small subset of each new generation is created from heavily mutated offspring spawned by chromosomes randomly selected from those deemed to be of below-average quality. This additional step ensures that the observed solution space does not become stale, by essentially giving a second chance to those chromosomes that would not have lived on otherwise. This also affords the EOS the capacity to search for candidate solutions in other parts of the solution space (by mutating some offspring chromosomes away from any solution spaces that are already under investigation). Without additional heavily-mutated chromosomes, it is conceivable that as the solution space narrows with subsequent generations an equally promising set of chromosome configurations goes undetected and ignored.

4.3.4 The Evolutionary Optimisation Algorithm

The EOS collects configuration parameters from a graphical user interface (GUI) component (see Section 4.3.5). Once parameter values have been set the evolutionary optimisation algorithm (which incorporates crossover, elitism and mutation as described in Sections 4.3.2 & 4.3.3) is initiated and proceeds as according to the flow chart in Figure 53.

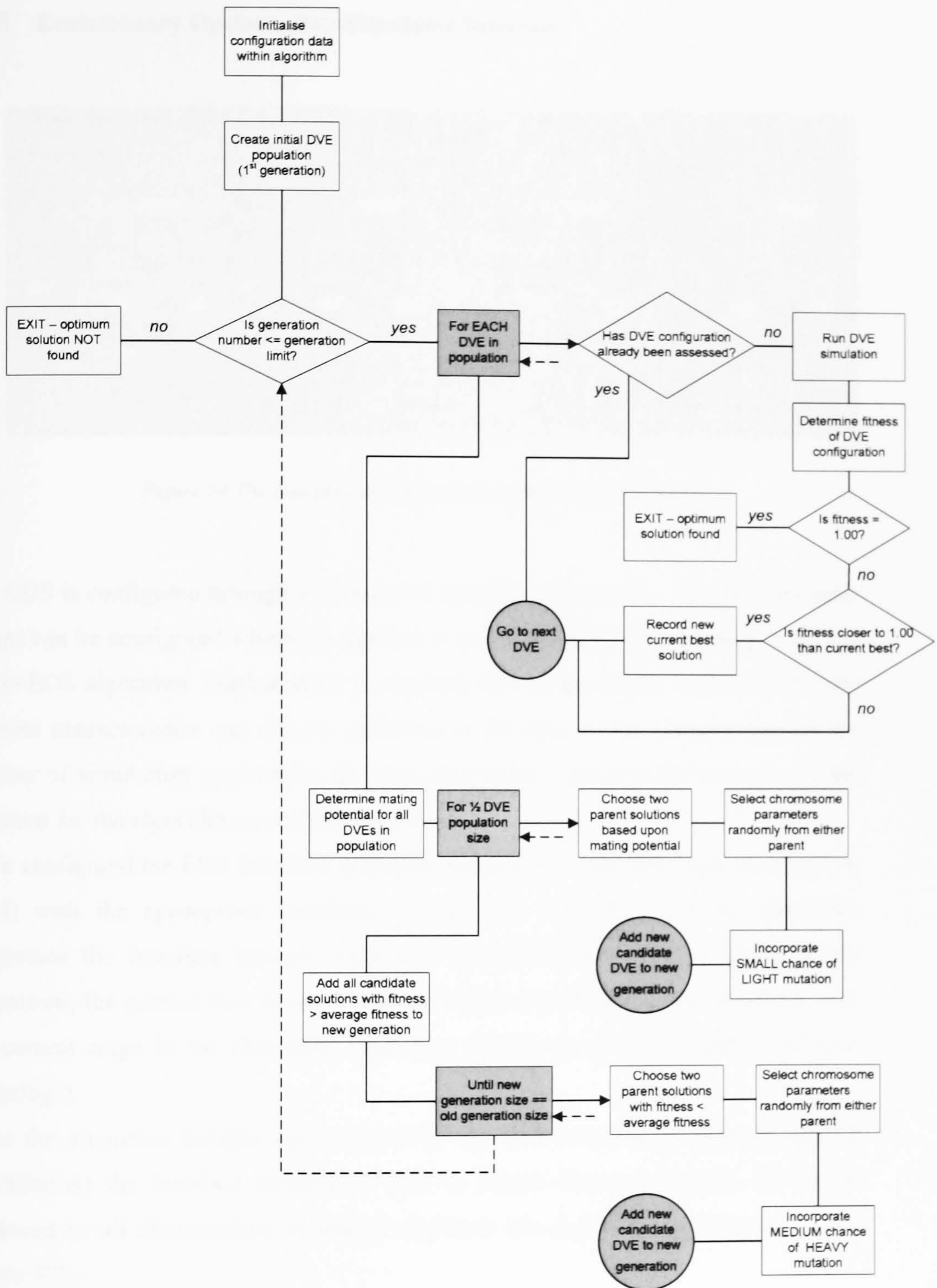


Figure 53 EOS internal algorithm logic

4.3.5 Evolutionary Optimisation Simulator Interface

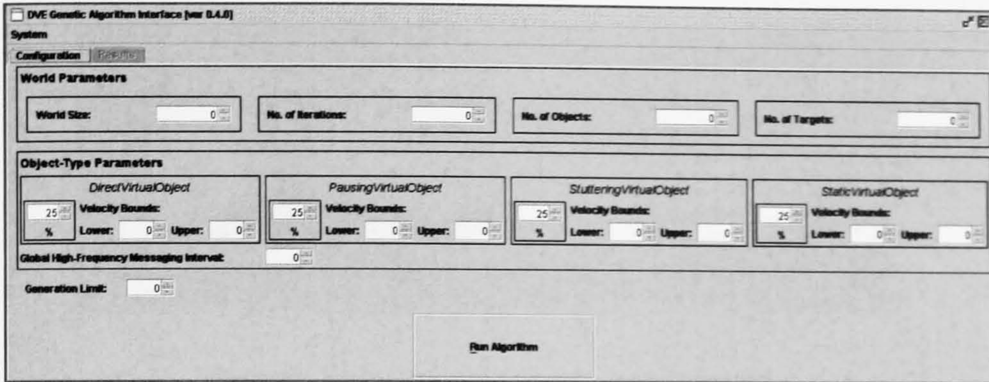


Figure 54 The Evolutionary Optimisation Simulator interface

The EOS is configured through a Java-based interface (Figure 54). 'World' parameter values can be configured which are applied to every candidate simulation in each run of the EOS algorithm. Further static parameters such as per-object class velocity and quotient characteristics can also be calibrated in this way. Users can also specify the number of simulation generations the algorithm should generate (so as to avoid the potential for the algorithm to run indefinitely).

Once configured the EOS interface initialises the genetic algorithm logic (see Section 4.3.4) with the appropriate parameter values and initiates it. As the algorithm progresses the interface actively updates a number of results: the current active generation; the current best fitness value of all the simulations so far analysed, and; the current stage in the algorithm logic (e.g. "analysing DVEs", "generating new offspring").

Once the algorithm finishes (or is halted by the user prematurely through explicit cancellation) the interface produces a pair of linked charts describing the results produced by all chromosome variations visited in the algorithm run (Figure 55 and Figure 57).

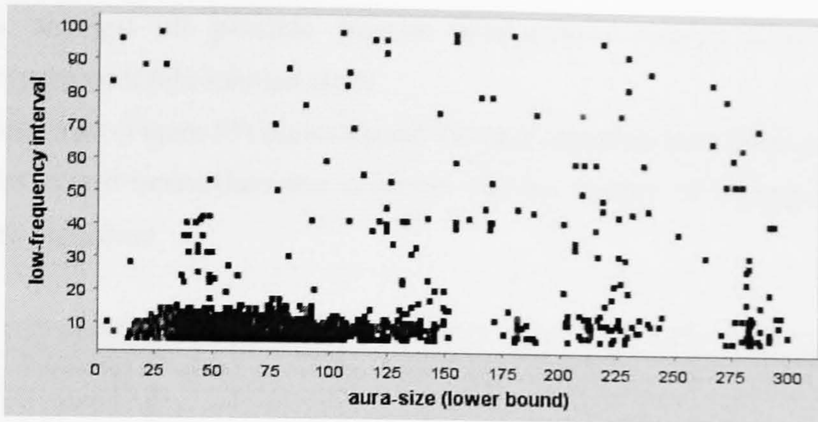


Figure 55 Chart of aura-size against heartbeat interval

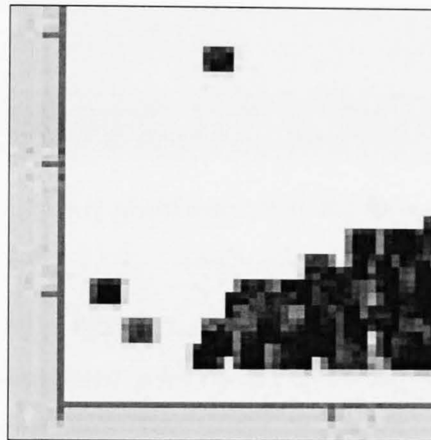


Figure 56 Enlarged segment of aura-size against heartbeat interval chart

The first chart (Figure 55) draws a point to represent each candidate solution analysed by the algorithm. Each point is positioned by the heartbeat interval and aura size values used during the associated simulation. The quality of the candidate solution with respect to its 'fitness' is also illustrated: the darker the point, the better the associated chromosome values were for both reducing the occurrence of missed interactions and minimising message exchanges. This can be seen more clearly in Figure 56, where different points of varying grade have built up around and over each other. The darkness (i.e. quality) of each point is determined with respect to all other DVE configurations visited in the algorithm run, and as such may not represent an 'ideal solution'. There is an assumption that users wish to find the most promising

solutions amongst all possible Interest Management configurations, and not necessarily the optimum solution alone.

The second chart (Figure 57) draws a point for each candidate simulation to relate the number of missed interactions that occurred with the number of messages produced during the simulation:

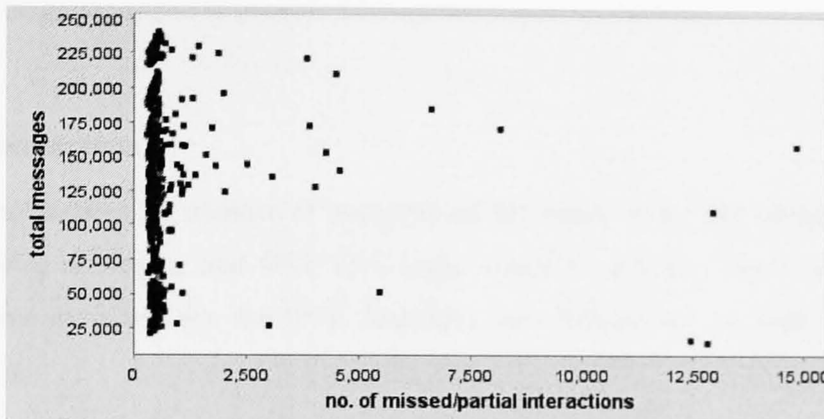


Figure 57 Chart of missed interactions against messages

Both of the charts are linked – when a user holds the mouse pointer over a specific point in either chart, the associated point in the adjoining chart is also highlighted (with both points then coloured red). Text is also displayed next to the mouse pointer to describe the fitness and configuration values for the associated simulation. This allows a DVE developer to pursue their own criteria, such as chromosome fitness, levels of message production or the avoidance of missed interactions. For example, if a DVE application is being run across a high-performance network wherein the number of messages produced is less critical to system performance than the occurrence of missed interactions, these values can be examined accordingly.

4.4 Application of DVE QoS Measures

The DVE Simulator and Evolutionary Optimisation Simulator (EOS) create scope to find virtual environment configurations that reduce message production while retaining consistency within the end-user experience of all users. It is worth examining how the DVE evaluation logic developed for use in these applications can

be deployed in a QoS monitoring infrastructure similar to the MeCo Framework described in Chapter 3.

To combine the constructs of the MeCo Framework with the QoS metric definitions and monitoring logic derived within the DVE Simulator, there must first be an examination of the deployment issues involved. Here there is discussion of how MeCo components can be applied to a DVE infrastructure, as well as how DVE QoS logic can be incorporated to provide DVE performance monitoring.

4.4.1 Assumptions

In the first instance, a number of assumptions are made about the environment to which MeCo constructs and DVE QoS logic would be applied. These include the assumptions described for the DVE Simulator (see Section 4.2.1), with additional restrictions:

- Message-Oriented Middleware (MOM) is being used to provide communication between entities. This is validated in [MorganAcm05], and better reflects the needs of a peer-to-peer DVE with aura-based Interest Management.

  Some DVEs already use Message-Oriented Middleware to facilitate communication of world events. As an example, Sun Microsystems' Project Darkstar [Darkstar] is an open-source MMOG server platform that illustrates use of publish/subscribe message channels to communicate information between clients through a central server. It can be envisaged that the use of MOM to support DVE applications in this way will become more prevalent in the future.

- It is assumed that monitoring components can subscribe to receive messages from an application-controlled message channel without needing to navigate application-specific subscription logic. This is analogous to the assumption that the MeCo Probe (Section 3.3.4) is capable of probing a service provider.

- If a monitoring component needs to sample system messages it can connect to the same Internet Service Provider (ISP) as the DVE nodes.

- Each user is associated with a messaging application or server that is in no way shared with any other users active within the DVE. This simplifies the process of determining accountability for individual system messages.

- To provide complexity it is assumed that an observed DVE is providing a Massively Multiplayer Online Game (MMOG). This then incorporates issues of fairness (e.g. discovery of malicious behaviour), reliability, resource usage, and consistency of end-user experience, which must all then be considered.

4.4.2 Monitoring DVE Performance

The DVE Simulator and EOS tool assume a peer-to-peer approach to DVE management, but evaluate system behaviour by accumulating performance data in a centralised manner. The actions of simulated user objects are collated in one place (the simulator algorithm, as in Section 4.2.8), from which the occurrence of missed interactions is determined. Monitoring and evaluation is enacted in this way as there is no means for individual end-user nodes to determine the occurrence of missed interactions based only on the messages that they would normally receive.

In order to determine the quality of the end-user experience (or essentially the quality of the DVE provision itself), there would be a need to correlate events from all end-user machines to create a global view of DVE performance. A global view such as this could then be used to determine QoS measures as described by the DVE Simulator (e.g. the occurrence of missed interactions).

Centralised monitoring processes would ideally have some means of monitoring the behaviour of user entities within the virtual world, directly at the point at which user actions are imprinted upon the observed system. In the peer-to-peer approach this would be the messaging server or application associated with each DVE user. This is analogous to the monitoring of client behaviour within an application server as in the Provider-side MeCo (Section 3.3.3). In DVEs the 'client behaviour' would in fact be the behaviour of the shared DVE as managed collectively by all of the 'clients' (end-users).

In order to achieve a global view of DVE behaviour all of the messages produced by individual messaging applications (associated with individual end-users) would have

to be observed. One approach may be to have a monitoring application subscribe to the message channels associated with each user. However high-frequency messages are transmitted to specific entities within the DVE (i.e. those objects that are interacting with the sender). A monitoring station does not represent any object within the shared environment and so could not observe the complete set of messages (as it would not receive any high-frequency messages).

Another means of observing all messages produced during the lifetime of a DVE in a transparent manner would be to exploit an interceptor mechanism in the MOM subsystem. The use of interceptor-capable MOM platforms is illustrated in [MorganAcm05] and envisaged in [Banavar99]. It could be envisaged that use of interceptor mechanisms to extend DVE functionality will increase to provide greater customisation of game behaviour in open-source DVE platforms such as the one in Project Darkstar. Extensions could then include the provision of QoS control capabilities such as resource management and performance monitoring.

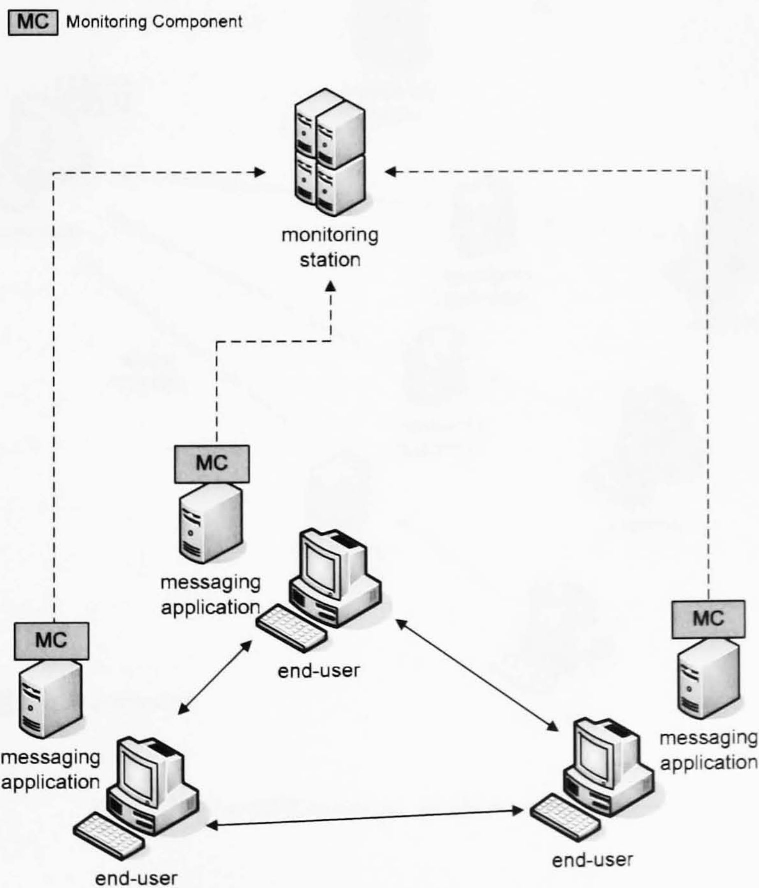


Figure 58 How DVE monitoring would work in practice

With this premise MOM servers or applications could incorporate an 'interceptor' mechanism similar to that described for MeCo Interceptors (Section 3.3.3). Additional functionality in the message processing stack could then be used to transparently monitor inter-object messages (as shown in Figure 58). The behavioural and positional properties of entities could then be determined from the messages that are being sent on their behalf. Individual object profiles could be collected centrally and then observed to determine how objects are interacting. Other observation processes could then be added afterwards if required, for example to establish Interest Management properties such as the occurrence of missed interactions.

4.4.3 Monitoring DVE Provision

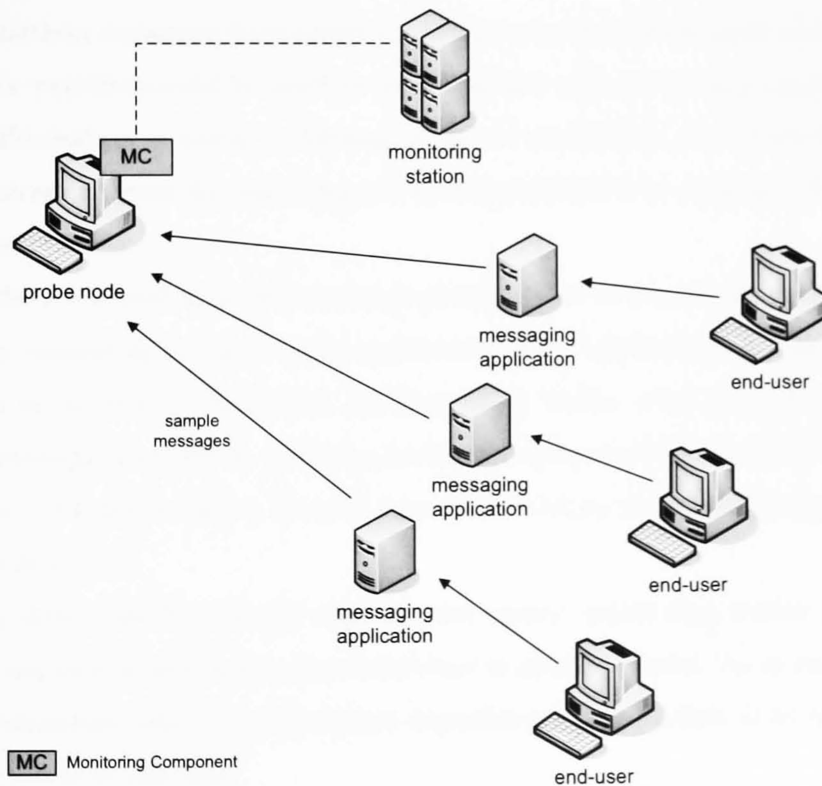


Figure 59 How DVE provision would be monitored

It may be necessary to monitor aspects of DVE provision outside of the virtual world experience. These include properties such as message availability and message

processing time within messaging servers (as opposed to Interest Management consistency properties as discussed previously). DVE provision in a peer-to-peer system should be observed as the view a system node has of other nodes. This is most readily achieved by determining the performance of message-sending between DVE nodes. To this end a dummy user machine could conceivably be integrated into the monitoring framework, similar to the MeCo Probe (Section 3.3.4). This probe machine could for instance be configured to receive heartbeat messages from all of the end-user messaging servers or applications.

As previously stated, when comparing a DVE monitoring infrastructure to the MeCo Framework the end-user entity within the DVE could be compared to a 'service client'. Users would be most readily affected by properties such as DVE consistency i.e. the occurrence of missed interactions etc. The collection of network nodes supporting the DVE is then analogous to a 'service provider'. Each 'provider' node is then concerned with 'service provision' metrics. For example if the probe machine receives heartbeat messages from nodes at set intervals, the arrival times of messages at the probe machine could be used to infer qualities such as message transmission jitter (i.e. fluctuation in latency). Messaging server availability and reliability could also be determined from the percentage of anticipated heartbeat messages that arrive at the probe machine.

In a DVE there is a need to control message production so as to maintain scalability as an arbitrary number of end-users enter and leave a DVE application. Just as artificial probe calls in the MeCo Framework add to network traffic, extra positional updates sent from messaging servers to a dummy machine could potentially flood the network and degrade DVE performance. As such care must be taken when deploying a 'probe' component in a DVE.

Monitoring data from the dummy machine and central monitoring station could be correlated and used to attribute system behaviour to specific parties. As an example, it could be determined whether performance degradation is attributable to an individual messaging server or end-user.

4.4.4 Augmenting the DVE Evaluation Framework

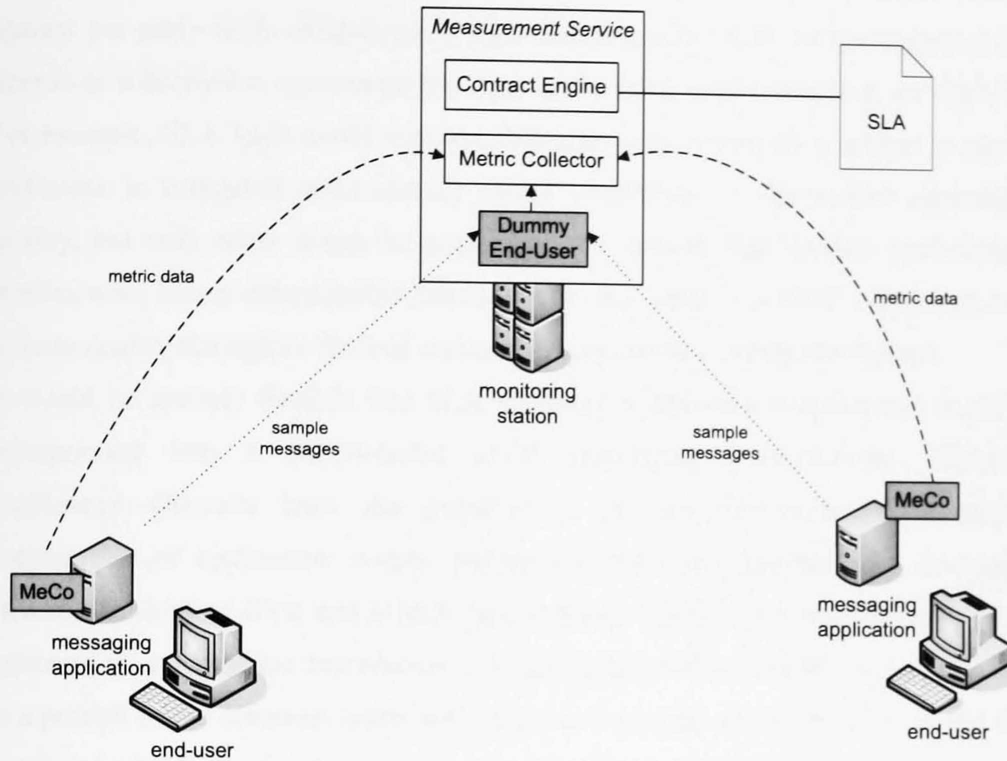


Figure 60 Incorporating contract evaluation into the DVE monitoring infrastructure

If monitoring of both application and system performance were realised, contract evaluation components could feasibly be incorporated into the DVE monitoring infrastructure (as in Figure 60). Service Level Agreements (SLAs) or simple performance expectations could be incorporated to manage the actions of interacting entities and DVE nodes. For example the concept of dynamic QoS management within a DVE for the management of resource allocation is discussed in [Nutt00]. SLAs may also be used where end-users pay to enter a DVE or MMOG and expect a certain quality of DVE provision. Conversely SLAs could be put in place to dictate expected end-user behaviour while participating in DVEs that require expensive infrastructure.

Logic for the evaluation of DVE performance is exemplified in the DVE Simulator and Evolutionary Optimisation Simulator (EOS). It has been shown that a centralised monitoring approach can be used to infer measurements relating to the performance of

both individual user machines and the DVE as a whole. Performance properties could potentially be inferred on a per-user or per-server basis, providing a foundation for accountability in service provision. Automated evaluation of performance metrics against per-party SLA obligations is then feasible, and could be incorporated into license or subscription agreements for a particular DVE application (e.g. an MMOG). For example, SLA logic could stipulate that a process to provide a refund to paying end-users is instigated automatically under conditions of diminished application quality, but only when it can be unambiguously proven that network performance metrics were below determinable thresholds. In this sense if a DVE experience is of inferior quality through no fault of end-users, they can be suitably reimbursed.

It would be entirely feasible that SLA violation notification mechanisms could be incorporated into a MOM-based DVE monitoring infrastructure. Violation notification channels have the potential to provide personalised and precise notifications of application events, perhaps beyond the capabilities of messaging channels in existing DVE and MMOG applications. DVE users could for instance be informed of an identified degradation in system performance, whether it is attributable to a portion of the communication subsystem or a specific end-user. Users could also be alerted of individuals who have been found to have altered the end-user application to change the behaviour of the DVE for their own gain. An example of such DVE disruption is when a game participant intercepts system messages within a P2P DVE that are not intended for them [Kabus05].

4.4.5 Different DVE Configurations

Different DVE configurations may affect how a DVE monitoring framework is deployed. If a DVE is managed by a centralised server (and not between peer nodes) the server alone could be monitored for application-level and service provision QoS properties. If a cluster of servers is used (as is the case with larger MMOGs) performance measurements would need to be taken within separate server machines. These separate views of application or server performance would then be coordinated centrally at a monitoring station.

Network-level protocols are used to support communication in most DVE applications today. To enable monitoring in these cases specialised hardware would

be required at the network level or within the DVE application itself, which is an impractical expectation.

4.5 Summary

- Distributed Virtual Environments (DVEs) are shared virtual worlds inhabited by geographically remote users. Users of a virtual world are able to interact with the generated environment and other entities such as in-game characters and other users. Data referring to user actions is propagated to other users through positional update messages.

- To achieve some measure of operational scalability DVEs incorporate Interest Management techniques to reduce the number of messages sent between users. Messages are then only exchanged between entities that are interacting. With the use of Interest Management it is however possible for interactions between entities to be missed.

- In order to examine the missed interaction problem on a per-DVE basis the configuration of the associated Interest Management mechanisms should be analysed. Through the application of principles developed in the MeCo monitoring & evaluation framework, a means of observing how a DVE configuration contributes to the occurrence of missed interactions has been created. This is presented in the form of the DVE Simulator [Parkin06].

- The DVE Simulator allows an application developer to configure simulated world parameters such as size and entity characteristics. Entity objects are created according to a configured set of behaviours. Their movements are followed throughout the DVE simulation as they travel towards assigned targets. During a simulation a number of parameters are observed including the number of messages that each object sends to other objects, and the number of missed interactions that have occurred between objects. This data allows an application developer to determine the performance of Interest Management attributes for a particular DVE configuration.

- The DVE Simulator relies upon informed knowledge of Interest Management on the part of the application developer. A desirable alternative would be to have a tool that can automatically determine optimum configurations for individual DVEs. The Evolutionary Optimisation Simulator (EOS) [Parkin07] was developed as an extension to the DVE Simulator. The EOS incorporates genetic algorithm techniques to discover and optimise promising Interest Management configurations for a given DVE.

5. Experimental Results

The effectiveness of the MeCo monitoring & evaluation framework (as described in Chapter 3) and both the DVE Simulator and Evolutionary Optimisation Simulator (EOS) (described in Chapter 4) must be assured through meaningful testing. For each application a series of tests was devised and conducted, with reasoning and test results presented here.

5.1 MeCo Framework

A series of tests was conducted with an instance of the MeCo Framework to determine how well it performed in operation, while also considering the requirements of the framework (as described in Section 2.6.4 and further discussed in Section 3.2).

5.1.1 Test Configuration

The MeCo Framework was tested across a LAN network using two machines. One machine acted as a server running the JBoss 3.2.7 Application Server with a Provider-side MeCo installed. The other machine managed instances of the MeCo Measurement Service. Both machines ran the Eclipse IDE (Version 3.0.2) with Version 0.5.33 of the Eclipse Colorer Profiler Plugin [Eclipsecolorer] integrated into the IDE. This allowed remote and local profiling of application components. The Profiler Plugin measured method call durations within the MeCo classes (with millisecond accuracy) and system memory usage for either set of observed classes (i.e. either the whole Provider-side MeCo or the Measurement Service). Relevant class packages were explicitly included within the profiling scheme, to allow observation of per-method timings and method call ratios. As a result of the Profiler Plugin co-existing on the same machines as the MeCo components, an undeterminable (but assumedly small) portion of the resource usage results can be attributed to the profiling tools.

The specifications of the two test machines were as follows:

- Server Machine:
    - *Model:* DELL Dimension 5150
    - *OS:* Windows XP Home Edition
    - *Processor:* Intel Pentium-4 2.80 Ghz
    - *Memory:* 512 MB RAM

- Measurement Service Machine:
    - *Model:* IBM ThinkPad T40p
    - *OS:* Windows XP Professional Edition
    - *Processor:* Intel Pentium-M 1400 Mhz
    - *Memory:* 512 MB RAM

EJB support is provided by default as part of the JBoss server installation. The JBoss server used Version 1.2.1 of the Axis API to provide SOAP support. The Measurement Service used the SLAng contract engine developed at University College London (UCL) [Skene04] to process contracts within instances of the 'SLAngSimpleEvaluationManager' class (see Section 3.3.6 for more information). The Provider-side MeCo and Measurement Service used the Java Message Service (JMS) [Jms] to communicate. For simplicity and observational purposes the JBoss server machine also acted as the JMS server.

Metric collection was restricted to a small number of service parameters to make it easier to discern the actions of different system components. A single metric calculation class was deployed in the JBoss server for each observed service protocol. This class measures how long a request takes to enter and leave the target application. A measurement class within the MeCo Probe (Section 3.3.4) determined the round-trip-time of fabricated requests for each service protocol (i.e. the time between each request leaving the probe and it returning). A 'complex measurement' class (see Section 3.3.6.2) within the Measurement Service calculated the average number of requests processed per minute for each observed protocol.

A number of services were deployed for testing purposes. Derivatives of a service that calculated the first ten numbers of the Fibonacci sequence were used. Computational

costs could then be observed while providing service logic simple enough to be deployed within both Enterprise JavaBeans (EJBs) and SOAP services. This provided consistent service behaviour when comparing monitored services using different application technologies.

5.1.2 Experiments

Experiments were conducted to establish whether the MeCo Framework satisfied its requirements (as in Sections 2.6.4 & 3.2) and to uncover any further notable performance characteristics. Analysis of the test results is provided in Section 5.1.4.

5.1.2.1 Deployment Profiling – Provider-side MeCo

It is useful to determine the resource requirements of deploying a Provider-side MeCo within the JBoss server. In the Provider-side MeCo deployment tests the initialisation stage of the supporting JBoss server was observed with different MeCo configurations. Results are discussed in Section 5.1.3.1.

The initialisation costs of the Provider-side MeCo and Provider Environment (see Section 3.3.3.2) were measured primarily as the time required to complete notable configuration processes, and the amount of system memory resources the overall system required (i.e. the JBoss server and integrated Provider-side MeCo). The Profiler Plugin was configured to exclusively monitor the complete set of Provider-side MeCo class packages and the main JBoss program (but not any specific JBoss class packages). This provided MeCo-specific performance data relative to the performance of the JBoss server.

The test configurations visited during deployment profiling of the Provider-side MeCo are as follows:

A.1 A standard JBoss server deployment with additional Axis SOAP libraries. This provided a base of comparison relative to the performance of a server with a Provider-side MeCo installed (as in tests *A.4-A.6*).

A.2 A standard JBoss server deployment as for test *A.1*, with a single EJB-based service deployed to the server to provide measurements of resource usage attributable to an individual service.

A.3 A standard JBoss server deployment as for test *A.1*, with a single SOAP-based service deployed to the server. In conjunction with test *A.2* this would illustrate any inherent differences in resource requirements between E-Commerce technologies.

A.4 A standard JBoss server deployment (as for test *A.1*) augmented with a Provider-side MeCo with no MeCo Interceptors attached to it. This would illustrate the basic resource requirements of the Provider Environment.

A.5 A Provider-side MeCo deployment with an EJB-enabled MeCo Interceptor configured from the Measurement Service (including details for a single EJB service contract). This would indicate the resource requirements of specialised MeCo Interceptor initialisation.

A.6 As for test *A.5* but with a SOAP-enabled MeCo Interceptor and a SOAP service contract deployed. Both tests *A.5* and *A.6* would illustrate any differences across MeCo Interceptor deployments.

To compensate for discrepancies between observations each test was carried out three times and average performance results calculated from the complete result set.

5.1.2.2 Deployment Profiling – Measurement Service

Tests were performed to determine the initialisation cost of a Measurement Service instance. This cost was measured in terms of both memory usage and the time taken to complete configuration of the Measurement Service. The Profiler Plugin was calibrated to exclusively observe all class packages within the Measurement Service. The test scenarios visited during profiling of the Measurement Service deployment were:

B.1 Initialising the Measurement Service with an empty contract set. This would illustrate the basic configuration costs of its core components.

B.2 Initialising the Measurement Service with an SLA pre-positioned in the file system ready for loading prior to initialisation. The SLA describes an EJB-based service. This would reveal the costs of per-contract configuration.

B.3 As with test *B.2*, but with 2 SLAs describing identical EJB-based services. This would show how deployment cost increases with an additional monitored service.

B.4 As with test *B.2*, but with 3 SLAs describing identical EJB-based services. The results of this test may suggest trends concerning increased numbers of monitored service contracts.

B.5 As for test *B.2* but with an SLA that describes a SOAP-based service. Comparison of the results for tests *B.2* and *B.5* would reveal differences in configuration cost attributable to different application technologies.

B.6 As with test *B.5* but with 2 SLAs describing identical SOAP-based services.

B.7 As with test *B.6* but with 3 SLAs describing identical SOAP-based services

Results for the aforementioned tests are found in Section 5.1.3.2. As with the experiments described in Section 5.1.2.1 tests were carried out three times and average results obtained.

5.1.2.3 Operational Performance Profiling – Multiple Services

Profiling of the MeCo Framework was conducted as it monitored multiple service contracts simultaneously. This allowed observations to be made as to how the system behaved over a meaningful space of time as the number of monitored services and service types increased. A number of contracts and services were deployed across the

MeCo Framework. This included a combination of both EJB- and SOAP-based services.

Tests were conducted over ten minute periods to demonstrate system behaviour over a short but meaningful period of time. The MeCo Probe registered as a 'client' to each observed service, generating the service requests that were monitored (with a 15 second probing interval). This illustrated the capabilities of the Measurement Service to simulate valid service consumer activity. The Profiler Plugin applications in both the Provider-side MeCo and the Measurement Service provided reciprocal sets of performance results.

The tests were as follows:

C.1 A single EJB-based service.

C.2 Two EJB-based services. It is desirable to determine how monitoring processes operate when multiple (but similar) services are monitored.

C.3 Three EJB-based services. The results of this test may serve to illustrate how the MeCo Framework scales to observe multiple services.

C.4 A single SOAP-based service. The results for both EJB- and SOAP-based services provide a comparison between the two technologies.

C.5 Two SOAP-based services.

C.6 Three SOAP-based services.

C.7 One EJB-based service & one SOAP-based service. This would illustrate any notable behaviour that arises when different service technologies are monitored simultaneously.

The MeCo Framework was evaluated in terms of processing time and system memory requirements. One intention of this set of tests was to determine how monitoring data is managed over time. Results pertaining to method call durations are also described in terms of their average cost over the observation period. As with previous tests

profiling data for each test was obtained from averages over three distinct observations. The results for this group of tests are discussed in Section 5.1.3.3.

5.1.2.4 Accuracy Testing – Correctness of Measurements

A set of tests was conducted to determine whether the measurement data retrieved by the different components of the MeCo Framework can be regarded as valid. To test the validity of MeCo measurements, the metric data gathered from the Provider-side MeCo and MeCo Probe was observed with a number of altered services. Measurements of server processing time and request round-trip-time were recorded, from the Provider-side MeCo and MeCo Probe respectively. Direct measurements of accuracy cannot be made at the exact points at which the MeCo components operate without essentially mimicking their functionality, and so a series of services were monitored with differing artificial delays inserted within the internal logic. These services are derivatives of the Fibonacci service described in Section 5.1.1, and so also include logic to calculate a series of numbers in the Fibonacci sequence. This process should take a measurable amount of time to complete, and should also register within the timing measurements. The tests that were conducted are described as follows:

D.1 An EJB-based Fibonacci service with no additional delay.

D.2 An EJB-based Fibonacci service with a 100 millisecond delay incorporated into the service logic.

D.3 An EJB-based Fibonacci service with an added 2 second delay.

D.4 An EJB-based Fibonacci service with an added 5 second delay.

D.5 A normal SOAP-based Fibonacci service, with no additional delay.

D.6 A SOAP-based Fibonacci service with an added 100 millisecond delay.

D.7 A SOAP-based Fibonacci service with an added 2 second delay.

D.8 A SOAP-based Fibonacci service with an added 5 second delay.

No system profiling was conducted as part of these tests. The simple aim of these tests was to determine if consistent measurements were obtained both within the server and from the MeCo Probe, and whether the artificial delays were accurately observed by the MeCo Framework components. As such, the measurements taken by the MeCo Framework itself were the product of each test. As with previous tests, results were obtained as averages across three distinct instances of each test configuration.

5.1.3 Experimental Results

5.1.3.1 Deployment Profiling – Provider-side MeCo

The results for tests *A.1-6* (see Section 5.1.2.1) are described here. In test *A.1* a basic server requires on average 25.5MB of memory to operate. In test *A.2* (a single EJB service) once the JBoss server is configured with an EJB-based service it requires approximately 2-3MB of memory over that found in test *A.1*. However when a single SOAP-based service is deployed (as in test *A.3*) system memory requirements are directly comparable to those of a basic server. In test *A.4* the JBoss server has a Provider-side MeCo installed, which demands approximately 2.5-3MB of additional system memory.

In test *A.5* monitoring logic is deployed to the Provider-side MeCo for a single EJB-based service. This accounts for an approximate 5.5MB of additional memory usage over the basic deployment of test *A.1*. This correlates with the combined deployment costs of a single EJB service and a standalone Provider-side MeCo. Deployment of monitoring logic for a SOAP-based service (test *A.6*) registers memory usage comparable to that of the basic server and Provider-side MeCo combined. The results for tests *A.5* & *A.6* indicate that monitoring logic for a single service requires negligible resources.

In tests *A.4-6* configuration of the Provider-side MeCo consistently took less than 5% of the total server start-up time (which itself ranged between 16 and 22 seconds). During tests *A.4-6* the MeCo XMBean (see Section 3.3.3.3) took 83ms to initialise (less than 1% of start-up time). In test *A.4* the Provider Environment (see Section

3.3.3.2) took 26ms to initialise without a contract. In tests *A.5* & *A.6* the Provider Environment took 42ms (0.35% of the start-up time) to complete configuration through update calls from the MeCo XMBean. When configuring the JMS messaging component (see Section 3.3.5) the JMS subsystem took 120ms to deploy during tests *A.5* & *A.6*. An additional 39ms was also required in the latter tests to register the relevant JMS topics. Configuration of a Provider-side MeCo therefore appears to take little time in relation to the requirements of the associated server.

When deploying an EJB-based MeCo Interceptor (test *A.5*) the 'ClientIPInterceptor' (see Section 3.3.3.3) took close to zero milliseconds to initialise. Both the EJB-based MeCo Interceptor in test *A.5* and the SOAP-based MeCo Interceptor deployed in test *A.6* took close to zero milliseconds to configure. These results illustrate that MeCo Interceptor configuration is relatively inexpensive independent of the associated application technology.

5.1.3.2 Deployment Profiling – Measurement Service

The results accompanying tests *B.1-7* (see Section 5.1.2.2) are detailed here. In test *B.1* (a basic Measurement Service deployment) memory resource requirements settled at approximately 10MB once initialisation of the Measurement Service was completed. This sustained draw on memory can be attributed to the SLAng engine within the SLA Manager instance, the Measurement Service GUI and the parser objects used to read XML-based configuration files. These objects must all be held in memory during the lifetime of the Measurement Service.

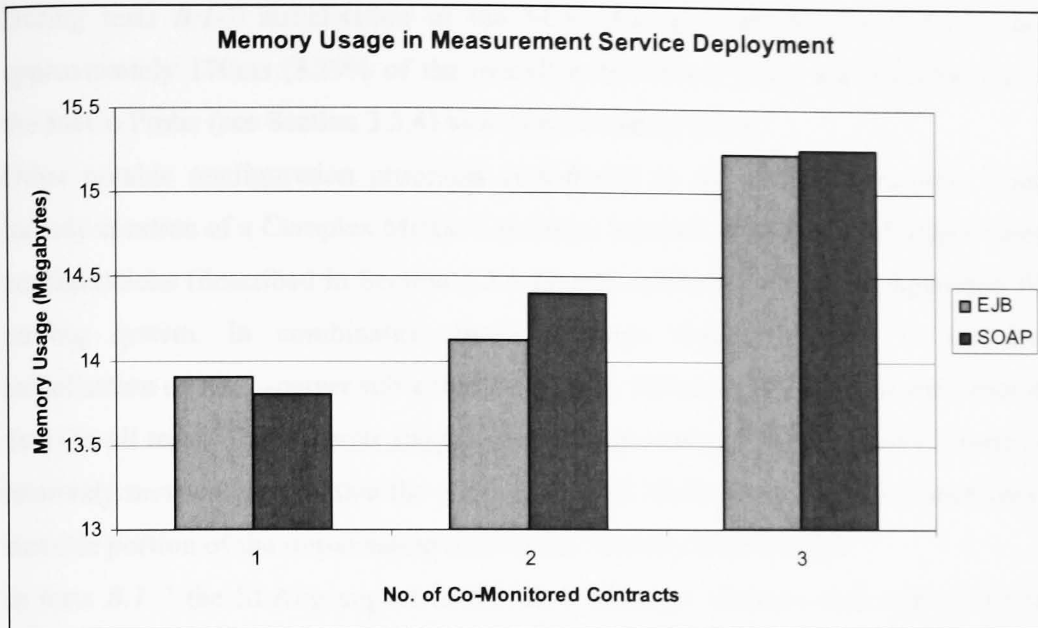


Figure 61 Memory usage during deployment of the Measurement Service

In tests *B.2-7* there are increased memory demands over those of a basic Measurement Service deployment (as shown in Figure 61). In tests *B.2-7* the increases in memory usage can be attributed to the SLA engine reading in and retaining data relating to service contracts. In both tests *B.2* (a single EJB service contract) & *B.5* (a single SOAP service contract) the system required around 13.8-13.9MB after a contract was loaded. This suggests each contract requires a sustained amount of system memory relative to a basic Measurement Service deployment. It also indicates that initialisation costs for application-specific components (i.e. the MeCo Probe) are similar across different service technologies.

In tests *B.3* (two EJB services) & *B.6* (two SOAP services) memory usage was again comparable. Memory usage when deploying two EJB-based service contracts averaged 14.13MB, whereas for two SOAP-based service contracts memory usage reached an average of 14.4MB. These results indicate increased memory usage attributable to storage of data pertaining to additional contracts. This is further validated in tests *B.4* (three EJB services) & *B.7* (three SOAP services), where memory usage increases to an average of around 15.24-15.25MB for both service types. These results indicate that the memory required to retain a contract is relatively small compared to the requirements of the Measurement Service itself.

During tests *B.1-7* initialisation of the SLA Manager (see Section 3.3.6.2) took approximately 170ms (3.39% of the overall initialisation time), and initialisation of the MeCo Probe (see Section 3.3.4) took approximately 65ms.

Other notable configuration processes contributed to the initialisation time. These include creation of a Complex Metric Calculator instance, finalisation of helper object configurations (described in Section 3.3.5.2), and calibration of the configuration file polling system. In combination these elements required 198ms to initialise. Initialisation of XML-parser sub-components took 703ms (13.97% of the initialisation time) in all tests. These results suggest that configuration of MeCo-specific classes is relatively inexpensive, but that the configuration of XML-parsing classes constitutes a sizeable portion of the initialisation time of the Measurement Service.

In tests *B.1-7* the SLAng engine in the SLA Manager instance took approximately 1.3s to initialise (contributing 25.92% of the total initialisation time). In tests *B.2* & *B.5* (which involved a single EJB- and SOAP-based service contract respectively) the SLAng engine took 200ms (approximately 3.98% of the total time) to read a single contract. This is contrasted with a configuration time for the internal contract store of close to zero milliseconds. The SLAng engine therefore contributed greatly to the initialisation time of the Measurement Service. However, the processing of an individual contract and subsequent contract-specific configuration of monitoring and evaluation processes is relatively cheap in terms of processing resources.

In tests *B.1-7* the MeCo MBean Communicator (see Sections 3.3.3.3 & 3.3.6) took around 350ms to initialise (6.5% of total initialisation time), including time to communicate with the JBoss server to determine the status of the MeCo Interceptor. Another 20ms were required to configure the Provider-side MeCo to monitor an individual service in tests *B.2* & *B.5*. During test *B.1* JMS server lookup operations took 191ms (3.78% of the total initialisation time) while creation of a single metric topic in tests *B.2* & *B.5* took 21.67ms. As with other measurements that include time for processes to communicate over the network, these figures are subject to the conditions of the supporting network environment. The processing time required by the JMS communication subsystem is small, and calibration of individual communication channels is inexpensive.

During test *B.1* the GUI component (see Section 3.3.6.4) was a major contributor to the initialisation time, requiring 252ms to initialise and 221ms to create the GUI window. In tests *B.2* & *B.5* per-contract data displays and chart instances derived

from the third-party API [Jfreechart] took 226ms to configure. For a single contract GUI initialisation constituted approximately 15% of the entire Measurement Service initialisation.

In tests *B.2*-7 the top-level Probe Manager instance within the MeCo Probe (see Section 3.3.4) took 15ms (0.30% of the total initialisation time) to read data from each probe descriptor (see Section 3.3.4). In test *B.2* a single EJB 'Probe Activator' instance (a simulated service consumer) took 363ms (approximately 8% of overall time) to configure and 13ms to become operational. In test *B.5* a single SOAP 'Probe Activator' instance required 343ms to configure and 20ms to become operational. This indicates that the resource requirements of both EJB- and SOAP-based Probe Activators are comparable. These results also illustrate that the MeCo Probe only requires a minimum of resources to configure.

5.1.3.3 Operational Performance Profiling – Multiple Services

The results for tests *C.1-7* (see Section 5.1.2.3) are described in this section.

When considering memory usage patterns within the JBoss server, the various test configurations required similar amounts of system memory (ranging between 25-32MB across all scenarios). This is in line with the results of the deployment tests in Section 5.1.3.1. The amount of memory required by the Provider-side MeCo remained constant throughout each test run. This is because the Provider Environment requires a small, static amount of configuration data to observe a service, and measurement data is not retained on the server-side during service monitoring.

During request interception the EJB-based MeCo Interceptor (tests *C.1-3 & C.7*) took close to zero milliseconds to process a single request (regardless of the number of co-monitored EJB services). The SOAP-based MeCo Interceptor used in tests *C.4-6 & C.7* required a similar amount of time to process a request. This indicates that different MeCo Interceptor implementations require a comparable amount of time to monitor individual service requests.

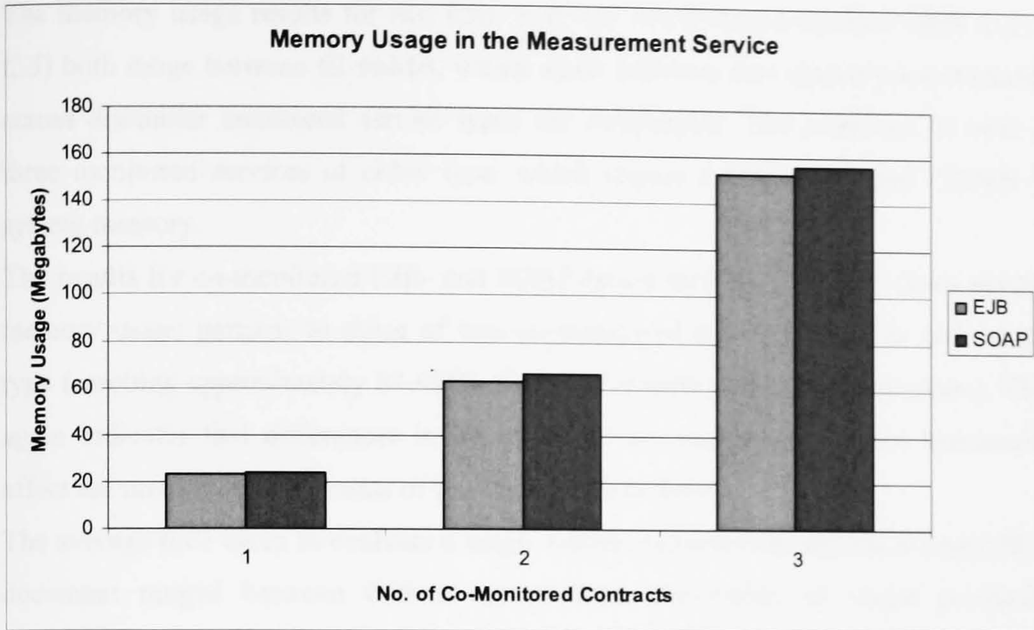


Figure 62 Memory usage in the Measurement Service

Figure 62 illustrates memory usage after 10 minutes of operation within the Measurement Service for tests *C.1-6*. Memory usage for both a single EJB service (test *C.1*) and a single SOAP service (test *C.4*) are comparable. After 10 minutes of deployment both service types reach a level of memory usage of approximately 23MB. The amount of memory being used is significantly higher than at initial deployment (described in Section 5.1.3.2). This suggests that extra memory is being used by some of the components of the Measurement Service. As the Contract Manager and SLA Manager do not store any information relating to service usage, the graphical interface is the greatest potential contributor to the increased memory requirements. The GUI retains measurement data but also presents the data in chart and plot objects which must be held in memory.

Increases in memory usage are also evident across tests *C.2* & *C.5* (two co-monitored EJB- & SOAP-based services respectively), and tests *C.3* & *C.6* (three co-monitored EJB- & SOAP-based services respectively). Memory usage may be expected to increase as additional services are co-monitored, as more service usage data must be maintained by the Measurement Service, but also displayed in the graphical component.

The memory usage results for two EJB- and two SOAP-based services (tests *C.2 & C.5*) both range between 62-66MB, which again indicates that memory requirements across dissimilar monitored service types are comparable. The same can be said of three monitored services of either type, which require between 151 and 158MB of system memory.

The results for co-monitored EJB- and SOAP-based services (test *C.7*) show similar memory usage patterns to those of two co-monitored services of either technology type (reaching approximately 64.6MB after a 10 minute period of observation). This again indicates that differences in observed service technology do not necessarily affect the memory requirements of the Measurement Service.

The average time taken to evaluate a single metric measurement against a single SLA document ranged between 0.08ms and 0.75ms. Processing of single per-probe measurements and calculation of 'complex metric' measurements took close to zero milliseconds in test scenarios *C.1-7*. These results indicate that per-measurement evaluation is inexpensive.

For EJB-based services (tests *C.1-3 & C.7*) a single activation of the MeCo Probe took close to zero milliseconds, whereas for SOAP-based services (tests *C.4-6 & C.7*) a single probe activation took between 20 and 100 milliseconds. This indicates that service technology influences the processing demands of the MeCo Probe.

Within the server the 'ClientIPInterceptor' used in the EJB-based configurations (tests *C.1-3 & C.7*) consistently required around 45ms when called. This can be attributed to negotiation between the client and server across the network.

Measurement updates from the Provider Environment took between 0.12ms and 0.5ms to create and deliver to the MOM subsystem in all test cases. In all tests *C.1-7* the time required to process metric information within the MOM component of the Measurement Service varied between 0.6ms and 2.5ms. The time required to communicate measurement data between remote components of the MeCo Framework can be regarded as minimal in light of the latter results.

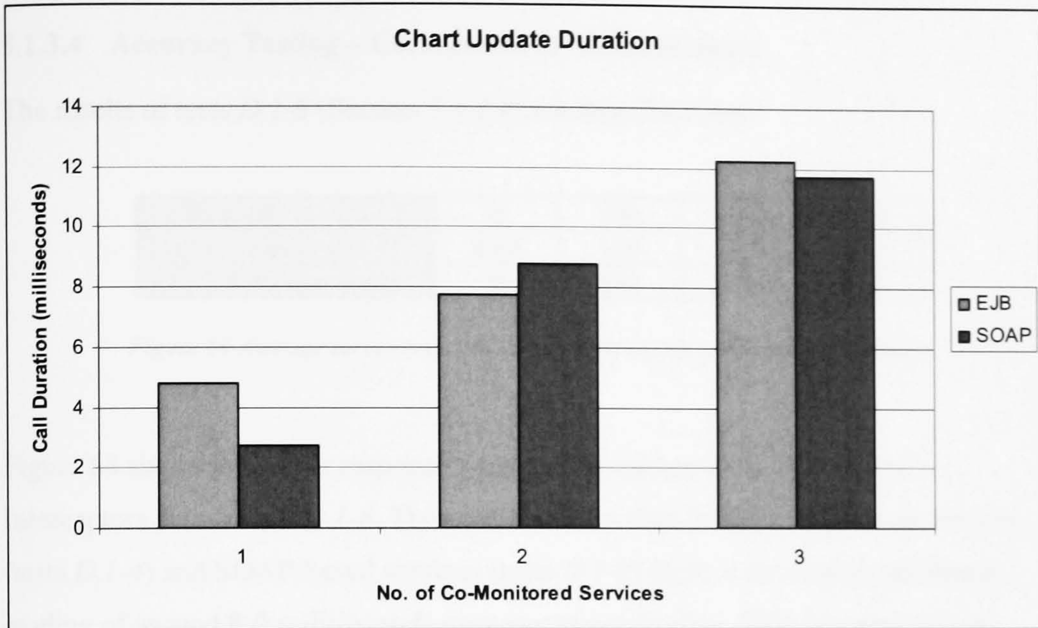


Figure 63 Chart update duration during operation of the Measurement Service

Figure 63 illustrates the average duration of update calls to the Measurement Service GUI chart objects after 10 minutes of activity for tests *C.1-6*. The processing time required to update the interface can be seen to increase with the number of simultaneously monitored services. This is logical as there will be more service activity to be presented to observers. Considered in combination with the memory usage results in Figure 62, it may be concluded that as time progresses the interface charts are the largest contributor to resource demands within the Measurement Service.

5.1.3.4 Accuracy Testing – Correctness of Measurements

The results of tests *D.1-8* (Section 5.1.2.4) are described here.

Delay (ms)	0	100	2000	5000
EJB Measurement	6.67	109	2007	5010
SOAP Measurement	8	109	2008	5008

Figure 64 Average server response time measurements with introduced delays

Figure 64 shows the server response time measurements taken by the MeCo Interceptors across tests *D.1-8*. These results show that for both EJB-based services (tests *D.1-4*) and SOAP-based services (tests *D.5-8*) there is an average additional reading of around 8-9 milliseconds over any injected delay. This suggests that on average the observed service logic takes just under 10 milliseconds to compute the requested chain of numbers within the Fibonacci sequence.

In all of the tests the measurements that were taken account for the delays that were introduced, suggesting that the Provider-side MeCo is capable of collecting accurate metric measurements. The consistency of the readings across all of the test scenarios also indicates that the measurements are valid. The similarity of results between service types also indicates that both the EJB-based and SOAP-based implementations of the MeCo Interceptor are able to take measurements at points that are equally close to the observed service logic. This suggests that there is no disadvantage from applying monitoring to one service technology over another.

Delay (ms)	0	100	2000	5000
EJB Measurement	10	110	2010	5008
SOAP Measurement	10	110	2008	5009

Figure 65 Round-trip-time measurements with introduced delays

Figure 65 shows round-trip-time measurements for tests *D.1-8*. The results suggest that when taking into account any artificial delays and the time to process numbers in the Fibonacci sequence, the round-trip-time (RTT) of requests is between 0 and 2 milliseconds.

The results are consistent across both service types, suggesting that the service technology being observed does not alter the accuracy of measurements compared to any other.

5.1.4 Performance Analysis

The results in Section 5.1.3.1 (Provider-side MeCo deployment) show that resource requirements during configuration of the Provider Environment are minimal. It was found that with a basic JBoss server the Provider-side MeCo requires at most 3MB of system memory (of an approximate 30MB total used by the server). The results in Section 5.1.3.1 illustrate that deployment of protocol-specific MeCo Interceptors and per-metric measurement capabilities is relatively inexpensive. The same can also be said of the MOM subsystem.

The Measurement Service exhibits sustained system memory resource demands (as seen in Section 5.1.3.2). This is due to the SLA Manager, the contracts being observed and the GUI component, which must all be maintained in memory. Within the SLA Manager the SLAng engine retains contracts once they have been read into the system. From Figure 61 (depicting memory demands during deployment of the Measurement Service) it can be seen that the storage of data relating to an individual contract requires little system memory. This suggests that once the SLAng engine and SLA Manager have been configured they are relatively inexpensive to maintain. The observations of Measurement Service deployment suggest that memory requirements depend in part upon the contract engine that is used. The SLA Manager implementation can also influence how long the Measurement Service takes to become operational.

Additional results from Section 5.1.3.2 illustrate that initialisation costs of different protocol-specific MeCo Probes are equivalent. As such, the service technology being observed does not affect the deployment readiness of the Measurement Service.

Meaningful load testing of service participant scalability was planned through use of the Apache JMeter load-testing application [Jmeter]. This could not be conducted due to the memory requirements of the profiling tool under intensive observation conditions. However, resource requirements based on service load were inferred from Section 5.1.3.3 (multiple service monitoring). Memory usage within the Measurement Service over time is governed primarily by the aggregated costs of displaying

monitoring data in the GUI. As more data associated with more services is collected, charts and tables in the interface must retain more usage information. These trends are natural as more data held within the system equates to greater requirements for system memory. Applied scalability engineering (as recommended in Section 2.6.4.4) may be employed to ease such demands. There is potential for such features to be incorporated into the Measurement Service in the future.

It is worth noting that the resource demands of the GUI are governed by not just the number of services being monitored, but also the regularity with which monitored services are both probed by the MeCo Probe and used by consumers. For instance, if service clients use an observed service more, there will be more measurement data to present. Also, if service probes are conducted less often, active measurement updates will be presented with reduced regularity. As such it may be difficult to gauge the resource requirements of the Measurement Service before it becomes operational.

The contract engine that is used also has the potential to contribute to memory demands during extended use of the MeCo Framework. If a contract engine is chosen which retains a great deal of information about service usage (much in the same way that the GUI does), it will require more memory to maintain records of usage data. It would therefore be preferable to employ a contract engine which has been designed to accommodate extended use (e.g. through data-archival techniques that reduce the resource footprint of old monitoring data).

In the results of Section 5.1.3.3 it is apparent that within the Measurement Service memory usage is comparable between service configuration instances using different protocols (e.g. two EJB- or two SOAP-based services). Any differences in system performance are encountered in the first instance from the activation of probe calls. The average durations of individual probe activations suggest that different activation costs and processing loads are exhibited depending on the nature of the service technology being used. This ultimately means that processing requirements could differ when monitoring different types of services.

The results in Section 5.1.3.3 regarding the Provider-side MeCo showed that once metric collection components were calibrated the memory requirements remained constant and did not exceed the original deployment costs. Monitoring data remains in the Provider Environment only until it is transmitted to the Measurement Service and as such is short-lived. This helps to restrict the memory demands of the Provider-side MeCo during deployment. The limited and constant resource requirements of the

Provider-side MeCo indicate that it is relatively simple to manage on a server platform, and would therefore require little in the way of resource management on the part of the server operator.

During long-term service monitoring the MeCo Framework was relatively fast at collecting and processing metric data. This can also be said of the JMS interface, which proved to be very fast at packaging, transmitting, and unpacking metric updates as they moved around the system.

The results of the accuracy tests in Section 5.1.3.4 indicate that no one service technology is more or less accurate for measuring service behaviour within the MeCo Framework. The results also indicate that the MeCo Framework can accurately measure metrics such as server response time and request round-trip-time.

Finally, it is worth noting that outside of this thesis an earlier version of the MeCo Framework had been successfully deployed to monitor an existing auction-style Internet application (as part of the "Trusted and QoS-Aware Provision of Application Services" project [Tapas]). This provides an example of the framework in operation with a complete and complex system.

5.2 DVE Simulator

Experiments were conducted with the DVE Simulator to determine appropriate values for Interest Management parameters for a particular DVE configuration. The goal was to determine the optimal time interval at which heartbeat messages should be sent and to what size object auras should be set. Optimal values for these parameters would minimise the occurrence of missed interactions and limit the use of networking and processing resources, so as to maintain consistency and scalability. Experiments with the Evolutionary Optimisation Simulator (EOS) are described in Section 5.3.

5.2.1 Experiments

Two sets of experiments were conducted to determine the influence of various object aura sizes and heartbeat message intervals upon the occurrence of missed interactions (both complete and partial, as described in Section 2.4.4) and the number of messages sent during a DVE simulation. In each experiment the majority of global and object-class-specific parameters kept the same values:

- Number of objects: 50
- Virtual world size: 5000^3
- Number of iterations: 500
- Number of targets: 2
- Distribution of object types: 25% (equal distribution)
- Achievable object velocity: ranging between 10-20 for all objects
- High-frequency message interval: 5 (for all objects)
- Network latency: 2 (for all objects)
- Processing latency: 1 (for all objects)

These values were selected so as to model an environment with a low number of objects in a high performance network (i.e. latency effects are present but small). Objects move relatively quickly in the given simulation – objects will be generated that are capable of traversing the world within the execution time of the experiment. An object that enters the world at one corner of the virtual space then has the potential to reach a target that may be positioned at the opposite corner before the simulation completes. The low number of targets increases the chance of interactions occurring between objects as the simulation progresses, while the object-type distribution introduces a degree of variety in object behaviours within the simulation.

Targets do not relocate during the experiments, giving most objects a reasonable chance of reaching their designated targets during the execution period. This guarantees that aura overlaps will be common, increasing in regularity as a simulation progresses (thereby modelling DVE performance during increased object activity).

In the first series of experiments the heartbeat message interval was gradually increased from 5 through to 50 inclusive, leaving aura sizes for all objects at 80 for all tests. In the second series of experiments the aura size was increased from 5 through to 300 inclusive for all objects, with heartbeat message intervals fixed at 25 for all objects. To account for inherently random object behaviour, in each experiment ten simulations were conducted and averaged results derived.

Sample object and target distribution at the start of a DVE simulation is shown in Figure 66. Entity distribution along the X and Y axes is described in the left-hand diagram, with distribution along the Y and Z axes shown in the right-hand diagram.

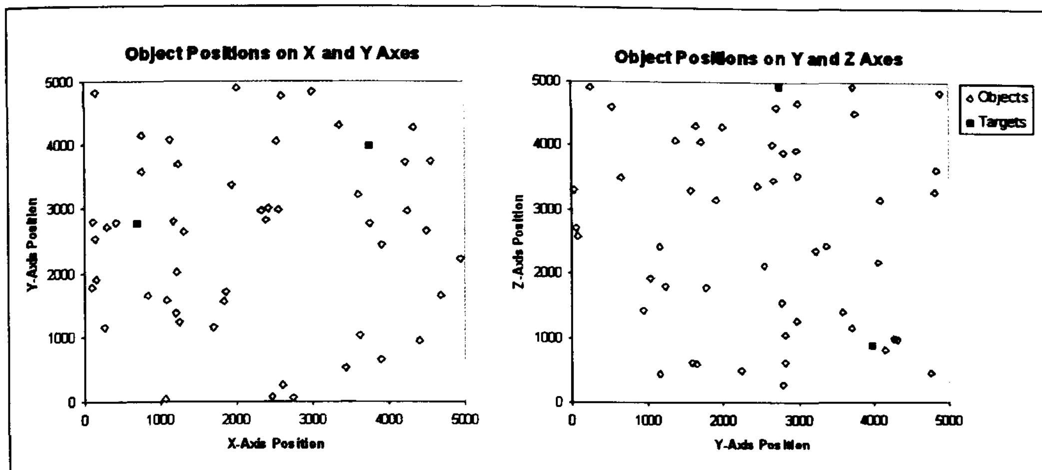


Figure 66 Object and target distribution within a sample DVE simulation

5.2.2 Experimental Results

Experimental results are shown in Figure 67-Figure 72. First to be considered are the graphs of different heartbeat message intervals (Figure 67-Figure 69).

5.2.2.1 Heartbeat Interval

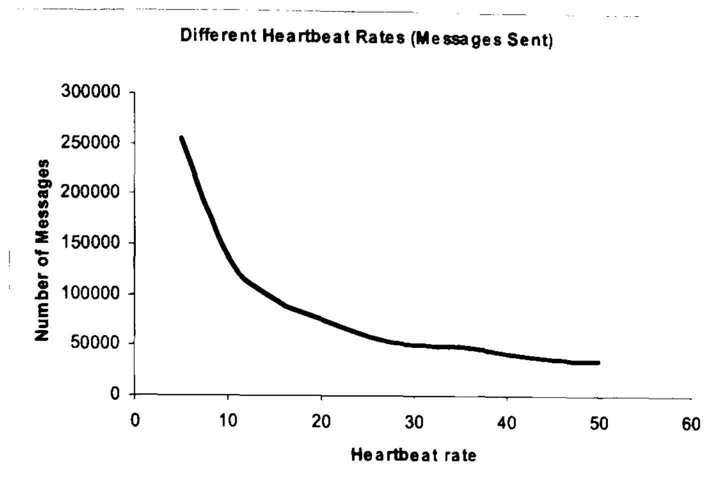


Figure 67 Number of messages sent as function of heartbeat interval

In Figure 67 it is observed that as the heartbeat message interval increases the number of messages sent decreases. This decrease is significant in that at a heartbeat interval of 5 there are ~250,000 messages sent, compared with ~50,000 when the heartbeat interval is 50. This indicates that heartbeat messages dominate the number of sent messages. This is to be expected since a heartbeat message from one object is broadcast to all other objects. The following calculation approximates an estimation that a DVE application developer may make when determining how many messages are sent in the lifetime of a DVE with this particular configuration:

1) Determine number of messages sent if all objects send messages to all other objects:

$$Number\ of\ objects * Number\ of\ objects - 1 = 50 * 49 = 2450$$

2) Determine the number of times all objects would send messages to all other objects (using heartbeat intervals of 5 and 50 for these examples):

$$Number\ of\ iterations\ /\ heartbeat\ interval = 500\ /\ 5 = 100 \qquad (interval = 5)$$

Number of iterations / heartbeat interval = 500 / 50 = 10 *(interval = 50)*

3) From the above calculations determine, approximately, how many heartbeat messages are sent overall:

*No. of messages * No. of message-sends = 2450 * 100 = 245, 000 (interval = 5)*
*No. of messages * No. of message-sends = 2450 * 10 = 24, 500 (interval = 50)*

Figure 67 approximates the estimate of a heartbeat interval of 5 as in the above calculations. However as the heartbeat interval increases to 50 the difference between the associated estimate and the actual values deviates significantly. This indicates that high-frequency message exchange (dictated by aura overlap detection) constitutes a greater quotient of the messages produced as the regularity of heartbeat messages decreases. With respect to messages sent, Figure 67 indicates that the simulator provides results equivalent to those that a developer can best estimate and is not, therefore, revealing any unexpected results. The graph is most informative as heartbeat messages become less dominant among all of the messages sent. The results show that the estimates that a DVE application developer may make as to the number of messages produced during the lifetime of a DVE may not correlate with the actual performance of the DVE itself. Furthermore these results validate use of the DVE Simulator over ad-hoc developer estimates in determining optimal Interest Management configurations.

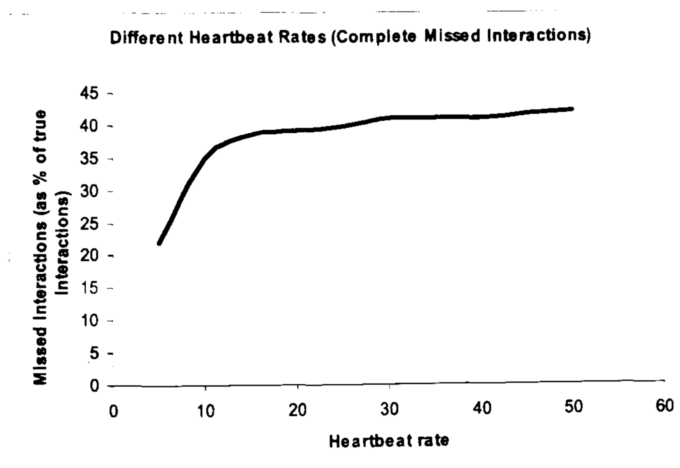


Figure 68 Quotient of missed interactions as function of heartbeat interval

The graph in Figure 68 relates the heartbeat interval to the quotient of missed interactions that occurred with specific DVE configurations. The latter quotient is arrived at by determining how many object interactions occurred during a completed simulation, and how many of these interactions were completely missed by the interacting objects.

Figure 68 shows a rise of missed interaction occurrence from 20% to almost 40% for heartbeat intervals of 5 to 10. After an interval value of 10 the graph flattens out, rising only a few percentage points between heartbeat interval values of 10 and 50. This indicates that if a developer wants to avoid complete missed interactions for this particular DVE configuration, heartbeat message intervals lower than 10 should be considered. Figure 68 also indicates that a heartbeat interval of 50 could be used over a value of 15 with comparable results (thereby saving networking and processing resources).

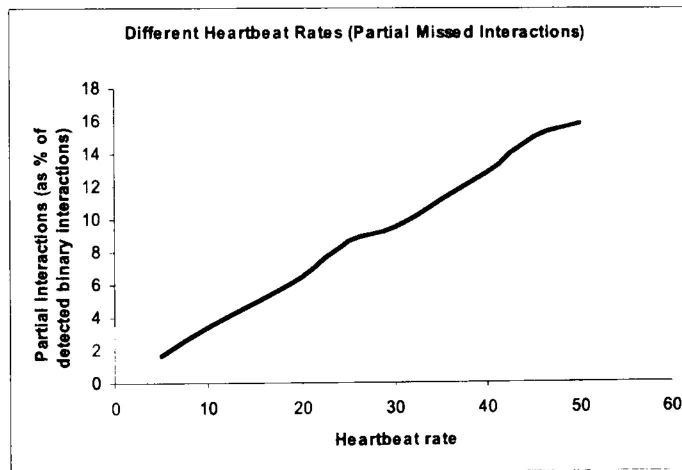


Figure 69 Quotient of partial missed interactions as function of heartbeat interval

In Figure 69 the percentage of partial missed interactions rises in correlation with the heartbeat interval. This graph does not flatten out in the same manner as the complete missed interactions graph of Figure 68. Although the number of complete missed interactions that occurs differs little between heartbeat rates of 10 and 50, there is an associated increase in occurrence from 4% to 16% for partial missed interactions in the same range.

5.2.2.2 Object Aura Size

The graphs of Figure 70-Figure 72 show the effects of varying object aura size in relation to the number of messages sent during a simulation, together with the number of complete missed interactions and partial missed interactions that occur. For DVE application developers, estimating appropriate object aura size values is not as straightforward as estimating how many messages will be sent (as in Section 5.2.2.1). Application developers could assume the worst-case scenario when determining aura size by envisaging how far two objects can travel directly towards and past each other at full speed during a heartbeat message interval. It can then be argued that object auras must be large enough to cover this distance. In the observed scenario this would give a large aura size, greater than 200. Such an estimate might not prove useful however, as objects may not necessarily be moving towards each other. Even if this were the case it cannot be assumed that movement is directed towards a specific object.

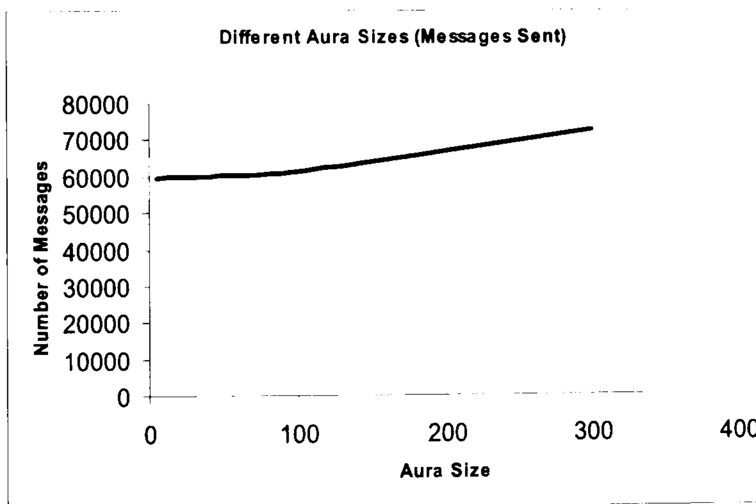


Figure 70 Number of messages sent as function of aura size

Figure 70 indicates that the number of messages sent increases slightly as object aura size increases. This suggests that heartbeat messages contribute heavily to message counts (given the earlier estimate based around Figure 67). As the heartbeat interval remains unchanged the rising curve illustrates how high-frequency messages form a

larger portion of all the messages sent as object aura size increases. As size of the virtual environment remains the same across all of the simulations, this can be reasoned with the increase in aura sizes for all objects. As aura size is increased it is more likely that object aura overlaps will occur, and with this that high-frequency message sending will be enacted while object interactions persist. Varying object aura sizes is then shown to have little impact on the volume of messages sent.

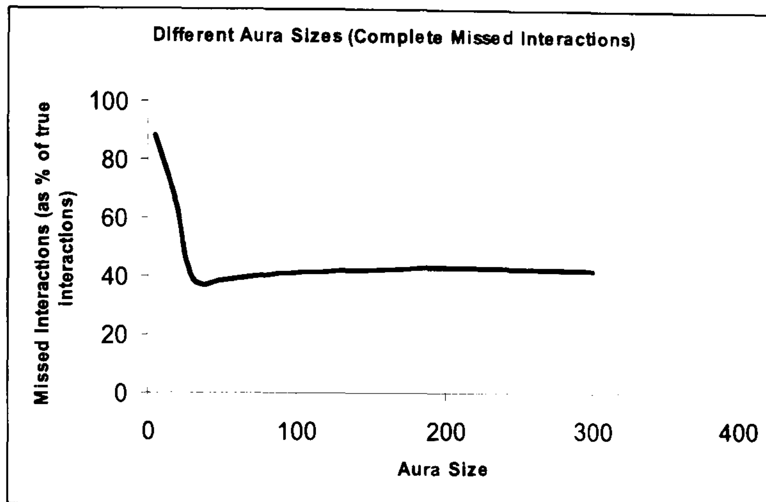


Figure 71 Quotient of complete missed interactions as function of aura size

Inspection of Figure 71 shows that the fewest number of missed interactions for the observed DVE configuration occurs when object aura size is just under 40 (at the bottom of the downward curve). The quotient of missed interactions at this point stands at just under 40%. As object aura size grows from this point there is only a minor rise (1% - 3%) in the occurrence of complete missed interactions. This may be explained by an increase in object aura overlaps for objects that briefly experience overlapping auras as they pass each other on their way towards different targets. The optimum object aura size discovered as a result of these tests is much smaller than the worst case estimate. Had the estimated aura size been used in practice it would have resulted in an increased regularity of missed interactions in the DVE. This discovery again validates the use of a DVE Simulator in determining optimum Interest Management configurations.

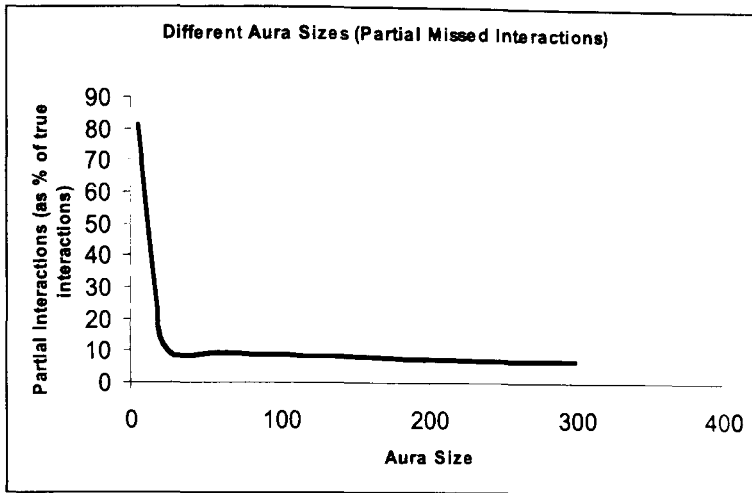


Figure 72 Quotient of partial missed interactions as function of aura size

Figure 72 depicts the number of partial missed interactions occurring across a range of object aura sizes. A similar curve can be seen to that of complete missed interactions (as in Figure 71), but in Figure 72 there is a small reduction in the quotient of partial interactions after the curve (as opposed to the small rise seen with complete missed interactions).

If a trade-off was sought between complete missed interactions and partial interactions, the optimum aura size for complete missed interactions of just under 40 (as found in Figure 71) would also provide near-optimal reduction of partial interactions.

5.3 Evolutionary Optimisation Simulator

A series of experiments was conducted to evaluate the Evolutionary Optimisation Simulator (EOS) component of the DVE simulation suite. The main focus of these experiments was to determine how variations in the evolutionary techniques that were employed would contribute to the derivation of optimum Interest Management parameter values. This would be measured by how chromosome fitness (see Section 4.3.2) across successive generations of simulation configurations is improved and maintained throughout a run of the EOS. In the majority of the experiments virtual world parameters have fixed values, allowing comparisons to be made between different evolutionary-algorithm configurations. The final experiment (Section 5.3.5)

tests different world configurations to ascertain if the EOS works as expected with different virtual world models.

All experiments are conducted three times and average results obtained.

5.3.1 Simulator Settings

The simulator was assessed using a base set of global configuration parameters:

- World size: 5000
- Number of iterations: 500
- Number of objects: 50
- Number of targets: 2
- Generation limit: 100
- Networking latency: 2
- Processing latency: 1

Fixed parameter values were applied to all object-class instances in all of the tests:

- Object class quotient: 25%
- Lower velocity bound: 10
- Upper velocity bound: 20
- High-frequency message interval: 5

A series of experiments was enacted to evaluate different aspects of the simulator:

- *Population Size (Section 5.3.2):* varying the population size to determine how it affects the capacity to retain promising solutions.

- *Mutation (Section 5.3.3):* varying the levels of mutation incorporated into the elite offspring, determining any effects upon the variety and fitness of subsequent candidate solutions.

- *High-Frequency Message Interval (Section 5.3.4):* varying the global high-frequency message interval to see how the quality of generated solutions is affected.

- *Different Scenarios (Section 5.3.5):* consideration of how the simulator solves the configuration problem for different DVE scenarios (wherein the base set of global parameters is changed).

A set of measurements was taken in each experiment to ascertain effectiveness of the simulator. This comprised the average fitness values and the variance of the fitness values across all of the chromosomes in each generation. These measurements were derived from the point when all chromosomes in a generation had been evaluated (through DVE simulation runs). The measurements would serve to show how the quality of the solution set changes over time, both in the level of quality achieved and also in the consistency of the candidate solution set. The focus is not only on the suitability of optimised solutions, but also how long it takes to derive improved solutions and how the quality of candidate solutions varies with each generation.

5.3.2 Population Size

The simulator was tested with a small set of different population sizes, specifically 10, 30, 50, 70 and 90 chromosomes. The results are shown in Figure 73 and Figure 74.

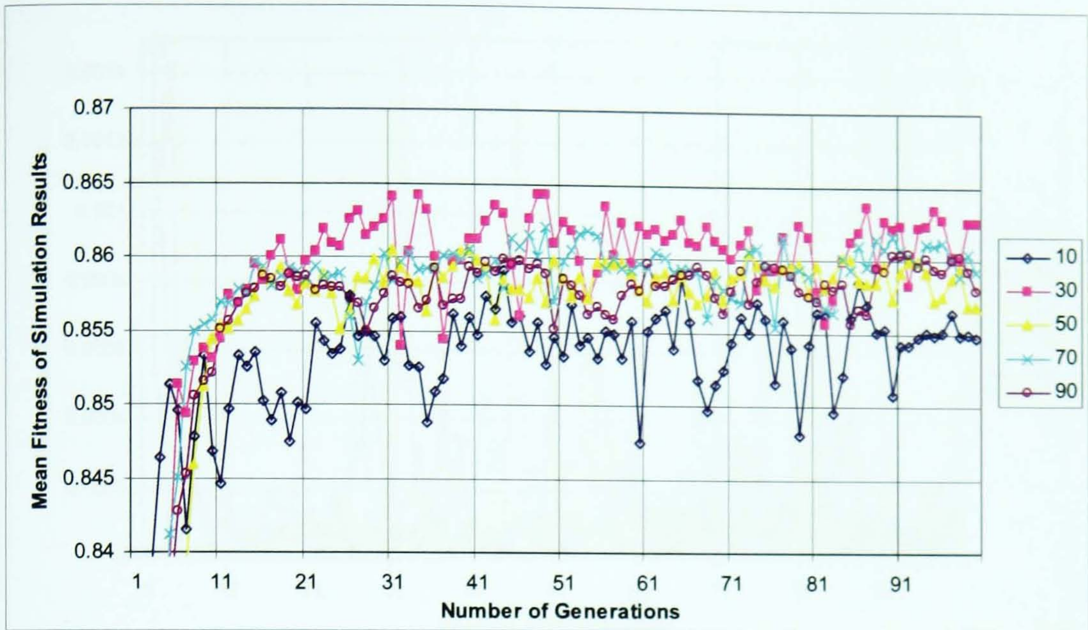


Figure 73 Chart of generations against average fitness for different population sizes

In Figure 73 for all population sizes the simulator quickly starts to converge on higher quality solutions and for the most part is able to pursue these promising solutions.

Referring to the simulated DVE, increasing the population size above 30 chromosomes does not affect the capacity to retain viable solutions as the average fitness remains relatively uniform in such instances. A low population size of 10 chromosomes shows frequent and sizeable fluctuations in the average fitness of the population, suggesting that a working population this small would be unsuitable for retaining promising candidate solutions.

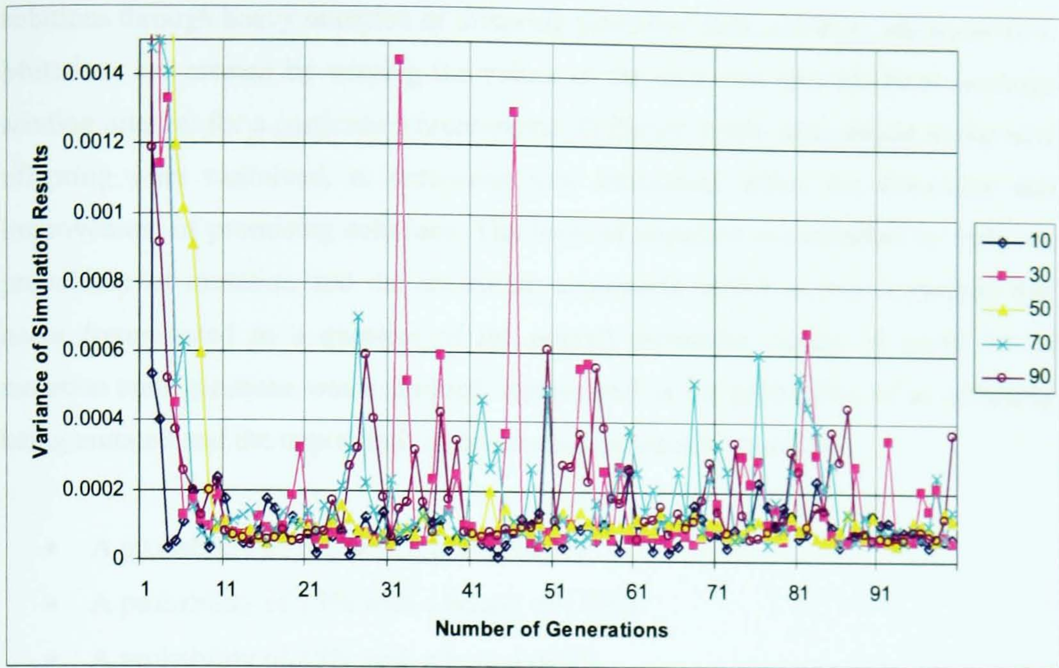


Figure 74 Chart of generations against variance for different population sizes

Figure 74 shows how the variance in the solution set is quickly reduced as each instance progresses. This illustrates how the simulator focuses on promising solutions to increase the quality of the population, maintaining this level of quality throughout the run. Results for all population sizes show infrequent spikes in variance, attributed to the simulator searching out new solution spaces using heavily mutated offspring. A population size of 30 shows greater but less frequent spikes than a population size of 70 for instance. Despite these spikes the general level of variance is kept constant, again proving that the simulator is able to refine the solution space and maintain improvements to the solution set.

A population size of 50 (the default value) appears to provide the most consistent results, with little variance in the simulation results as generations progress and chromosome fitness improves.

5.3.3 Mutation

Mutation affords examination of specific solution spaces by essentially fine-tuning the solution set with minor random mutations (in the case of offspring created from elite chromosomes). Mutation also provides a rapid means of evaluating unexplored

solutions through heavy mutation of offspring generated from non-elite chromosomes. Mutations are created by varying the values of the aura size and heartbeat message sending interval for a particular chromosome. Different levels of mutation in the elite offspring were examined, to determine how alterations affect the discovery and improvement of promising solutions. The level of mutation is controlled by both the probability of mutation and the maximum achievable extent of any mutations that occur (represented as a quotient of the overall parameter range). A small set of mutation configurations was examined, represented as the probability of an offspring being mutated and the upper limit of an enacted parameter mutation:

- A probability of 5% with a bound of 1%
- A probability of 15% with a bound of 1/75th
- A probability of 25% with a bound of 2%
- A probability of 40% with a bound of 5% (the default configuration)
- A probability of 50% with a bound of 10%

The results of these tests are shown in Figure 75 and Figure 76:

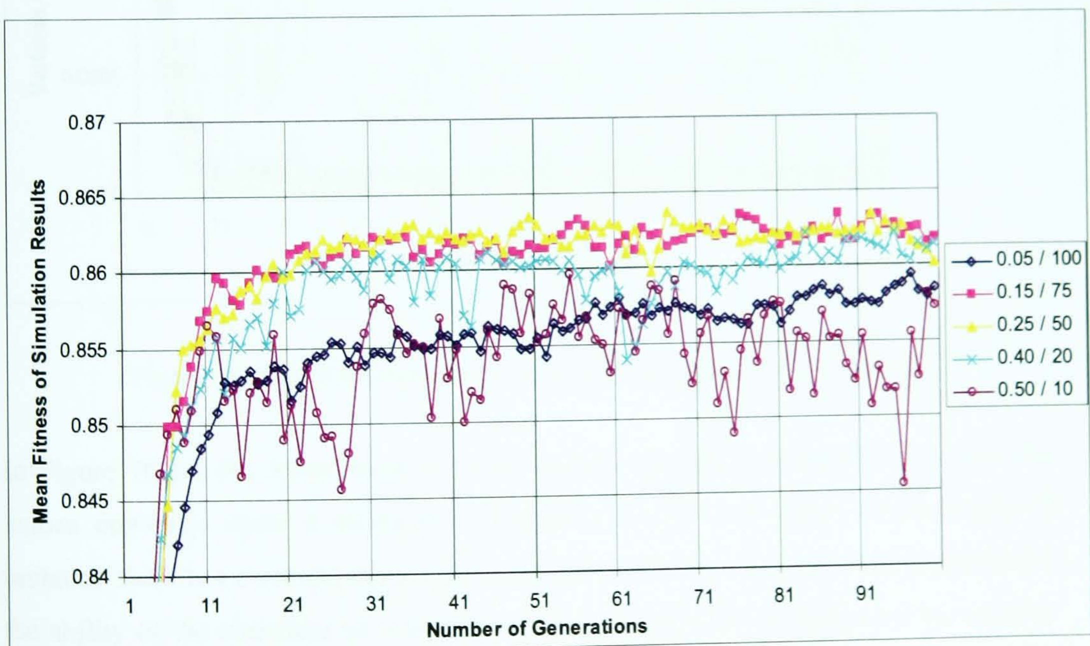


Figure 75 Chart of generations against average fitness for different mutation levels

In Figure 75 for a mutation level of 5% with a 1% bound the average fitness quickly converges on a set of promising solutions. The simulator is however relatively slow at fine-tuning the solution set and increasing the average fitness, albeit with no falls in overall quality. With the two most extreme levels of mutation it is evident that the solution set in general suffers lasting (and in some places severe) dips in fitness, affecting the ability to retain and tune solutions. When considering the DVE configuration observed within these tests, it appears that a chance of mutation of around 15-25%, with a bound of $1/75_{th}$-$1/50_{th}$ of the parameter range is the most suitable choice. Also, it appears that extremes of mutation (either too small or too large) generally restrict the discovery of optimised solutions and should not be used.

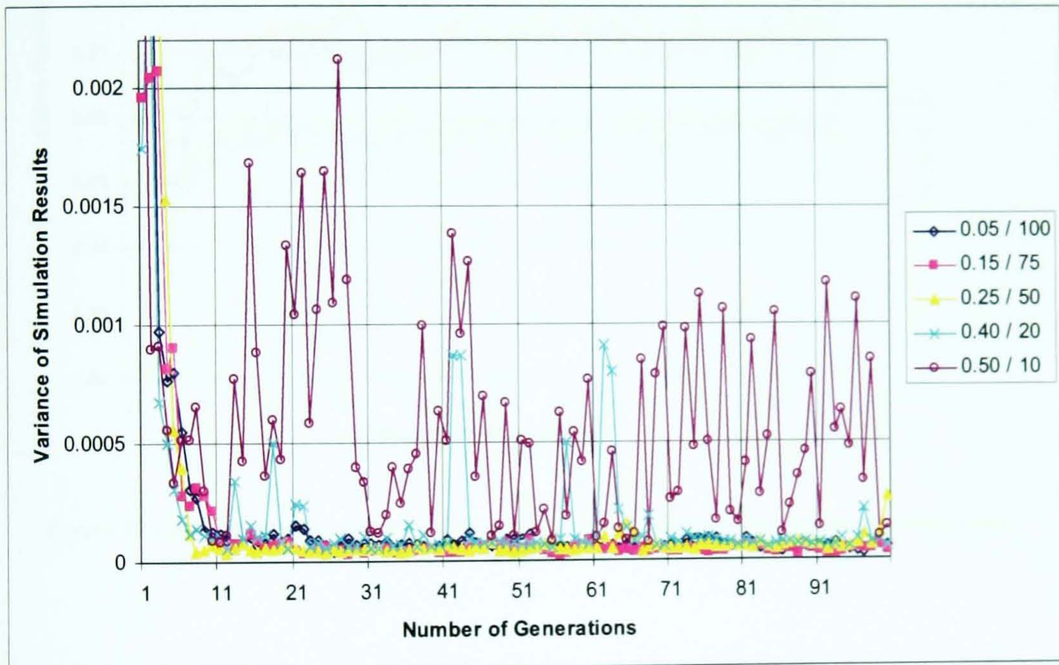


Figure 76 Chart of generations against variance for different mutation levels

In Figure 76 for the lesser three mutation levels variance is relatively low once the values converge upon a minimum. However for the two most severe levels of mutation there is a more pronounced fluctuation in fitness variance. This suggests that the ability of the simulator to retain and tune its solution set is hampered by extreme levels of mutation. As such it is recommended that only moderate to small levels of mutation are employed.

5.3.4 High-Frequency Message Interval

Experiments were conducted to test the capacity of the simulator to find optimum solutions given variability in one of the fitness function parameters (in this case the number of messages sent). DVEs were configured with the same base world parameters but with varying high-frequency messaging intervals. The specific test set ranged across high-frequency messaging intervals of 1 iteration, 2 iterations, 5 iterations, and 8 iterations.

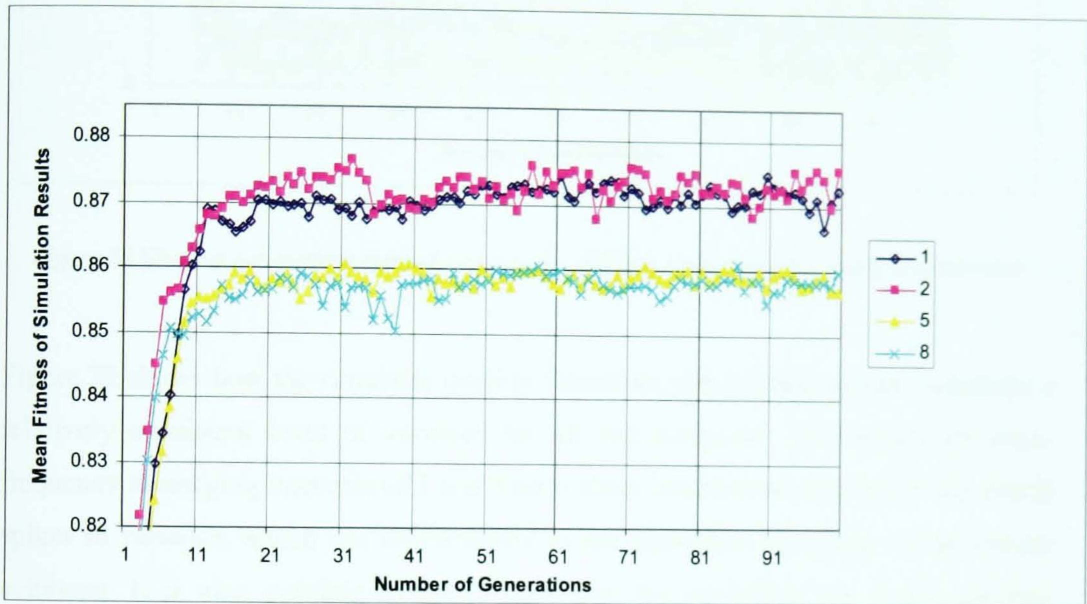


Figure 77 Chart of generations against average fitness for different high-frequency messaging intervals

From Figure 77 it is evident that the simulator can quickly converge on promising solutions across the different scenarios. The graph also indicates that the simulator is able to find more promising solutions as the high-frequency messaging interval is decreased. For messaging intervals of 1 and 2 iterations the average fitness is similar, and the same can be said of 5 and 8 iterations. For the smaller pair of values results suggest that the DVE configuration being observed is improved if the high-frequency message interval is decreased. These results suggest that it may be worthwhile in future to automatically evaluate different high-frequency messaging intervals during evolutionary optimisation of DVE configurations.

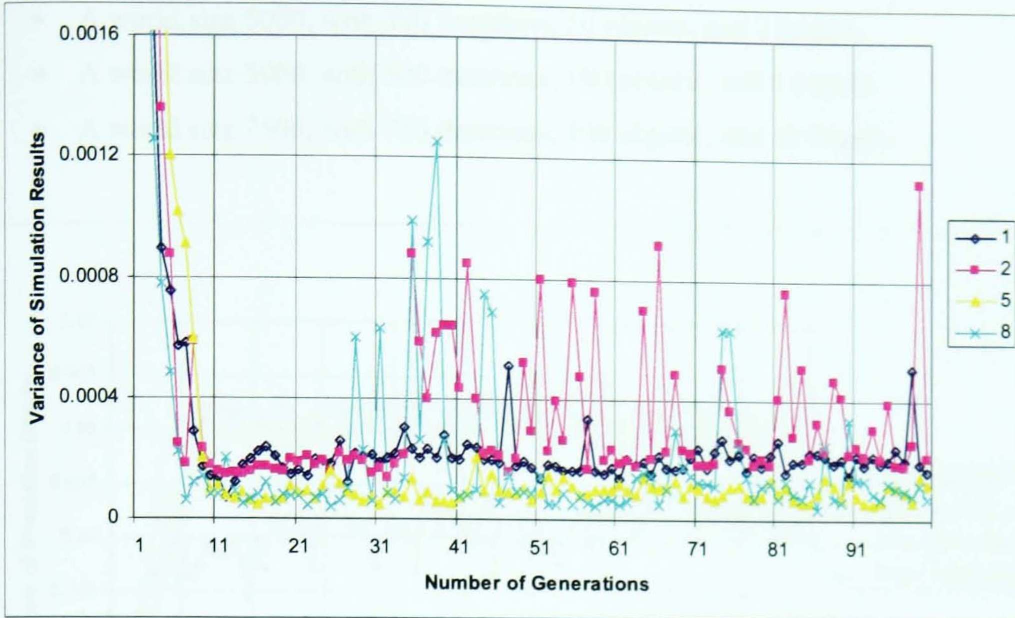


Figure 78 Chart of generations against variance for different high-frequency messaging intervals

Figure 78 shows how the simulator quickly fine-tunes the solution set and maintains a relatively consistent level of variance for all test instances. The results for high-frequency messaging intervals of 2 and 8 units show intermittent (but relatively small) spikes in variance, which can be attributed to the algorithm searching out promising solutions. It is also possible in these cases that the algorithm has found the best solutions that it can with the given DVE configuration early on in the algorithm runs, without them necessarily being optimal solutions. As such the algorithm would spend much of the time during the runs looking for new solution spaces through chromosome mutation. The variance measurements for 2 and 8 unit messaging intervals also correlate with the small deviations in associated average chromosome fitness as illustrated in Figure 77.

5.3.5 Different Scenarios

To test the simulator under varying DVE conditions different virtual world parameters were tested (i.e. world size, number of iterations, number of objects, and number of targets). The set of scenarios tested was as follows:

- A world size of 2500, with 500 iterations, 50 objects, and 10 targets.

- A world size 5000, with 500 iterations, 50 objects, and 2 targets.
- A world size 5000, with 500 iterations, 100 objects, and 5 targets.
- A world size 7500, with 750 iterations, 100 objects, and 10 targets.

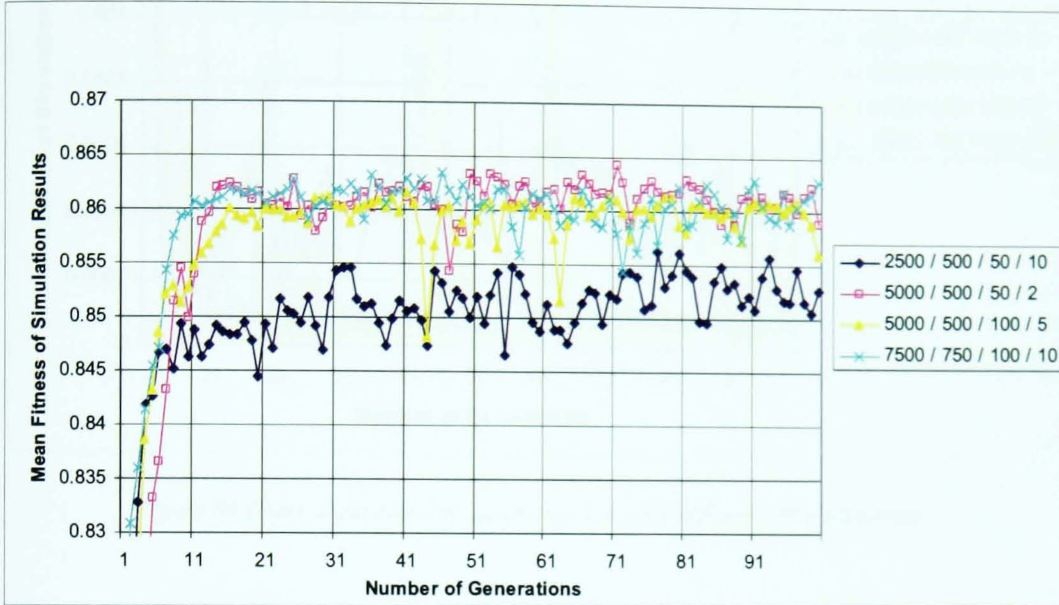


Figure 79 Chart of generations against average fitness for different DVE scenarios

Figure 79 shows that for a range of different DVE scenarios the simulator quickly converges on promising solutions and maintains a consistently high-quality set of candidate solutions. The theoretical maximum achievable fitness is a value of 1, and the fitness of the observed chromosomes is improved to around 0.86 for all but the smallest world. It could be speculated that alterations to the genetic algorithms or fitness criteria that were used could produce chromosomes with fitness values closer to 1.

The lower average fitness for the smallest world can be attributed to the fixed heartbeat message frequency interval, which is perhaps not low enough to accommodate the limitations of the environment (i.e. in reality such a DVE could be regarded as badly configured). There may be an increased chance of objects needing to interact as they would be in closer proximity to each other more often, resulting in more missed interactions. This shows that the EOS is capable of highlighting improper Interest Management configurations, just as it is able to find optimised ones.

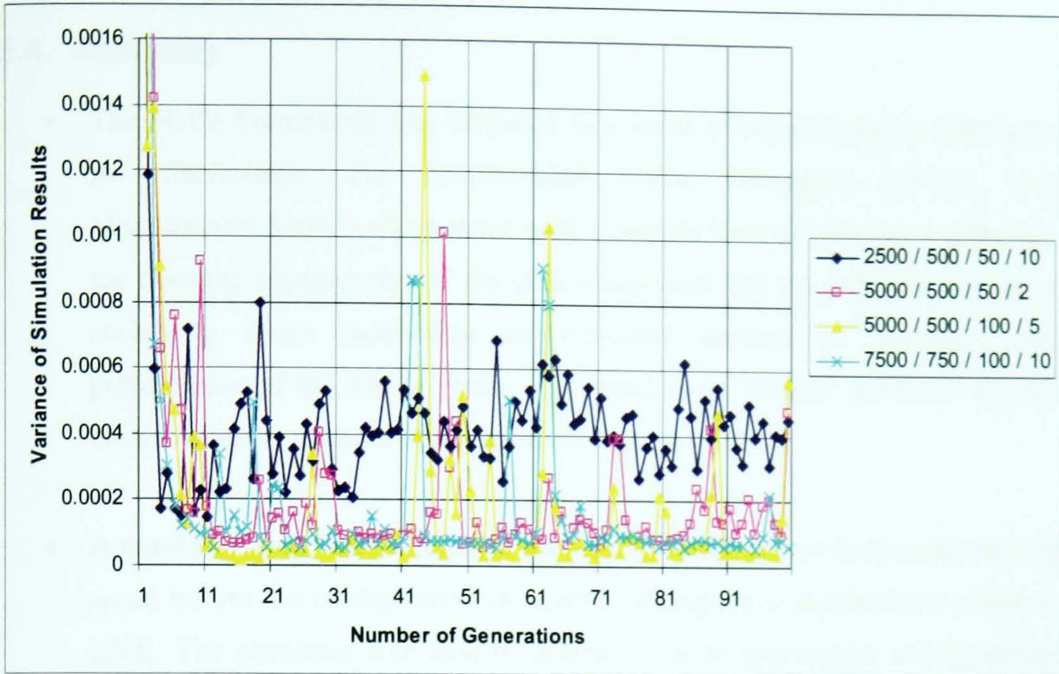


Figure 80 Chart of generations against variance for different DVE scenarios

In Figure 80 the variance quickly reduces, with a relatively low level maintained throughout each run. A small number of noticeable spikes are visible, perhaps attributable to the simulator upon occasion seeking out solutions elsewhere in the solution space which ultimately lack promise. The generally higher variance seen in the first scenario could be attributed to the lower average fitness achieved, which would prompt the algorithm to search out more promising solutions. If the simulator requires a sufficiently long time to determine an optimum solution it is more likely to test a greater range of candidate solutions. In all other test cases variance is kept low after approximately 10 generations, with only occasional spikes in variance as alternate solutions are sought.

5.4 Summary

- The MeCo Framework was subjected to a series of experiments to determine its effectiveness. The Provider-side MeCo, Messaging Service, and Measurement Service all operated with a notable level of efficiency, although the resource requirements of the GUI component had the potential to hinder scalability when monitoring an increased number of services. The performance of the MeCo Probe was found to be directly governed by the communication protocol of the observed service.

- A number of tests were conducted upon the DVE Simulator to ascertain how it could inform the configuration of Interest Management mechanisms within a DVE. The simulator was able to determine more appropriate configuration values than would have been discovered through the intuitive deliberation of a DVE application developer. That is to say, basic estimations of the sort that a DVE application developer may deduce are not necessarily sufficient for determining appropriate values for Interest Management parameters within a DVE. The DVE Simulator acts to provide performance-related information that is not readily determinable using estimation alone.

- A series of experiments was conducted with the Evolutionary Optimisation Simulator (EOS) to observe how aspects of the evolutionary optimisation process could be altered to optimise the discovery of optimal DVE configurations. The EOS tool was capable of rapidly improving a candidate solution set by retaining and fine-tuning promising solutions. At the same time the EOS tool demonstrated a capacity to search out previously unknown areas of the solution space for further investigation.

6. Conclusion

Through experimental evidence and reasoning it has been shown that the MeCo Framework and both the DVE Simulator and Evolutionary Optimisation Simulator (EOS) tools have made progress towards their respective aims. It is worth revisiting their achievements, while also looking beyond to how the work could be progressed in the future.

6.1 Thesis Summary

The work described in this thesis is aimed at providing heterogeneous QoS monitoring and SLA evaluation mechanisms for use in complex Internet service environments. With respect to the MeCo Framework there were a number of issues that needed consideration, as determined through examination of the existing technologies and related work:

- Distributed services do not all rely on the same communication technologies or application languages, and this must be taken into account when deploying a monitoring and evaluation framework. This is especially important in SLA-driven service environments where the behaviour and performance of individual service entities must be represented in a meaningful way.

- The QoS monitoring requirements of a service environment may be unique or may have attributes similar to existing services. There needs to be a way to re-use existing, proven monitoring and evaluation logic while also providing the capacity to dynamically introduce new capabilities where required.

- A prospective SLA monitoring and evaluation framework should cause as little disruption as possible to the service environment it is monitoring, in its deployment and in its continued operation and maintenance.

To address these issues the Metric Collector (MeCo) monitoring & evaluation framework [MorganIfip05] was developed, as described in Chapter 3. The framework consists of a series of components:

- *Provider-Side Metric Collector (MeCo):* monitors server-side performance i.e. what is happening inside the application server with respect to client behaviour. This allows observation of client adherence to SLA obligations.

- *Metric Collector (MeCo) Probe:* simulates client requests to the application server, as a means to determine how an observed service is performing in satisfying both client requests and the provider's own SLA obligations. In this way it is hoped that there is no need to directly monitor service clients.

- *Measurement Service:* gathers and evaluates measurement data against SLAs, using an internal Contract Engine.

- *Messaging Service:* manages transmission of measurement data from the Provider-Side MeCo to the Measurement Service, and from the Measurement Service to those parties interested in receiving violation notifications. The Messaging Service is built upon Message-Oriented Middleware (MOM).

The components of the MeCo Framework were developed to be generic. For instance the Provider-Side MeCo and MeCo Probe are capable of dynamically loading measurement classes for specialised per-service data-gathering. The MeCo Probe can also be configured to probe services built upon various communication protocols. The Measurement Service can be connected to any SLA evaluation engine with an exposed interface, and the backbone Messaging Service can be deployed across any Java-based Message-Oriented Middleware (MOM) technology that supports the publish/subscribe event notification paradigm. All that is required is that component implementations adhere to the appropriate interfaces. This adaptability means the MeCo Framework can be deployed to a wide variety of application environments, with the capacity to tailor monitoring and evaluation processes to the respective needs of a service relationship.

To further utilise the monitoring and evaluation constructs developed in the MeCo Framework, the same mechanisms were applied to another service domain other than E-Commerce. The same centralised approach to data evaluation was applied to Distributed Virtual Environments (DVEs) as described in Chapter 4.

Quantifiable measures for determining the performance of a DVE were developed in a centralised evaluation model, specifically the capabilities of DVE Interest Management mechanisms to detect missed object interactions. These performance criteria were incorporated into the DVE Simulator [Parkin06]. The simulator allowed a simulated DVE to be created and used to determine how best to configure the Interest Management mechanisms therein. Realistic modelling of participant activities in the DVE Simulator was achieved by simulating the behaviour of human crowding, in combination with a variety of object behaviour patterns. The DVE Simulator provides the capability to test and optimise Interest Management configurations while reducing the reliance upon the experience (i.e. ad-hoc estimations) of a DVE application developer.

As an extension of the DVE Simulator the Evolutionary Optimisation Simulator (EOS) [Parkin07] was created. The EOS is capable of automatically determining optimum values for Interest Management parameters for a particular DVE configuration by utilising genetic algorithm techniques. Through these techniques the EOS removes any need for human intervention or guesswork during the process of optimising Interest Management configurations.

The suite of DVE simulation tools is also novel in that it re-appropriates the centralised evaluation paradigm developed in the MeCo Framework not just for monitoring and evaluating simulated DVEs, but also to solve outstanding problems in the domain of DVEs.

6.2 Contribution of Thesis

Ideally monitoring and evaluation software for Internet applications would be suitable for use over a range of service domains. Monitoring software should also be able to automatically generate internal logic as required to monitor an observed service. This includes the ability to re-use existing monitoring code and accommodate new types of data measurements. Such features would save time and effort during development of a monitoring framework, and allow dynamic tailoring of the framework to meet the

needs of differing service environments. The deployment and maintenance of a monitoring framework should also be automated, so as to reduce the contributions required of both the administrator of the monitoring framework and those parties directly involved in the monitoring process.

The work described in this thesis has made some progress with these goals. The MeCo Framework (described in Chapter 3) interprets electronic service contracts to automatically generate monitoring and evaluation capabilities, as dictated by the requirements of each contract. These capabilities can also be automatically and dynamically deployed where required, for instance in the Provider-side MeCo (Section 3.3.3) and MeCo Probe (Section 3.3.4). Minimal preparation is required at the server-side for deployment, and no effort is required by service clients when deploying the framework. It is also possible to use new and existing monitoring logic across the MeCo Framework.

While investigating the monitoring of other domains an unexplored area was uncovered. This centred on the use of monitoring and evaluation techniques to better accommodate user influence and interest in a scalable Distributed Virtual Environment (DVE) in real-time. This work (described in Chapter 4) achieved some success. Evaluation techniques were used to provide a means of optimising DVE configurations where user influences are used to limit message production within the DVE. These techniques were incorporated into a suite of DVE simulation and optimisation tools. There is however still scope for further work in this area of research.

6.3 Future Work

A number of avenues for further work are evident based upon the achievements of the MeCo Framework, DVE Simulator and EOS:

- The MeCo Framework could be integrated with an existing DVE application in conjunction with recently developed scalability measures [Lu06, MorganAcm05]. The DVE evaluation logic that was developed within the DVE Simulator could be incorporated into the Measurement Service evaluation engine (see Section 3.3.6). This would require suitable Metric Collector (MeCo) implementations to be created to determine per-participant

performance in a shared DVE. The potential issues involved in this work are discussed in Section 4.4.

- Investigations into scalability measures in the MeCo Framework (see Section 3.3.6.5) could be revisited and extended. Further to this, the MeCo GUI component could be permanently decoupled from the MeCo Framework and other (less resource-intensive) graphical interface technologies employed. This would be worthwhile considering the findings of Section 5.1 with regards to the resource requirements of the MeCo GUI during extended use.

- The MeCo Framework uses informal names to refer to metric measurements (e.g. 'ejbResponseTime', 'soapClientThroughput'). To structure this convention, formalised metric mappings could be defined in a self-describing format such as can be achieved with the Resource Descriptor Framework (RDF) specification [Rdf] (see Sections 2.6.4.1 & 3.2.4).
  Metadata could be created to unambiguously describe the basic elements of QoS measurement. Examples include "the time between a request entering the network and it reaching the server processing stack" and "the number of requests received into the application server from clients during the previous minute". Protocol-specific metric definitions could also be described, such as "the length of time between an Enterprise JavaBean (EJB) request reaching the EJB container within the application server and the associated response leaving the EJB container". These building blocks could then be combined to describe composite metric measurements, clearly defining the applicability of measurements and which entities they refer to within the service environment.

- Although the Measurement Service allows monitoring of multiple service contracts, in its current state each Measurement Service instance can only communicate with a single Provider-side MeCo. There is an assumption that any other providers that are being monitored are managed from other Measurement Service instances. This could be remedied by extending the system to allow configuration of multiple provider connections from the Measurement Service (i.e. multiple MeCo MBean Communicator instances –

see Section 3.3.6). This would require more detailed remote-communication configurations to allow dynamic addition of multiple servers to the monitoring process as new contracts are added. A directory of known service providers would also have to be retained within each Measurement Service instance to enable communication of SLA measurement configurations when required.

- The MeCo Framework can adapt to new service contracts entering the system, configuring the various measurement and evaluation processes appropriately (as described in Section 3.3.6.2). The existing functionality could be extended to accommodate the modification of service contracts during active monitoring of the associated service environment. This may include complete contract termination, either as a part of the lifecycle of the service or as a result of disagreements relating to service provision.
  Alterations would need to be made to the configuration logic of the MeCo Framework (to dynamically update or disable measurement capabilities) and the internal contract-polling mechanisms (to detect changes in the set of monitored contracts).

- The DVE Simulator and EOS tools could be extended to provide richer simulated entity behaviour. This could include distinct behaviour patterns assignable to individual objects within a simulation, such as propensities to confront, avoid, accompany or ignore other objects. This would model social factors within group-based scenarios more closely. It may also be worthwhile to include the ability to describe a 'landscape' within a DVE simulation, which would influence how and where objects within the simulation move.

- The defining constructs of the MeCo Framework could be applied to other service domains such as Peer-to-Peer (P2P) applications (e.g. BitTorrent [Cohen03], Skype [Skype]), streamed-media applications [Lee05], or interactive media applications. Each service domain would have its own respective requirements and intricacies to consider, much as there was a need to consider the differences between E-Commerce applications and DVEs in applying techniques across these two domains.

- A visual component could be created for the DVE Simulator (and perhaps the EOS tool) to provide a three-dimensional representation of the movements and per-iteration behaviour traits of objects within a DVE simulation. The tools are already capable of recording per-simulation object trace information, so such an extension is feasible. A visual component would allow observation of entity behaviour in a simulated DVE. If coupled with configurable object behaviours and DVE landscapes, application developers would have the potential to model their applications more directly, and observe how their own application design decisions directly affect the performance of a DVE before deployment.

6.4 Summary

- The MeCo Framework (described in Chapter 3) made progress towards its goals of providing domain-agnostic, tailored and simple-to-deploy monitoring and evaluation capabilities for Internet applications.

- The DVE Simulator and Evolutionary Optimisation Simulator (EOS), both described in Chapter 4, provided monitoring and evaluation techniques to assist in the configuration of DVE Interest Management components. There is potential for a great deal of further research into the monitoring and evaluation of distributed virtual worlds.

- Suggested avenues for future work based on the MeCo Framework include the development of formalised metric mappings, the investigation of scalability features, and deployment of the framework within service domains such as peer-to-peer data-sharing applications.

- Potential future work based upon the DVE Simulator and EOS tools includes integration of the internal monitoring and evaluation logic into the MeCo Framework for deployment within a real DVE application. It is also possible that richer simulation content and visualisation components could be developed and integrated with the existing applications.

7. Bibliography

[Alonso04] Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju, "Web Services: Concepts, Architectures and Applications", Springer-Verlag, 2004

[Arjuna] Arjuna Technologies, "Arjuna Messaging Service", http://www.arjuna.com/products/arjunams/index.html, as viewed November 2004

[Asgari03] Abolghasem Asgari, Panos Trimintzios, Mark Irons, Richard Egan, George Pavlou, "Building Quality-of-Service Monitoring Systems for Traffic Engineering and Service Management", Journal of Network and Systems Management, Vol. 11, No. 4, 2003

[Aurrecoechea96] Cristina Aurrecoechea, Andrew T. Campbell, Linda Hauw, "A Survey of QoS Architectures", Multimedia Systems, 1996

[Axis] Apache Software Foundation, "Apache Axis Toolkit", http://ws.apache.org/axis/, as viewed 22/09/06

[Banavar99] Guruduth Banavar, Tushar Chandra, Robert Strom, Daniel Sturman, "A Case for Message Oriented Middleware" , in Proceedings of the 13th International Symposium on Distributed Computing, Pgs. 1-18, 1999

[Bharambe02] A.R. Bharambe, S. Rao, S. Seshan, "Mercury: A Scalable Publish-Subscribe System for Internet Games", In Proceedings of the 1s Workshop on Network and System Support for Games, ACM Press, p. 3-9, 2002

[Brocklehurst05] D. Brocklehurst, D. Bouchlaghem, D. Pitfield, G. Palmer, K. Still, "Crowd circulation and stadium design: low flow rate systems", Structures & Buildings, (2005), Volume 158, Issue 5

[Carzaniga00] Antonio Carzaniga, David S Rosenblum, Alexander L. Wolf, "Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service", Symposium on Principles of Distributed Computing, 2000

[Carzaniga01] Antonio Carzaniga, David S Rosenblum, Alexander L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service", ACM Transactions on Computer Systems, Vol. 19, No. 3, August 2001, Pages 332–383

[Chan00] Mun Choon Chan, Yow-Jian Lin, Xin Wang, "A Scalable Monitoring Approach for Service Level Agreements Validation", Proceedings of the 2000 International Conference on Network Protocols, 2000

[Chen98] T. M. Chen, S. S. Liu, M. J. Procanik, D. C. Wang, D. D. Casey, INQIRE: A Software Approach to Monitoring QoS in ATM Networks, IEEE Networks, Volume 12 Issue 2, 1998

[Cohen03] Bram Cohen, "Incentives Build Robustness in BitTorrent". In Workshop on Economics of Peer-to-Peer Systems, 2003

[Colouris01] George Coulouris, Jean Dollimore, Tim Kindberg, "Distributed Systems: Concepts and Design (Third Edition)", Addison-Wesley, 2001

[Corba] Object Management Group Inc., "Common Object Request Broker Architecture (CORBA)", http://www.corba.org, as viewed 22/09/06

[CorbaNS] Object Management Group Inc., "CORBA Notification Service", http://www.omg.org/technology/documents/formal/notification_service.htm, as viewed 22/09/06

[Cukier98] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W.H. Sanders, D.E. Bakken, M. Berman, D.A. Karr, R.E. Schantz, "AQuA: An Adaptive Architecture That Provides Dependable Distributed Objects", in Proc. Of the 17th IEEE Symposium on reliable Distributed Systems, p. 245-253, Oct. 1998

[Curry04] Edward Curry, Desmond Chambers, Gerard Lyons, "Extending Message-Oriented Middleware using Interception", International Workshop on Distributed Event-based Systems (DEBS 2004) W18L Workshop - 26th International Conference on Software Engineering (2004/918), p. 32 -37

[Darkstar] Sun Microsystems Inc., "Project Darkstar Overview", http://research.sun.com/projects/dashboard.php?id=168, as viewed 01/06/07

[Debusmann03] Markus Debusmann, Alexander Keller, "SLA-Driven Management of Distributed Systems Using the Common Information Model", in Proceedings of the 8th IFIP/IEEE IM, 2003

[Diffserv] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, "An Architecture for Differentiated Services", http://tools.ietf.org/html/rfc2475, December 1998, as viewed 22/09/06

[Dilman01] Mark Dilman, Danny Raz, "Efficient Reactive Monitoring", INFOCOM, 2001

[Eclipsecolorer] John-Mason P. Shackleford, Konstantin Scheglov, Eduardo Perez Ureta, "Eclipse Profiler Plugin", http://www.sourceforge.net/projects/eclipsecolorer, as viewed 22/09/06

[Ejb] Sun Microsystems Inc., "Enterprise JavaBeans Technology", http://java.sun.com/products/ejb/, as viewed 22/09/06

[Eugster00] Patrick Th. Eugster, Rachid Guerraoui, Joe Sventek, "Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction", ECOOP 2000, LNCS 1850, pp. 252{276, Springer-Verlag Berlin Heidelberg 2000

[Eugster03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, Anne-Marie Kermarrac, "The Many Faces of Publish/Subscribe", ACM Computing Surveys (CSUR), Volume 35, Issue 2 (June 2003), Pages: 114 – 131, 2003

[Fogel94] D. B. Fogel, "An Introduction to Simulated Evolutionary Optimization", IEEE Transactions on Neural Networks: Special Issue on Evolutionary Computation, Vol. 5, No. 1, pp. 3-14, 1994

[Forouzan03] Behrouz A. Forouzan, "TCP/IP Protocol Suite", 2nd Edition, McGraw-Hill, 2003

[Gore01] P. Gore, R. Cytron, D. Schmidt, C. O'Ryan, "Designing and Optimizing a Scalable CORBA Notification Service", in Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, p. 196-204, 2001

[Greenhalgh95] C. Greenhalgh, S. Benford, "MASSIVE: a distributed virtual reality system incorporating spatial trading", Proceedings IEEE 15th International Conference on distributed computing systems (DCS 95), Vancouver, Canada, (1995)

[Halflife] Wikimedia Foundation Inc., "Half Life [Valve Software] Wikipedia Page", http://en.wikipedia.org/wiki/Half-Life, as viewed 22/09/06

[Haefal01] Richard Monson-Haefal & David A. Chappell, "Java Message Service", O'Reilly, 2001

[He01] Jun He, Mohan Rajagopalan, Matti A. Hitunen, Richard D. Schlichting, "Providing QoS Customization in Distributed Object Systems", Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, 2001

[Helbing05] D. Helbing, L. Buzna, A. Johansson, T. Werner, "Self-Organized Pedestrian Crowd Dynamics: Experiments, Simulations, and Design Solutions", Transportation Science, Vol. 39, No. 1, (2005), pp. 1–24

[Hsiao05] Tsun-Yu Hsiao, Shyan-Ming Yuan, "Practical Middleware for Massively Multiplayer Online Games", IEEE Internet Computing, Volume 9, Issue 5 (September 2005). Pgs 47 – 54, 2005

[Ince02] Darrell Ince, "Developing Distributed and E-Commerce Applications", Addison-Wesley, 2002

[Javakeystore] Sun Microsystems Inc., "Java Keystore", http://java.sun.com/j2se/1.4.2/docs/api/java/security/KeyStore.html, as viewed 22/09/06

[Javarmi] Sun Microsystems Inc., "Java Remote Method Invocation (RMI)", http://java.sun.com/products/jdk/rmi/, as viewed 22/09/06

[Jboss] Red Hat Middleware LLC, "JBoss", http://www.jboss.com/, as viewed 22/09/06

[JbossMQ] Red Hat Middleware LLC, "JBossMQ", http://www.jboss.org/wiki/Wiki.jsp?page=JBossMQ, as viewed 22/09/06

[JbossMessaging] Red Hat Middleware LLC, "JBossMQ", http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossMessaging, as viewed 22/09/06

[Jbossjmx] Red Hat Middleware LLC, "JBoss JMX Console", http://wiki.jboss.org/wiki/Wiki.jsp?page=JMXConsole, as viewed 22/09/06

[Jbossxmbean] Red Hat Middleware LLC, "JBoss XMBean", http://wiki.jboss.org/wiki/Wiki.jsp?page=XMBean, as viewed 22/09/06

[Jfreechart] Object Refinery Limited, "JFreeChart", http://www.jfree.org/jfreechart, as viewed 22/09/06

[Jiang00] Yuming Jiang, Chen-Khong Tham, Chi-Chung Ko, "Challenges and Approaches in Providing QoS Monitoring", International Journal of Network Management, Pg. 323-334, 2000

[Jimenez04] Carlos Molina-Jimenez, Santosh Shrivastava, Jon Crowcroft, Panos Gevros, "On the Monitoring of Contractual Service Level Agreements", In Proceedings of the IEEE Conference on Electronic Commerce CEC\04, San Diego, 2004

[Jmeter] Apache Software Foundation, "Apache JMeter", http://jakarta.apache.org/jmeter/, as viewed 22/09/06

[Jms] Sun Microsystems Inc., "Java Message Service", http://java.sun.com/products/jms, as viewed 22/09/06

[Jmx] Sun Microsystems Inc., "Java Management Extensions (JMX)", http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/, as viewed 22/09/06

[Jsp] Sun Microsystems Inc., "JavaServer Pages Technology", http://java.sun.com/products/jsp/, as viewed 22/09/06

[J2ee] Sun Microsystems Inc., "Java 2 Enterprise Edition", http://java.sun.com/javaee/, as viewed 22/09/06

[J2eeservlet] Sun Microsystems Inc., "Java 2 Servlet Technology", http://java.sun.com/products/servlet, as viewed 22/09/06

[J2se] Sun Microsystems Inc., "Java 2 Standard Edition", http://java.sun.com/javase, as viewed 22/09/06

[Kabus05] Patric Kabus, Wesley Terpstra, Mariano Cilia, Alejandro Buchmann, "Addressing Cheating in Distributed Massively Multiplayer Online Games", International Workshop on NetGames (NetGames'05), NY, October 2005

[Kakadia01] Deepak Kakadia, "Tech Concepts: Enterprise QoS Policy Based Systems & Network Management", Sun Microsystems, 2001

[Karr97] C. R. Karr, D. Reece, R. Franceschini, "Synthetic Soldiers" IEEE Spectrum, p. 39-45, March 1997

[KellerLisa02] Alexander Keller, Heiko Ludwig, "Defining and Monitoring Service Level Agreements for Dynamic E-Business", LISA 2002

[KellerIbm02] Alexander Keller, Heiko Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services", IBM Research Report, 2002

[Kim01] Joong-Han Kim, R.S. Ramakrishna, Yoo-Sung Kim, "LODIN: Load Distribution Mechanism in CORBA Using Interceptor", IEEE International Conference on Electrical and Electronic Technology (TENCON), 2001

[Koster00] Rainer Koster, Thorsten Kramp, "Structuring QoS-Supporting Services with Smart Proxies", Proceedings of the IFIP/ACM Middleware Conference (Middleware), 2000

[Lee05] Jack Y. B. Lee, "Scalable Continuous Media Streaming Systems: Architecture, Design, Analysis and Implementation", John Wiley & Sons, 2005

[Little99] M. C. Little, S. K. Shrivastava, "A method for combining replication with cacheing", International Workshop on Reliable Middleware Systems, 1999

[Lu06] Fengyun Lu, Simon Parkin, Graham Morgan, "Load Balancing for Massively Multiplayer Online Games", Proceedings of the ACM-SIGCHI 5th Workshop on Network & System Support for Games, 2006

[Ludwig02] H. Ludwig, A. Keller, A. Dan, R. King, "A Service Level Agreement Language for Dynamic Electronic Services", in Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2002), 2002

[Macedonia94] M.R. Macedonia, M.J. Zyda, D.R. Pratt, P.T. Barham, S. Zeswitz, "NPSNET: A Network Software Architecture for Large Scale Virtual Environments", MIT Presence 3(4), 1994

[Mani02] Anbazhagan Mani, Arun Nagarajan, "Understanding Quality of Service for Web Services", IBM DeveloperWorks, 2002

[Mastaglio05] T.W. Mastaglio, R. Callahan, "A Large-Scale Complex Virtual Environment for Team Training," Computer, p. 49-56, July 1995

[Merabti04] Madjid Merabti, Abdennour El Rhalibi, "Peer-to-Peer Architecture and Protocol for a Massively Multiplayer Online Game", IEEE Communications Society Globecom 2004 Workshops, 2004

[Microsoftmsg] Microsoft Corporation, "MSN Messenger", http://messenger.msn.com, as viewed 16/11/05

[MorganAcm05] G. Morgan, F. Lu, K. Storey, "Interest Management Middleware for Networked Games", Proceedings of the I3D 2005. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, Washington, DC, April 3-6, 2005 pp. 57-63 ACM SIGGRAPH 2005

[MorganIfip05] G. Morgan, S. Parkin, C. Molina-Jimenez, J. Skene, "Monitoring Middleware for Service Level Agreements in Heterogeneous Environments", In the proc. of the fifth IFIP conference on e-Commerce, e-Business, and e-Government (I3E 2005), October 26-28, (2005), IFIP Volume 189 pp. 79-93

[Msmq] Microsoft Corporation, "Microsoft Message Queuing", http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx, as viewed 22/09/06

[Muller99] Nathan J. Muller, "Managing Service Level Agreements", International Journal of Network Management, Pg. 155-166, 1999

[Mysql] MySQL AB, "MySQL", http://www.mysql.com/, as viewed 22/09/06

[Nahrstedt01] Klara Nahrstedt, Dongyan Xu, Duangdao Wichadakul, Baochun Li, "QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments", IEEE Communications Magazine, Volume 39 Issue 11, 2001

[Narasimhan99] Priya Narasimhan, Louise E. Moser, P.M. Melliar-Smith, "Using Interceptors to Enhance CORBA". IEEE Computer Magazine, p. 62-68, 1999

[Nutt00] Gary J. Nutt, Scott Brandt, Adam J. Griff, Sam Siewert, Marty Humphrey, Toby Berk, "Dynamically Negotiated Resource Management for Virtual Environment Applications," IEEE Transactions on Knowledge and Data Engineering, Vol. 12, No. 1, January/February 2000, pp. 78-95.

[Overton02] Chris Overton, "On the Theory and Practice of Internet SLAs", Journal of Computer Resource Measurement 106, 32–45, Computer Measurement Group (April 2002)

[Parkin06] S. E. Parkin, G. Morgan, "Managing Missed Interactions in Distributed Virtual Environments", EUROGRAPHICS Symposium on Virtual Environments (2006), 2006

[Parkin07] Simon Parkin, Peter Andras, Graham Morgan, "Evolutionary Optimization of Parameters for Distributed Virtual Environments", IEEE Congress on Evolutionary Computation (CEC 2007), 2007

[Pruyne00] Jim Pruyne, "Enabling QoS via Interception in Middleware", HP-Labs Report HPL-2000-29, February 2000

[Qosforum99] QoS Forum, "QoS Protocols & Architectures – White Paper", Quality of Service Forum, 1999

[Rdf] W3C, "RDF/XML Syntax Specification", http://www.w3.org/TR/rdf-syntax-grammar, as viewed 22/09/06

[Rsswiki] Wikimedia Foundation Inc., "RSS Wikipedia Entry", http://en.wikipedia.org/wiki/RSS_(file_format), as viewed 22/09/06

[Rsvp] R. Braden, L. Zhang, S. Berson, S. Herzog, S. Jamin. "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification", http://tools.ietf.org/html/rfc2205, September 1997, as viewed 22/09/06

[Sahai01] Akhil Sahai, Anna Durante, Vijay Machiraju, "Towards Automated SLA Monitoring for Web Service", Research Report HPL-2001-310 (R.1), Hewlett-Packard Laboratories, Palo Alto, 2001

[Sahai02] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Li Jie Jin, Fabio Casati, "Automated SLA Monitoring for Web Services", Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications, 2002

[Schantz99] Richard Schantz, John Zinky, David Karr, David Bakken, James Megquier, Joseph Loyall, "An Object-Level Gateway Supporting Integrated-Property Quality of Service", ISORC '99, 1999

[Secondlife] Linden Research Inc., "Second Life", http://www.secondlife.com/, as viewed 22/09/06

[Segall00] Bill Segall, David Arnold, Julian Boot, Michael Henderson, Ted Phelps, "Content Based Routing with Elvin4", In Proc. AUUG2K, June 2000

[Singhal99] S. Singhal, M. Zyda, "Networked Virtual Environments, Design and Implementation", Addison Wesley, (1999)

[Skene03] J. Skene and W. Emmerich (2003), "Model Driven Performance Analysis of Enterprise Information Systems", Electronic Notes in Theoretical Computer Science, 82(6), 2003

[Skene04] James Skene, Davide Lamanna, Wolfgang Emmerich, "Precise Service Level Agreements", Proceedings of the 26th International Conference on Software Engineering, 2004

[SkeneEdoc04] James Skene, Wolfgang Emmerich, "Generating a Contract Checker for an SLA Language", in Proc. of the EDOC 2004 Workshop on Contract Architectures and Languages, Monterey, California, IEEE Computer Society Press, 2004

[Skype] Skype Limited, "Skype", http://www.skype.com/intl/en-gb/, as viewed 22/09/06

[Snell02] James Snell, Doug Tidwell, Pavell Kulchenko, "Programming Web Services With SOAP", O'Reilly, 2002

[Soap] DevelopMentor, International Business Machines Corporation, Lotus Development Corporation, Microsoft, UserLand Software, "Simple Object Access Protocol (SOAP) 1.1", http://www.w3.org/TR/SOAP/, as viewed 22/09/06

[Srinivasan95] R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2", Sun Microsystems Internet RFC 1831, August 1995

[Sweeney99] T. Sweeney, "Unreal Networking Architecture", http://unreal.epicgames.com/Network.htm, 1999, as viewed 09/2005

[Tanenbaum02] Andrew S Tanenbaum, Maarten van Steen, "Distributed Systems: Principles and Paradigms", Prentice-Hall, 2002

[Tapas] Trusted and QoS-Aware Provision of Application Services, IST Project No: IST-2001-34069, http://tapas.sourceforge.net, as viewed 09/2005

[Tibco] TIBCO Software Inc., "TIBCO Rendezvous", http://www.tibco.com/software/messaging/rendezvous/default.jsp, as viewed 22/09/06

[Unreal] Epic Games Inc, "Unreal Tournament", http://www.unrealtournament.com, as viewed 16/11/05

[WebsphereMQ] International Business Machines Corp., "WebSphere MQ", http://www-306.ibm.com/software/integration/wmq/, as viewed 22/09/06

[Welsh01] Matt Welsh, David Culler, Eric Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services", 18th Symposium on Operating Systems Principles (SOSP-18), Chateau Lake Louise, Canada, October 21-24, 2001

[Wilson01] S. Wilson, H. Sayers, M.D.J. McNeill, "Using CORBA Middleware to Support the Development of Distributed Virtual Environment Applications", in Proceedings of WSCG Conference, 2001

[Wsfl] International Business Machines Corp., "IBM Service Oriented Architecture (SOA)", http://www-306.ibm.com/software/solutions/soa/, as viewed 22/09/06

[Wow] Blizzard Entertainment Inc., "World of Warcraft", http://www.worldofwarcraft.com, as viewed 16/11/05

[Wsdl] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, "Web Services Definition Language (WSDL)", http://www.w3.org/TR/wsdl, as viewed 22/09/06

[Wsif] Apache Software Foundation, "Web Services Invocation Framework" (WSIF), http://ws.apache.org/wsif/, as viewed 22/09/06

[Xml] World Wide Web Consortium (W3C), "Extensible Markup Language (XML)", http://www.w3.org/XML/, as viewed 22/09/06

[Yahoomsg] Yahoo! Inc., "Yahoo! Messenger", http://messenger.yahoo.com, as viewed 16/11/05

[Zhang93] L. Zhang, S. Deering, D. Estrin, S. Shenker, D. Zappala, "RSVP: a new resource reservation protocol", IEEE Network, Sep 1993, p. 8-18

[Zyda91] M. J. Zyda, D. R. Pratt, "NPSNET: A 3D visual simulator for virtual world exploration and experience", In Tomorrow's Realities Gallery, Visual Proceedings of SIGGRAPH 91, (1991) p. 30

8. Appendix A – MeCo Installation Guide

8.1 Provider-Side MeCo Deployment

To allow MeCo Interceptors to collect observation data on behalf of the Metric-Collector (MeCo) Framework a number of JBoss configuration files must be altered, and a small set of additional files added to the JBoss file system. These steps are described here (with example configuration files listed in Section 8.4).

8.1.1 Apache Axis Configuration

Note: This file alteration is only applicable when enabling SOAP MeCo Interceptors

Alter *WEB-INF\server-config.wsdd* within the SOAP service file-set to include `<request-flow>` and `<response-flow>` element definitions. These declarations should include the `uk.ac.ncl.cs.meco.interceptors.SOAPMecoInterceptor` handler class.
A further optional step is to alter *WEB-INF\web.xml* to include a `<listener>` declaration that triggers pre-loading of the SOAP Interceptor within the JBoss server. The `uk.ac.ncl.cs.meco.server.listeners.MecoContextListener` class should be included in this reference. If this change is to be made ensure that the DOCTYPE of the file references Version 2.3 of the servlet specification.

8.1.2 JBoss Configuration

- *jbossmq-destinations.xml* (typically found in \deploy\jms)

Note: This file alteration is only applicable if the JBoss server is acting as the JMS server for the MeCo Framework

Include the names of the JMS topics required for metric transmission. The topic names should be in the format <SLA_ID>_<operation_name>, where <operation_name> represents a valid contract operation.

If the JBoss instance is running the messaging server used to transmit SLA violations a topic must be created for each SLA, of the format <SLA_ID>.

- *standard-jboss.xml* (\conf)

Note: This file alteration is only applicable when enabling EJB MeCo Interceptors

Add a reference to the `uk.ac.ncl.cs.meco.ejb.interceptors.ClientIPInterceptor` class anywhere within the `<bean>` interceptor stack definitions for both the 'stateless-rmi-invoker' and 'clustered-stateless-rmi-invoker' `<invoker-proxy-binding>` elements. Also add a reference to the `uk.ac.ncl.cs.meco.interceptors.EJBMecoInterceptor` class anywhere in the interceptor stack for the 'Standard Stateless SessionBean' and 'Clustered Stateless SessionBean' `<container-configuration>` elements.

- *meco-management-bean.sar*

Deploy this file to the 'deploy' folder to enable management of the Provider-side MeCo from the JBoss JMX Console and the Measurement Service.

Note: once the MeCo is deployed the 'ProviderID' parameter MUST be set for the MeCo XMBean (from the JMX Console) before activating the Measurement Service. This is necessary to identify the service provider during SLA evaluation – the MeCo system will not function if this is not configured

- *Supplementary Class Libraries*

Copy the *meco-core.jar* and *meco-dynamic.jar* files to the \lib sub-directory.

8.2 Measurement Service Deployment

The Measurement Service collects data transmitted by the MeCo Interceptors and evaluates it against the relevant SLAs. Data charts and tabular records are presented, detailing metric performance, transmitted data (and associated message parameters) and details of any SLA violations that occur during the monitoring of a service.

8.2.1 Measurement Service Installation Files

- *meco-meas.zip*

  Contains the Measurement Service and required class libraries. This package can be deployed to any directory either on the same machine as the supporting JBoss server or at a remote location. When deployed, supplementary library files are extracted to a `\lib` sub-directory, with the Measurement Service class and configuration files extracted to the base directory.

- *meco-measurement-service.jar*

  Contains the necessary class files for running the Measurement Service and is deployed as part of the *meco-meas.zip* package.

- *measurement-service.xml*

  This XML file describes configuration details for the Measurement Service. Ensure that the `<contract>` sub-element of the `<SLAng>` configuration element points to the directory that contains the contract file (see Section 8.2.2).
  The `<log>` sub-element of the `<SLAng>` configuration element is useful only for debugging the internal SLAng checker and is not defined by default.
  The `<remoteConfiguration>` parameters must be calibrated to allow configuration of the MeCo Interceptors from the Measurement Service. Also ensure that the `<JMS>` configuration allows the Measurement Service to use the same JMS server as the MeCo Interceptors.

The <probe> element should be configured to point to the directory that is to be monitored for probe descriptors. The <chartDir> sub-element of the <gui> configuration should point to a location for storing chart snapshots captured during the use of the Measurement Service.

- *measurement-service.xsd*

This is the XML schema for the Measurement Service configuration file. By default this file is co-located with *measurement-service.xml* in the base directory of the Measurement Service. If moved from this location, the xsi:noNamespaceSchemaLocation attribute in *measurement-service.xml* should be changed accordingly.

8.2.2 Additional Files

- *Probe Configurations*

Note: If the monitored application is moved to another machine, the WSDL probe configuration files will have to be updated to correctly reference the appropriate host machine

Ensure that any additional application-specific class library files required by a probe component are copied into the \lib sub-directory of the Measurement Service installation.

Each Probe Configuration contains pre-set parameter values to serve as arguments to the target service (detailed in the <mprobe:probeMethodInfo> element). Ensure that if the target service performs data-lookup operations (such as a database record search) that probe values reference existing data records within the underlying data store(s) of the observed service.

- *Probe Descriptors*

Note: Before a contract is loaded into the system, the associated probe descriptor file MUST be placed into the appropriate observed directory

A Probe Descriptor is associated with at least one contract and must be placed in the designated probe-descriptor folder in order to be detected.

- *Contract Files*

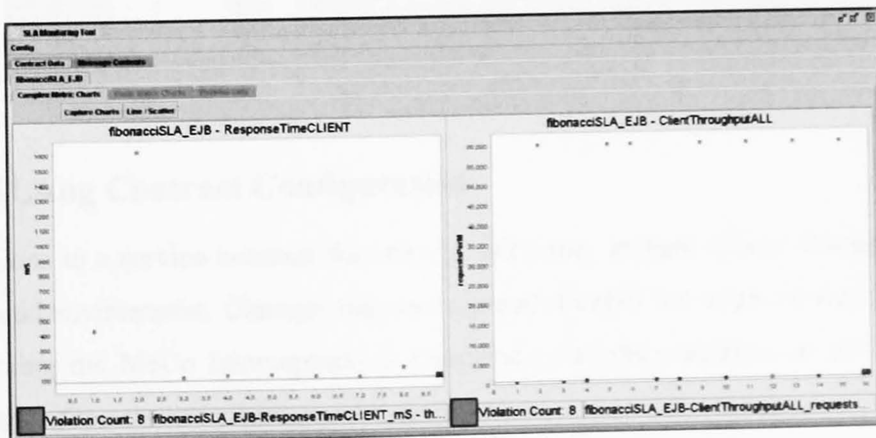
Contract files must be accessible from the Measurement Service (with their location referenced in the *measurement-service.xml* configuration file).

8.2.3 Using the Measurement Service

- *meas.bat / meas.sh*

To start the Measurement Service execute the appropriate file depending on whether a Windows (*.bat*) or Linux (*.sh*) environment is being used. Ensure that the JMS server being used is running before activating the Measurement Service otherwise it will fail to connect to the measurement update topics.

- *Graphical User Interface*



The user interface presents measurement data in a number of formats. These include 'complex' charts of measurements determined inside the Measurement Service, and charts of basic measurement data received from the Provider-side MeCo.

SLA violation data can also be viewed, as can the contents of measurement updates from the Provider-side MeCo.

8.3 SLAng Contract Configuration

Alterations to a service contract file may be necessary in light of any changes in the monitored environment. Changes may be required if either the target service platform (containing the MeCo Interceptors) is relocated to another machine or the network addresses of the service clients need updating.

Note: These configuration steps require knowledge of the SLAng language and associated implementation (including the contract-editing suite).

8.3.1 Service Clients

The clients of the target service must be identified in the associated SLAng contract file. Clients are identified by their IP addresses, and it is important that within each contract file there is a `<ServiceClient>` defined with a `name` attribute corresponding to the IP address of the client.

8.3.2 Provider Definition

Note the value of the name attribute for the linked `<Asset>` referenced within the `<ElectronicService>` declaration. This value identifies the service provider in metric data transmission processes (see Section 8.1.2).

8.3.3 Contract Schedule

The schedule constraining the use of the target service should be correctly defined to allow monitoring of the service. Note the expected dates of contract initiation and completion and convert them to the SLAng date system (where all dates are recorded as the number of milliseconds since January 1st 2000).

8.4 Example Configuration Files

The contents of a sample set of configuration files are included here for reference.

8.4.1 measurement-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<measurement-service
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:/eclipse/workspace/MeCo/misc/measurement-
service.xsd">
  <config>
    <slaEngine>SLAngSimpleEvaluation</slaEngine>
    <messaging>JMS</messaging>
    <replicationEnabled>false</replicationEnabled>
    <componentDecouplingEnabled>false</componentDecouplingEnabled>
    <logging>MySQL</logging>
    <security>JavaKeystore</security>
  </config>
  <replicationPolicy>
    <managedEventType>REQUEST</managedEventType>
    <noOfObjects>3</noOfObjects>
    <maxObjects>10</maxObjects>
    <maxIdleTime>10000</maxIdleTime>
    <whenExhaustedAction>WHEN_EXHAUSTED_GROW</whenExhaustedAction>
  </replicationPolicy>
  <remoteConfiguration>

<namingFactoryInitial>org.jnp.interfaces.NamingContextFactory</namingFactoryInitial>
    <namingProviderURL>jnp://localhost:1099</namingProviderURL>
    <namingFactoryURL>org.jboss.naming:org.jnp.interfaces</namingFactoryURL>
  </remoteConfiguration>
  <slaEngine>
    <contract>C:/eclipse/workspace/MeCo/misc/contracts/test/</contract>
    <loggingEnabled>false</loggingEnabled>
    <produceViolationNotifications>false</produceViolationNotifications>
  </slaEngine>
  <JMS>
    <!-- JBoss MQ -->
    <namingFactory>org.jnp.interfaces.NamingContextFactory</namingFactory>
    <providerURL>localhost:1099</providerURL>
    <connectionFactory>ConnectionFactory</connectionFactory>
    <jndiPrefix>topic</jndiPrefix>
    <listenerPrefix>queue</listenerPrefix>

    <produceViolationNotifications>false</produceViolationNotifications>
    <aggregationModel>NONE</aggregationModel>
    <aggregationContext>CLIENT</aggregationContext>
    <aggregationWindow>100</aggregationWindow>
    <aggregationCount>1</aggregationCount>
  </JMS>
  <probe>
    <configDir>C:/eclipse/workspace/MeCo/misc/probeConfigFiles/test</configDir>
  </probe>
  <gui>
    <chartDir>file:///C:/eclipse/workspace/MeCo/misc/mecoCharts</chartDir>
    <imageDir>C:/eclipse/workspace/MeCo/misc</imageDir>
  </gui>
  <complexMetrics>
    <complexMetric>
      <metricName>clientThroughput</metricName>
      <metricTitle>ClientThroughputALL</metricTitle>
      <metricUnit>requestsPerM</metricUnit>
    </complexMetric>
  </complexMetrics>
  <MySQL>
    <jdbcDriver>com.mysql.jdbc.Driver</jdbcDriver>
    <connection>jdbc:mysql://localhost/meco</connection>
    <user></user>
```

```
    <password></password>
    <metricLog>metric_log</metricLog>
    <violationLog>violation_log</violationLog>
  </MySQL>
  <JavaKeystore>
    <keystoreType>JKS</keystoreType>
    <signatureAlgorithm>MD5withRSA</signatureAlgorithm>
    <keystoreLocation>file:///C:/stores/store_A/</keystoreLocation>
    <localAlias>A</localAlias>
    <localPassword>mugwump</localPassword>
    <securityMode>DIGESTS</securityMode>
  </JavaKeystore>
</measurement-service>
```

8.4.2 EJB Probe Configuration – Fibonacci_EJB.wsdl

```
<?xml version="1.0" ?>

<definitions
targetNamespace="urn:Fibonacci_EJB"
xmlns:tns="urn:Fibonacci_EJB"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
xmlns:ejb="http://schemas.xmlsoap.org/wsdl/ejb/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:mprobe="http://homepages.cs.ncl.ac.uk/s.e.parkin/home.formal/meco/meco-
probe.xsd">

  <!-- message declns -->
  <message name="computeRequestMessage">
    <part name="number" type="xsd:int"/>
  </message>

  <message name="computeResponseMessage">
    <part name="fibString" type="xsd:string"/>
  </message>

  <!-- port type declns -->
  <portType name="FiboPort">
    <operation name="compute">
      <input name="computeRequest" message="tns:computeRequestMessage"/>
      <output name="computeResponse" message="tns:computeResponseMessage"/>
    </operation>
  </portType>

  <!-- binding declns -->
  <binding name="EJBBinding" type="tns:FiboPort">
    <ejb:binding/>
    <format:typeMapping encoding="Java" style="Java">
      <format:typeMap typeName="xsd:string" formatType="java.lang.String" />
      <format:typeMap typeName="xsd:int" formatType="java.lang.Integer" />
    </format:typeMapping>
    <operation name="compute">
      <ejb:operation
         methodName="compute"
         parameterOrder="number"
         returnPart="fibString"
         interface="remote" />
      <input name="computeRequest"/>
      <output name="computeResponse"/>
    </operation>
  </binding>

  <!-- service decln -->
  <service name="FibonacciEJBService">
    <mprobe:probeMethodInfo>
      <mprobe:probeFormat value="EJB"/>
      <mprobe:probeMethod value="compute"/>
      <mprobe:returnNamespace value="http://www.w3.org/2001/XMLSchema"/>
      <mprobe:returnType value="string"/>
```

```
      <mprobe:returnJavaType value="java.lang.String"/>
      <mprobe:probeArg>
        <mprobe:argName value="number"/>
        <mprobe:argValue value="10"/>
        <mprobe:argXMLNamespace value="http://www.w3.org/2001/XMLSchema"/>
        <mprobe:argXMLType value="int"/>
        <mprobe:argJavaType value="java.lang.Integer"/>
      </mprobe:probeArg>
    </mprobe:probeMethodInfo>
    <port name="EJBPort" binding="tns:EJBBinding">
        <!-- Put vendor-specific deployment information here -->
       <ejb:address className="tutorial.interfaces.FiboHome"
                        jndiName="ejb/tutorial/Fibo"
                  initialContextFactory="org.jnp.interfaces.NamingContextFactory"
                     jndiProviderURL="localhost:1099"/>
    </port>
  </service>

</definitions>
```

8.4.3 Probe Descriptor – Fibonacci EJB Service

```
<?xml version="1.0" encoding="UTF-8"?>
<probe-config
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:/eclipse/workspace/MeCo/misc/probe-config.xsd">

<!-- FIBONACCI EJB -->
<wsdlDefinition>file:///C:/eclipse/workspace/MeCo/misc/WSDL/Fibonacci_EJB.wsdl</wsdlDe
finition>
<wsdlNamespace>urn:Fibonacci_EJB</wsdlNamespace>
<service>FibonacciEJBService</service>
<contractIDs>fibonacciSLA_EJB</contractIDs>
<providerID>A</providerID>
<probeMetrics>
   <probeMetric>
      <metricName>responseTime</metricName>
      <metricTitle>ResponseTimeCLIENT</metricTitle>
      <metricUnit>mS</metricUnit>
   </probeMetric>
</probeMetrics>

</probe-config>
```

9. Appendix B – Sample SLAng Contract File


```
<?xml version="1.0" encoding="UTF-8"?>
<XMI version="1.2" xmlns:SLAng="URI pending">
<XMI.header/>
<XMI.content>
	<SLAng:Party xmi.id="mofid:4221705" name="A">
		<SLAng:Party.asset>
			<SLAng:Asset xmi.idref="mofid:18085121"/>
		</SLAng:Party.asset>
		<SLAng:Party.providerDefinition>
			<SLAng:ProviderDefinition xmi.idref="mofid:16667599"/>
		</SLAng:Party.providerDefinition>
	</SLAng:Party>
	<SLAng:Party xmi.id="mofid:30338042" name="client">
		<SLAng:Party.asset>
			<SLAng:Asset xmi.idref="mofid:11600335"/>
		</SLAng:Party.asset>
		<SLAng:Party.clientDefinition>
			<SLAng:ClientDefinition xmi.idref="mofid:18005115"/>
		</SLAng:Party.clientDefinition>
	</SLAng:Party>
	<SLAng:Operation xmi.id="mofid:7059772" name="compute">
		<SLAng:Operation.electronicService>
			<SLAng:ElectronicService xmi.idref="mofid:18085121"/>
		</SLAng:Operation.electronicService>
		<SLAng:Operation.operationDefinition>
			<SLAng:OperationDefinition xmi.idref="mofid:27350423"/>
		</SLAng:Operation.operationDefinition>
	</SLAng:Operation>
	<SLAng:ServiceClient xmi.id="mofid:11600335" name="169.254.139.18">
		<SLAng:ServiceClient.serviceClientDefinition>
			<SLAng:ServiceClientDefinition xmi.idref="mofid:6870277"/>
		</SLAng:ServiceClient.serviceClientDefinition>
		<SLAng:Asset.owner>
			<SLAng:Party xmi.idref="mofid:30338042"/>
		</SLAng:Asset.owner>
	</SLAng:ServiceClient>
	<SLAng:ElectronicService xmi.id="mofid:18085121" name="es1">
		<SLAng:ElectronicService.operation>
			<SLAng:Operation xmi.idref="mofid:7059772"/>
		</SLAng:ElectronicService.operation>
		<SLAng:ElectronicService.electronicServiceDefinition>
			<SLAng:ElectronicServiceDefinition xmi.idref="mofid:7473380"/>
		</SLAng:ElectronicService.electronicServiceDefinition>
		<SLAng:ServiceClient.serviceClientDefinition>
			<SLAng:ServiceClientDefinition xmi.idref="mofid:6870277"/>
		</SLAng:ServiceClient.serviceClientDefinition>
		<SLAng:Asset.owner>
```

```
                <SLAng:Party xmi.idref="mofid:4221705"/>
            </SLAng:Asset.owner>
        </SLAng:ElectronicService>
        <SLAng:Schedule xmi.id="mofid:13572035" name="schedule"
startDate="mofid:15497163" duration="mofid:6455597" period="mofid:11797571"
endDate="mofid:15755548">
            <SLAng:Schedule.scheduledClause>
                <SLAng:ScheduledClause xmi.idref="mofid:806126"/>
                <SLAng:ScheduledClause xmi.idref="mofid:32391332"/>
            </SLAng:Schedule.scheduledClause>
        </SLAng:Schedule>
        <SLAng:ProviderDefinition xmi.id="mofid:16667599" description="pDesc">
            <SLAng:ProviderDefinition.terms>
                <SLAng:Terms xmi.idref="mofid:29519284"/>
            </SLAng:ProviderDefinition.terms>
            <SLAng:ProviderDefinition.party>
                <SLAng:Party xmi.idref="mofid:4221705"/>
            </SLAng:ProviderDefinition.party>
        </SLAng:ProviderDefinition>
        <SLAng:ClientDefinition xmi.id="mofid:18005115" description="cDesc">
            <SLAng:ClientDefinition.terms>
                <SLAng:Terms xmi.idref="mofid:29519284"/>
            </SLAng:ClientDefinition.terms>
            <SLAng:ClientDefinition.party>
                <SLAng:Party xmi.idref="mofid:30338042"/>
            </SLAng:ClientDefinition.party>
        </SLAng:ClientDefinition>
        <SLAng:ClientPerformanceClause xmi.id="mofid:806126" name="CPC"
maximumThroughput="mofid:16335556">
            <SLAng:ClientPerformanceClause.operation>
                <SLAng:OperationDefinition xmi.idref="mofid:27350423"/>
            </SLAng:ClientPerformanceClause.operation>
            <SLAng:ClientPerformanceClause.conditions>
                <SLAng:ElectronicServiceConditions xmi.idref="mofid:1638183"/>
            </SLAng:ClientPerformanceClause.conditions>
            <SLAng:ScheduledClause.schedule>
                <SLAng:Schedule xmi.idref="mofid:13572035"/>
            </SLAng:ScheduledClause.schedule>
        </SLAng:ClientPerformanceClause>
        <SLAng:ElectronicServiceSLA xmi.id="mofid:13162031"
uniqueId="fibonacciSLA_EJB">
            <SLAng:ElectronicServiceSLA.electronicServiceTerms>
                <SLAng:ElectronicServiceTerms xmi.idref="mofid:29519284"/>
            </SLAng:ElectronicServiceSLA.electronicServiceTerms>
            <SLAng:ElectronicServiceSLA.electronicServiceConditions>
                <SLAng:ElectronicServiceConditions xmi.idref="mofid:1638183"/>
            </SLAng:ElectronicServiceSLA.electronicServiceConditions>
            <SLAng:SLA.terms>
                <SLAng:Terms xmi.idref="mofid:29519284"/>
            </SLAng:SLA.terms>
            <SLAng:SLA.conditions>
```

```
                <SLAng:Conditions xmi.idref="mofid:1638183"/>
            </SLAng:SLA.conditions>
        </SLAng:ElectronicServiceSLA>
        <SLAng:ServerPerformanceClause xmi.id="mofid:32391332" name="SPC"
maximumLatency="mofid:12864175" reliability="mofid:1170003"
maxTimeToRepair="mofid:25281771">
            <SLAng:ServerPerformanceClause.operation>
                <SLAng:OperationDefinition xmi.idref="mofid:27350423"/>
            </SLAng:ServerPerformanceClause.operation>
            <SLAng:ServerPerformanceClause.conditions>
                <SLAng:ElectronicServiceConditions xmi.idref="mofid:1638183"/>
            </SLAng:ServerPerformanceClause.conditions>
            <SLAng:ScheduledClause.schedule>
                <SLAng:Schedule xmi.idref="mofid:13572035"/>
            </SLAng:ScheduledClause.schedule>
        </SLAng:ServerPerformanceClause>
        <SLAng:ElectronicServiceTerms xmi.id="mofid:29519284">
            <SLAng:ElectronicServiceTerms.operationDefinition>
                <SLAng:OperationDefinition xmi.idref="mofid:27350423"/>
            </SLAng:ElectronicServiceTerms.operationDefinition>
            <SLAng:ElectronicServiceTerms.serviceClientDefinition>
                <SLAng:ServiceClientDefinition xmi.idref="mofid:6870277"/>
            </SLAng:ElectronicServiceTerms.serviceClientDefinition>
            <SLAng:ElectronicServiceTerms.electronicServiceDefinition>
                <SLAng:ElectronicServiceDefinition xmi.idref="mofid:7473380"/>
            </SLAng:ElectronicServiceTerms.electronicServiceDefinition>
            <SLAng:ElectronicServiceTerms.electronicServiceSLA>
                <SLAng:ElectronicServiceSLA xmi.idref="mofid:13162031"/>
            </SLAng:ElectronicServiceTerms.electronicServiceSLA>
            <SLAng:Terms.sLA>
                <SLAng:SLA xmi.idref="mofid:13162031"/>
            </SLAng:Terms.sLA>
            <SLAng:Terms.providerDefinition>
                <SLAng:ProviderDefinition xmi.idref="mofid:16667599"/>
            </SLAng:Terms.providerDefinition>
            <SLAng:Terms.clientDefinition>
                <SLAng:ClientDefinition xmi.idref="mofid:18005115"/>
            </SLAng:Terms.clientDefinition>
        </SLAng:ElectronicServiceTerms>
        <SLAng:ElectronicServiceConditions xmi.id="mofid:1638183">
            <SLAng:ElectronicServiceConditions.serverPerformanceClause>
                <SLAng:ServerPerformanceClause xmi.idref="mofid:32391332"/>
            </SLAng:ElectronicServiceConditions.serverPerformanceClause>
            <SLAng:ElectronicServiceConditions.clientPerformanceClause>
                <SLAng:ClientPerformanceClause xmi.idref="mofid:806126"/>
            </SLAng:ElectronicServiceConditions.clientPerformanceClause>
            <SLAng:ElectronicServiceConditions.electronicServiceSLA>
                <SLAng:ElectronicServiceSLA xmi.idref="mofid:13162031"/>
            </SLAng:ElectronicServiceConditions.electronicServiceSLA>
            <SLAng:Conditions.sLA>
                <SLAng:SLA xmi.idref="mofid:13162031"/>
```

```
            </SLAng:Conditions.sLA>
        </SLAng:ElectronicServiceConditions>
        <SLAng:OperationDefinition xmi.id="mofid:27350423" description="opDef"
failureCriteria="null">
            <SLAng:OperationDefinition.serverPerformanceClause>
                <SLAng:ServerPerformanceClause xmi.idref="mofid:32391332"/>
            </SLAng:OperationDefinition.serverPerformanceClause>
            <SLAng:OperationDefinition.clientPerformanceClause>
                <SLAng:ClientPerformanceClause xmi.idref="mofid:806126"/>
            </SLAng:OperationDefinition.clientPerformanceClause>
            <SLAng:OperationDefinition.terms>
                <SLAng:ElectronicServiceTerms xmi.idref="mofid:29519284"/>
            </SLAng:OperationDefinition.terms>
            <SLAng:OperationDefinition.operation>
                <SLAng:Operation xmi.idref="mofid:7059772"/>
            </SLAng:OperationDefinition.operation>
        </SLAng:OperationDefinition>
        <SLAng:ServiceClientDefinition xmi.id="mofid:6870277" description="scDef">
            <SLAng:ServiceClientDefinition.terms>
                <SLAng:ElectronicServiceTerms xmi.idref="mofid:29519284"/>
            </SLAng:ServiceClientDefinition.terms>
            <SLAng:ServiceClientDefinition.serviceClient>
                <SLAng:ServiceClient xmi.idref="mofid:11600335"/>
                <SLAng:ServiceClient xmi.idref="mofid:18085121"/>
            </SLAng:ServiceClientDefinition.serviceClient>
        </SLAng:ServiceClientDefinition>
        <SLAng:ElectronicServiceDefinition xmi.id="mofid:7473380" description="esDef">
            <SLAng:ElectronicServiceDefinition.terms>
                <SLAng:ElectronicServiceTerms xmi.idref="mofid:29519284"/>
            </SLAng:ElectronicServiceDefinition.terms>
            <SLAng:ElectronicServiceDefinition.electronicService>
                <SLAng:ElectronicService xmi.idref="mofid:18085121"/>
            </SLAng:ElectronicServiceDefinition.electronicService>
        </SLAng:ElectronicServiceDefinition>
        <SLAng:Duration xmi.id="mofid:24590868" value="100.0" unit="mS"/>
        <SLAng:Duration xmi.id="mofid:25281771" value="15000.0" unit="mS"/>
        <SLAng:Duration xmi.id="mofid:12864175" value="15.0" unit="mS"/>
        <SLAng:Duration xmi.id="mofid:11797571" value="283.0" unit="day"/>
        <SLAng:Duration xmi.id="mofid:10732982" value="1950.0" unit="day"/>
        <SLAng:Duration xmi.id="mofid:6455597" value="283.0" unit="day"/>
        <SLAng:Duration xmi.id="mofid:31226686" value="1667.0" unit="day"/>
        <SLAng:Percentage xmi.id="mofid:1170003" value="1.0"/>
        <SLAng:Date xmi.id="mofid:15755548" sinceJan12000="mofid:10732982"/>
        <SLAng:Date xmi.id="mofid:15497163" sinceJan12000="mofid:31226686"/>
        <SLAng:Frequency xmi.id="mofid:16335556" period="mofid:24590868"/>
</XMI.content>
</XMI>
```