Department of Computer Science
Series of Publications A
Report A-2008-2

XML Messaging for Mobile Devices

Jaakko Kangasharju

Academic Dissertation

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XIV, University Main Building, on January 26th, 2008, at 10 o'clock.

University of Helsinki
Finland

ISSN 1238-8645
ISBN 978-952-10-4483-0 (paperback)
ISBN 978-952-10-4484-7 (PDF)
http://ethesis.helsinki.fi/

Computing Reviews (1998) Classification: C.2.4, H.4.3, C.4, I.7, E.2, E.3

Helsinki University Printing House
Helsinki, January 2008 (xxiv + 255 pages)

XML Messaging for Mobile Devices

Jaakko Kangasharju
Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
http://www.cs.helsinki.fi/u/jkangash/
jkangash@cs.helsinki.fi

Abstract

In recent years, XML has been widely adopted as a universal format for structured data. A variety of XML-based systems have emerged, most prominently SOAP for Web services, XMPP for instant messaging, and RSS and Atom for content syndication. This popularity is helped by the excellent support for XML processing in many programming languages and by the variety of XML-based technologies for more complex needs of applications.

Concurrently with this rise of XML, there has also been a qualitative expansion of the Internet's scope. Namely, mobile devices are becoming capable enough to be full-fledged members of various distributed systems. Such devices are battery-powered, their network connections are based on wireless technologies, and their processing capabilities are typically much lower than those of stationary computers.

This dissertation presents work performed to try to reconcile these two developments. XML as a highly redundant text-based format is not obviously suitable for mobile devices that need to avoid extraneous processing and communication. Furthermore, the protocols and systems commonly used in XML messaging are often designed for fixed networks and may make assumptions that do not hold in wireless environments.

This work identifies four areas of improvement in XML messaging systems: the programming interfaces to the system itself and to XML processing, the serialization format used for the messages, and the protocol used to transmit the messages. We show a complete system that improves the overall performance of XML messaging through consideration of these areas.

The work is centered on actually implementing the proposals in

a form usable on real mobile devices. The experimentation is performed on actual devices and real networks using the messaging system implemented as a part of this work. The experimentation is extensive and, due to using several different devices, also provides a glimpse of what the performance of these systems may look like in the future.

Computing Reviews (1998) Categories and Subject Descriptors:

C.2.4 Distributed Systems
H.4.3 Communications Applications
C.4 Performance of Systems—*Measurement Techniques*
I.7 Document and Text Processing
E.2 Data Storage Representations
E.3 Data Encryption—*Standards*

General Terms: Measurement, Performance, Security, Standardization

Additional Key Words and Phrases: XML messaging, binary XML, XML processing interfaces, mobile and wireless communication

Acknowledgements

First and foremost, I would not be here without my advisor, Kimmo Raatikainen. For nearly ten years, he has guided, supported, and trusted me. He always had time for me, and could in a few words show me what was good in my work or open up new avenues of research. I am also grateful to him for including me in standardization work, so I could see how my research can be useful for actual deployed systems as well.

Working in the Fuego Core project, I got to know many exceptionally bright people, and collaboration with them also strengthened my own work. Especially I would like to thank Sasu Tarkoma for his participatory leadership in the project, and Tancred Lindholm for all the discussions of my papers and XML processing work, without which this dissertation would have been considerably poorer. Also Oriana Riva, though never part of the project, was always supportive and interested in my progress. Without her to inspire me and to solve the problems with phone measurements ahead of me, this work would certainly have taken much longer to complete.

The Helsinki Institute for Information Technology provided an enjoyable and stimulating environment for carrying out this research. This is in no small part due to its director Martti Mäntylä and his commitment to the researchers at HIIT. Thanks are also due to the HIIT IT staff led by Pekka Tonteri who have always been cooperative and willing to look for ways to permit maximal freedom to researchers without compromising the integrity of the infrastructure.

Participating in XML-related standardization was a valuable experience that deepened my understanding of both XML and its

surrounding community. I am grateful to all members of the XML Binary Characterization and Efficient XML Interchange Working Groups for expanding my perspective, and in particular to Robin Berjon, Takuki Kamiya, and John Schneider for all the discussions we had over beers or other consumables.

I wish to thank the reviewers of this dissertation, Jari Porras and Antti Ylä-Jääski, as well as the reviewers of my licentiate thesis, Jukka Manner and Jari Porras again, for all their comments that helped me improve this work. And my heartfelt thanks to Tancred Lindholm for voluntarily reading this dissertation and commenting on all aspects of it.

Finally, throughout the 30-odd years of my life, my family has been there for me. My parents, Pentti and Riitta, have always supported and encouraged me, no matter what choices I have made. Considering that my ideas of my future career used to change practically yearly, I would not be here without their understanding attitude. Also, as my path has moved toward computer science and distributed computing, my brother Jussi has increasingly been assisting me and discussing technical matters with me.

Helsinki, December 13, 2007
Jaakko Kangasharju

Contents

List of Figures

List of Tables

List of Abbreviations

2G	second generation
3G	third generation
ALICE	Architecture for Location Independent CORBA Environments
AMME	Abstract Mobile Message Exchange
API	Application Programming Interface
ARC	Adaptive Replacement Cache
ARR	asynchronous request-response
ASN.1	Abstract Syntax Notation One
BEEP	Blocks Extensible Exchange Protocol
BER	Basic Encoding Rules
BXSA	Binary XML for Scientific Applications
CBMS	Convergence of Broadcast and Mobile Services
CD	content density
CDC	Connected Device Configuration
CDR	Common Data Representation
CLDC	Connected Limited Device Configuration
COA	Codec Omission Automaton

CORBA	Common Object Request Broker Architecture
CPU	central processing unit
DFA	deterministic finite automaton
DNS	Domain Name System
DOA	Decoding Omission Automaton
DOM	Document Object Model
DoS	Denial of Service
DTD	Document Type Definition
EDGE	Enhanced Data rates for GSM Evolution
EOA	Encoding Omission Automaton
EXI	Efficient XML Interchange
F/OSS	Free/Open Source Software
FP	Foundation Profile
GIOP	General Inter-ORB Protocol
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
HIP	Host Identity Protocol
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronic Engineers
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference

IP	Internet Protocol
JavaME	Java Micro Edition
JavaSE	Java Standard Edition
JAXB	Java Architecture for XML Binding
JIT	just-in-time
JVM	Java Virtual Machine
LAN	Local Area Network
LRU	Least Recently Used
MAN	Metropolitan Area Network
MEP	Message Exchange Pattern
MHM	Multiplexed Hierarchical Modeling
MIDP	Mobile Information Device Profile
MIME	Multipurpose Internet Mail Extensions
MPEG	Moving Picture Experts Group
MTOM	Message Transmission Optimization Mechanism
NAT	Network Address Translation
OASIS	Organization for the Advancement of Structured Information Standards
OMG	Object Management Group
ORB	Object Request Broker
OS	operating system
P/S	publish/subscribe
PAN	Personal Area Network
PDA	Personal Digital Assistant

PER	Packed Encoding Rules
PGP	Pretty Good Privacy
PPM	Prediction by Partial Matching
PSVI	post-schema-validation infoset
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTT	round-trip time
SAX	Simple API for XML
SDP	Service Discovery Protocol
SGML	Standard Generalized Markup Language
SIP	Session Initiation Protocol
SOAP	Simple Object Access Protocol
SRR	synchronous request-response
SSL	Secure Sockets Layer
StAX	Streaming API for XML
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
URI	Universal Resource Identifier
URL	Uniform Resource Locator
UWB	Ultra-Wideband

W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WBXML	WAP Binary XML
WG	Working Group
WiMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless LAN
WML	Wireless Markup Language
WSDL	Web Services Description Language
WTLS	Wireless Transport Layer Security
WWW	World Wide Web
XBC	XML Binary Characterization
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XOP	XML-binary Optimized Packaging
XSBC	XML Schema-based Binary Compression

Part I

Introduction

CHAPTER 1

Introduction

> It has *never* seemed to me a good idea to put anything in writing.

This dissertation examines the fundamental properties of messaging middleware for the distributed applications of the future. The key research question is how to take advantage of existing standardized, widely-used formats like XML and common Internet protocols in the future where the nature of distributed computing differs considerably from the environment in which the existing systems were developed.

1.1 Motivation

Extensible Markup Language (XML) [W3C, 2006a] is nowadays a very common format for representing structured data. This includes its use as a messaging format for distributed systems, and there is evidence that the applicability of XML is widening more and more. Therefore it seems that any distributed system of the future that is not inherently closed will need to be able to handle XML data in some manner.

The number of mobile phones in the world has increased dramatically during the past decade, easily surpassing the number of personal computers. The number of Internet-capable phones may

not be quite as large, but, e.g., Keshav [2005] makes the claim that phones are the devices that will drive the development of the Internet in the future. Therefore focusing on how phones can function as full-fledged members of the Internet appears to be necessary.

Reconciling these two, the open distributed systems in the future will be based on XML and include a large contingent of mobile phones. However, XML is a text-based format, intentionally designed to contain much redundancy. This is not at all compatible with mobile phones, which require efficiently-processable compact formats to spare their limited energy through decreased processing and communication time.

Distributed applications have already become sufficiently complex that it is no longer feasible to develop them directly on top of the operating system. Accordingly, a variety of *middleware platforms* [Aiken et al., 2000], which provide common functionality for communication, have emerged to ease the development burden. It is likely that most sophisticated systems of the future will be based on some form of a middleware platform.

Most current middleware platforms were originally designed for fixed networks, and this makes their suitability for mobile devices questionable. The main challenges come from the nature of mobility [Raatikainen et al., 2002]: as a device moves, the conditions around it change, and the middleware platform will need to adapt to these changing conditions. Adapting existing platforms does not seem sufficient, but rather new platforms will need to be designed to take into account the special needs of mobile devices.

1.2 Research Methodology

This work has from the beginning been close to applied research, with the intent to implement prototypes that run correctly on existing platforms and demonstrate the improvements. The methods have been chosen accordingly. The beginning of the work was to delineate the scope of the research to XML messaging on mobile devices, and after this, work on the identified subtopics proceeded independently of each other.

The main phases of the research process that we followed are illustrated in Figure 1.1. Research into each topic began by gath-

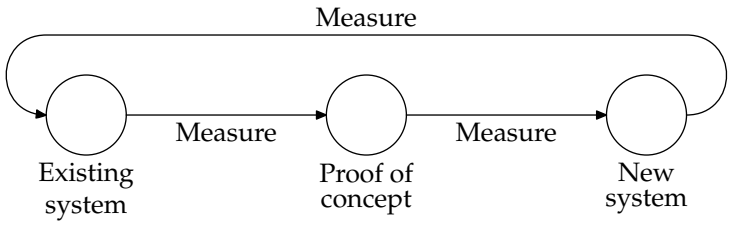


Figure 1.1: The research process

ering existing systems that implement the required functionality. The performance of these systems was measured and the areas of poor performance identified. Further analysis of the measurements then revealed the main causes of poor performance.

After the causes of poor performance were identified, the preliminary measurement framework was extended with proof-of-concept implementations of potential improvements. These implementations focused on each potential improvement separately, the purpose being approximate quantification of the amount of improvement possible. The intent was not to produce a complete system yet, but to guide the eventual design.

Measurements with the proof-of-concept implementations revealed the most fruitful avenues for further work. Based on these findings, the new complete system was designed in a manner that enables the identified improvements to be made. The design was kept modular, separating each individual area of research into its own component to make it possible to quantify the performance of each improvement in isolation. The implementation was coordinated with users of the prototype to ensure the presence of sufficient functionality.

After the implementation was complete, performance measurement code was written to compare the proposed system with existing systems. During experimentation the implementation of the system was improved to the extent possible with localized optimizations. The viability of the ideas was verified through positive measurement results.

When the new system was finished, and performance measurements had revealed that it achieves its goals, it was added to the set of existing systems, and the process started again from the beginning. Namely, experience in using the system brought to light both

missing functionality and poor performance in areas that were not originally considered. Correcting some of these problems necessitated redesigning parts of the system, so to ensure that the new design would be suitable, the same process was also followed for the new improvements.

1.3 Research History

The work in this dissertation was performed in the Fuego Core project[1] at the Helsinki Institute for Information Technology[2] during the years 2002–2006. The project investigated issues in middleware for the future mobile wireless Internet [Tarkoma et al., 2006], and the work presented here formed the investigation into using XML as a foundational building block in a middleware platform for mobile devices.

The middleware platform developed in the Fuego Core project, called *Fuego middleware*, has as its main components the XML messaging and processing system presented here, an event system for asynchronous many-to-many communication [Tarkoma, 2006], and a data synchronizer with specific support for XML data [Lindholm et al., 2005]. The complete platform is available under a Free/Open Source Software (F/OSS) license[3], and also includes components for mobile presence, event interoperability with Session Initiation Protocol (SIP) [Ramya, 2005], and Host Identity Protocol (HIP) [Komu et al., 2005].

The work on messaging began with experiments on the performance of a popular SOAP implementation [Kangasharju et al., 2003] using a variety of networks, protocols, message formats, and messaging interfaces. Based on this initial work, we determined the four central areas of improvement in XML messaging: system interfaces, XML processing interfaces, XML serialization formats, and messaging protocols.

A depiction of how our identified components join together to produce a messaging system is shown in Figure 1.2, with message

[1] http://www.hiit.fi/fi/fc/
[2] http://www.hiit.fi/
[3] http://hoslab.cs.helsinki.fi/homepages/fuego-core/

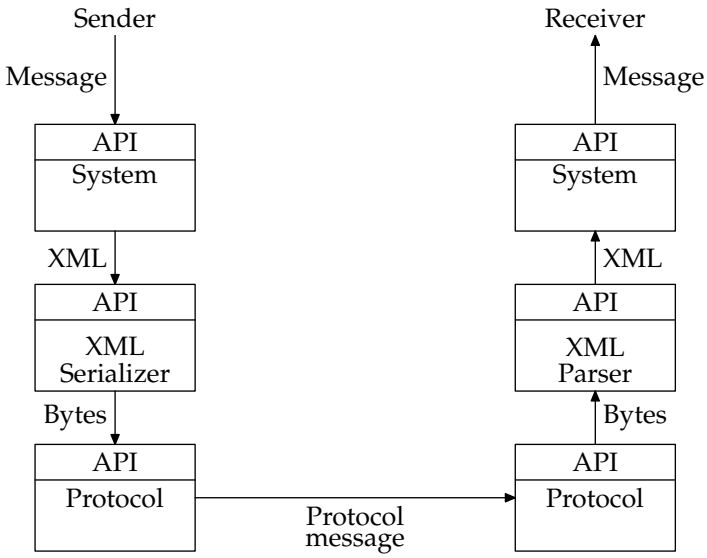


Figure 1.2: Components in XML-based messaging

flow from sender to receiver indicated by arrows and labeled according to the form the message takes at each point of the flow. As the Figure shows, we consider the Application Programming Interface (API) of each component to also be an important part of that component, worthy of study in its own right, and also necessary to achieve an integrated system without coupling the components tightly together.

The first version of our messaging system [Kangasharju et al., 2005a] was based on the requirements that we had identified from the rest of the middleware platform. We included improved versions of the components identified in Figure 1.2, including components for XML processing [Kangasharju and Lindholm, 2005] and XML serialization [Kangasharju et al., 2005b]. This system was also presented in the author's Licentiate of Philosophy thesis [Kangasharju, 2006].

This version of the messaging system could be said to be incomplete in some ways. For one, even though the system ran on mobile phones, we had run most of our experimentation on laptop computers, and the characteristics of the two platforms are very different. During experiments on mobile phones, we noted that some parts of the implemented messaging system seemed some-

what heavyweight for mobile phones, so we decided to further develop the application interface and the protocol layer.

Another concern, which is a crucial part in modern distributed systems, was security. There exist security specifications for XML messages, but there are no widely-available implementations for mobile phones. According to our methodology, we began work on security by investigating the performance of XML security [Kangasharju et al., 2006] and identified the key issues that would need to be overcome.

Implementation of the proof-of-concept XML security system for our preliminary experimentation also revealed some deficiencies in our original XML processing interface. We therefore decided to redesign this interface, applying the principles that we had found to work in our original implementation, but improving the interface's extensibility and processing efficiency [Kangasharju, 2007]. The complete new system, proceeding from requirements through design to actual implementation, is described in [Kangasharju et al., 2007a].

During the time this research was in progress, the World Wide Web Consortium (W3C) began considering alternate serialization formats for XML [W3C, 2003b]. Based on our own work in the area, we participated in this work from the very beginning [Kangasharju and Raatikainen, 2003]. We were involved in both the XML Binary Characterization (XBC) Working Group (WG)[4] that charted the field and the work needed, and the Efficient XML Interchange (EXI) WG[5] that is in the process of designing a format. The author of this dissertation is one of the editors of the EXI Measurements Note [W3C, 2007b].

The work presented in this dissertation was performed in its entirety by the author, with the exception of the EXI measurement framework presented in section 8.2 that was the joint work of Sun Microsystems and AgileDelta, based on a benchmark platform developed by Sun Microsystems. The XML differencing tool described in subsection 8.2.1 was designed and implemented by Tancred Lindholm, with the author responsible for its schema support. The measurements analyzed in section 8.3 were run by Ca-

[4] http://www.w3.org/XML/Binary/
[5] http://www.w3.org/XML/EXI/

rine Bournez on computers at the Naval Postgraduate School and by John Schneider on computers available to him, but the analysis, including the design of the analysis, is fully the author's own.

1.4 Contributions

This work has produced a complete messaging system implementation that runs on real devices in real conditions. This is not insignificant as implementing ideas in practice always turns up unforeseen issues, both in application design and in existing platforms. The implementation has been the subject of two demonstrations (WMCSA 2004, Pervasive 2007) as well as a basis for a context-aware middleware platform [Riva, 2006].

The main contributions are in the area of XML processing and serialization on mobile devices, the XML component of Figure 1.2. The XML processing API is a significant advance over existing APIs, providing an extensible and versatile system for processing XML in a number of different styles. This is evidenced by our ability to use its generic functionality to implement efficient and novel applications in widely different areas of XML processing.

The XML serialization format is significant as a publicly-available format that considers mobile phones explicitly in its design, and its compression performance is comparable to other modern formats. The format and the experiences gained during its development have also influenced corresponding standardization at the W3C, and some features of the designed format are candidates for adoption in the eventual standard.

The work on combining a binary XML format with XML security has not been performed before to a comparable extent. We have explored deeply the issues of this area and propose an implemented extension that goes a great deal towards improving performance. Furthermore, we also provide specific recommendations on how to avoid massive processing overhead on mobile phones when the other party of the communication has sufficient processing power available.

1.5 Structure of the Dissertation

The dissertation begins in Part II with an overview of the two most relevant areas of interest: XML and related technologies, with an emphasis on using XML for messaging, in chapter 2, and mobile computing, from networks and devices to writing distributed applications, in chapter 3. As there has so far been little overlap between these two areas, the text in these Chapters is intentionally written to assume little or no prior knowledge.

Part III covers all the components of the developed messaging system. The specifics of communication, at both application and protocol levels, are the topic of chapter 4. This Chapter begins with an overview of the system architecture, and then describes the components System and Protocol of Figure 1.2, as implemented in the system.

The rest of Part III covers the XML component of Figure 1.2. The processing API and how it provides efficient and versatile XML processing are covered in chapter 5 and the serialization format used for XML data by the system is described in chapter 6.

Measurements and their analysis are the topic of Part IV. In this Part, chapter 7 covers the extensive experiments that we performed on the messaging system, using real mobile devices and wireless networks. The analysis performed by the author for the EXI WG is presented in chapter 8, with an emphasis on documents and performance relevant to mobile messaging.

Finally, chapter 9 concludes the dissertation, summarizing the contributions, and looks at the future, both how the presented system is expected to affect coming developments and also the directions in which further development appears the most fruitful. Auxiliary code that is not necessary for full understanding of the work is provided in Appendix A.

Part II

Overview of XML and Mobile Computing

CHAPTER 2

The XML Stack

> We must choose one tribe, and remove our favor from all others.

XML [W3C, 2006a] has, since its inception, become a widely accepted markup language for all kinds of data. Its basic model of data is that of a tree of nodes. Since trees are also a fundamental construct in programming language data, XML has been applied to representing general structured data. This is useful for interchange purposes as it provides a standard way to represent the data to be exchanged between applications on varied platforms.

A multitude of technologies have sprung up around XML. The W3C has been active in producing many of them, but due to the large interest in XML, some have been produced by other organizations. This collection of XML-based technologies is often called the *XML stack*, based on the idea that they are stacked on top of the XML base. In addition to XML itself, we also cover those parts of the XML stack that we consider relevant to our topic.

2.1 Core XML Technologies

XML was originally born from the desire to streamline Standard Generalized Markup Language (SGML) [ISO, 1986] for use on the

World Wide Web (WWW). For this purpose the designers set the following design goals (from [W3C, 2006a]):

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

The intent of many of these design goals was to eliminate complexities in SGML that made it hard to implement processors and to understand documents.

2.1.1 Basic XML

The original XML definition [W3C, 1998] was completed in 1998. Currently XML version 1.0 is in its fourth edition [W3C, 2006a], and there is also version 1.1 [W3C, 2006b] to address Unicode [Unicode Consortium, 2003] evolution and concerns about whitespace handling. However, as XML 1.1 is incompatible with XML 1.0 (this incompatibility was, in fact, the reason for the increased version number), adoption has not been enthusiastic.

We show an example XML document in Figure 2.1. The top line is the *XML declaration*, which declares common information about the document such as the version of XML that it conforms to. It also declares the encoding used for XML's character set, Unicode.

```
<?xml version="1.0" encoding="UTF-8"?>
<person nationality="DE">
  <name>
    <first>Richard</first>
    <last>Wagner</last>
  </name>
  <occupation>Composer</occupation>
  <born>1813-05-22</born>
  <died>1883-02-13</died>
</person>
```

Figure 2.1: An example XML document

The values shown are the defaults. The `<person>` *tag* starts the `person` *element* and the `</person>` tag ends it; an XML document may contain only one element at its top level, which is called its *root element*. Elements may include other elements (like `name` here), *text* (`Wagner`), or *attributes* (`nationality`).

Whatever is between an element's start tag and end tag is called its *content*, e.g., `Wagner` is the content of the `last` element in Figure 2.1. If an element contains both text and other elements, it is said to have *mixed content*. An element without any content, such as `<foo></foo>`, may also be represented by an *empty element tag*, `<foo/>`. An empty element tag may also include attributes.

While XML did achieve its goal of simplicity, at least when compared with SGML, use on the heterogeneous WWW requires more. The basic XML definition suffices for single-source vocabularies where every element's meaning is defined by a single entity. However, for wide-area distributed use it is beneficial to be able to define common vocabularies for general areas that can then be used for parts of such documents. For example, we could imagine the `person` element of Figure 2.1 to be defined by a genealogy institute and then used by anyone who wants to include data about people in their XML document.

This problem is solved by XML Namespaces [W3C, 2006c]. This specification splits an XML name into two parts, its *namespace* and *local name*, with the intent that each entity has complete control over its namespaces, permitting the creation of general vocabularies and avoiding collisions in names. Concretely, a namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<favorite-composers xmlns:p="http://example.org/people">
  <p:person>
    <p:name>
      ...
    </p:name>
    ...
  </p:person>
  <p:person>
    ...
  </p:person>
</favorite-composers>
```

Figure 2.2: An example XML document with namespaces

is identified with a Universal Resource Identifier (URI) [Berners-Lee et al., 2005], as recommended by the architecture of the World Wide Web [W3C, 2004a].

For use in an XML document, a generic URI would be cumbersome and usually not permitted as an XML name. Therefore, the namespace URIs need to be mapped to *prefixes* for use in XML documents. The complete name of an element is then presented as a combination of its namespace URI's prefix and its local name. An XML document that conforms to this specification is called *namespace-well-formed*.

The use of namespaces is demonstrated in Figure 2.2 where we have placed the `person` element of Figure 2.1, and the elements it contains, into the namespace `http://example.org/people`. This namespace is mapped to the prefix `p` by the attribute `xmlns:p` of the document's root element. The prefix is then used with the colon (`:`) to construct the *qualified names* of the elements from the corresponding namespace. The root element `favorite-composers` does not belong to any namespace.

2.1.2 XML Schema Languages

Applications using XML will typically not expect to process arbitrary documents, but only documents containing certain elements and attributes arranged in a certain way. For instance, a processor reading the document in Figure 2.2 will expect a `favorite-com-`

`posers` root element containing several `p:person` elements. To define these kinds of syntactic constraints for XML documents, there exist various *schema languages*.

XML documents conforming to the syntax rules of the XML definition are called *well-formed*, a term defined in the XML specification [W3C, 2006a]. Schemas divide the class of XML documents into two subclasses: *valid* documents conform to the schema that is being used, and *invalid* ones do not. An important point is that there does not need to be a fixed specification of which schema is used to validate an XML document, and in many applications the schema used will be solely determined by the document processor without input from the document creator.

The first schema language, originally defined for SGML but also included in simplified form in the XML specification [W3C, 2006a], is called Document Type Definition (DTD). Rules expressible in a DTD provide a simple grammar to describe the contents of XML documents. The XML specification allows an XML document to contain a hard-coded reference to its DTD or to even contain this DTD as an *internal subset*.

A possible DTD for the XML document in Figure 2.1 is given in Figure 2.3. The name in the `DOCTYPE` part defines the root element of valid XML documents. The content of each element is given in sequence, with optional parts marked with a `?`. Attributes of elements are given separately with the `ATTLIST` declaration, which gives the name, type, and default value for each attribute. The `#PCDATA` stands for *parsed character data*, i.e., text.

There are two problems with DTDs, both visible in Figure 2.3. The first is that they do not support namespaces at all. To get the effect of namespaces, the names in a DTD need to be declared with their prefixes, and hence the same prefixes need to be used everywhere when validating. The second problem is that there is no support for data types. In our example, the elements `born` and `died` are clearly dates, so it would be very useful if the schema language were to support declaring that.

These omissions are fixed with XML Schema [W3C, 2004f,g], an XML schema language developed by the W3C. Semantically speaking, XML Schema is a superset of DTDs [Murata et al., 2005], i.e., for any DTD there exists an XML Schema that validates exactly the same XML documents.

```
<!DOCTYPE person [
  <!ELEMENT person (name,occupation?,born,died?)>
  <!ATTLIST person nationality CDATA #IMPLIED>
  <!ELEMENT name (first,middle?,last)>
  <!ELEMENT first (#PCDATA)>
  <!ELEMENT middle (#PCDATA)>
  <!ELEMENT last (#PCDATA)>
  <!ELEMENT occupation (#PCDATA)>
  <!ELEMENT born (#PCDATA)>
  <!ELEMENT died (#PCDATA)>
]>
```

Figure 2.3: An example DTD for the example XML document

We show a part of an XML Schema for our example document in Figure 2.4. This only shows a part of the definition of the `person` element and the `born` element. As we can see, the `p` prefix for our namespace is declared in the root `xs:schema` element and used later in element names. The `targetNamespace` attribute ensures that the defined elements are also in our namespace. Finally, the definition of the `born` element illustrates the use of data types, also defined by XML Schema.

In addition to DTD and XML Schema, there exist several other schema languages. Many of these were merged into either XML Schema or another schema language, RELAX NG [OASIS, 2001]. This latter is based on the theory of tree languages [Brüggemann-Klein et al., 2001], and is seen by many to be a much cleaner language than XML Schema. In addition to the normative XML definition, RELAX NG also has a compact syntax potentially more familiar to programmers [OASIS, 2002b], an example of which is shown in Figure 2.5 for the example document of Figure 2.2. RELAX NG has been shown to be strictly more expressive of structure than either DTD or XML Schema [Murata et al., 2005].

The above schema languages are called *grammar-based* in that they specify the structure of elements declaratively. An alternative kind of schema language is *rule-based* where relationships between element contents are specified algorithmically. The best-known rule-based schema language is called Schematron [Jelliffe,

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           targetNamespace="http://example.org/people"
           xmlns:p="http://example.org/people">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="p:name"/>
        <xs:element minOccurs="0" ref="p:occupation"/>
        ...
      </xs:sequence>
      <xs:attributeGroup ref="p:nationality"/>
    </xs:complexType>
  </xs:element>
  ...
  <xs:element name="born" type="xs:date"/>
</xs:schema>
```

Figure 2.4: A partial XML Schema for the example XML document

```
namespace p = "http://example.org/people"
start = favorite-composers
favorite-composers = element favorite-composers {
  element p:person {
    attribute nationality { xsd:string { pattern = "\w\w" } },
    element p:name {
      element p:first { token },
      element p:middle { token }?,
      element p:last { token }
    },
    element p:occupation { token }?,
    element p:born { xsd:date },
    element p:died { xsd:date }?
  }+
}
```

Figure 2.5: A RELAX NG Compact Syntax schema for the example XML document

2002]. Grammar-based and rule-based languages can also be used to complement each other by using the grammar-based language to specify the structure and using the rule-based language to specify constraints that the grammar-based language does not support.

2.1.3 XML Data Models

The XML definition considers only the character-level syntax of XML (also called "Unicode with angle brackets"). However, an application that uses XML will often view it as representing a tree consisting of elements, attributes, and text, or as James Clark, co-author of RELAX NG, puts it [van der Vlist, 2003],

> The abstraction is a labelled tree of elements. Each element has an ordered list of children in which each child is a Unicode string or an element. An element is labelled with a two-part name consisting of a URI and local part. Each element also has an unordered collection of attributes in which each attribute has a two-part name, distinct from the name of the other attributes in the collection, and a value, which is a Unicode string.

The W3C has produced two different data models for XML. The older one is XML Information Set (Infoset) [W3C, 2004e], which attempts to faithfully capture all relevant information from a namespace-well-formed XML document and present it as a tree consisting of *information items*, each containing a small amount of information. In most XML-related specifications produced by the W3C, XML is viewed through the Infoset specification. XML Schema is specified to transform an XML Infoset into a *post-schema-validation infoset (PSVI)*, an information set where the information items are annotated with types.

Another data model produced by the W3C is the XQuery 1.0 and XPath 2.0 data model [W3C, 2007h]. This was produced for the needs of the XML processing languages XQuery [W3C, 2007g] and XSLT [W3C, 2007i], and their associated addressing language XPath [W3C, 2007f]. It extends the Infoset with support for type information and collection representation.

For the purposes of many applications, these various data models are perfectly suitable. However, as is pointed out in [W3C,

2005e], distinctions even in whether attribute values use single or double quotes can be significant for some applications (as an addition to the mentioned XML editors, we offer version control systems where tools should not change any such data indiscriminately). This calls for a lower-level data model, based on characters or bytes.

We can naturally see XML, produced by the grammar in the XML definition, possibly complemented with a character encoding, as a data model in its own right, which would be the perfect candidate data model for some applications. However, since XML processing systems typically cannot preserve this representation, there is a way to *canonicalize* XML [W3C, 2001]. Canonical XML is a way to have several independent XML processors produce the same byte sequence from two "equivalent" XML documents. There is no formal definition of this equivalence, but Canonical XML has been constructed so that people in the XML community would agree that two XML documents are equivalent if they have the same canonical form.

This proliferation of data models is a natural consequence of specifying only a character-level representation without attaching any semantics to any pieces of data. This is widely seen as a good thing [Sperberg-McQueen, 2005], as it allows XML to be modeled according to the application's needs, which is reflected in the number and variety of data models.

2.1.4 XML Application Programming Interfaces

To use XML in an application, some API is needed for the programming language used. The main concern in such an API is support for parsing XML, as that is the more difficult task. Usually, some way is also provided to serialize XML from the application data, but often applications simply include direct calls to the lower-lever character-writing routines, since writing XML is perceived to be a simple exercise.

The existing XML parsing APIs can be divided into two classes. A *streaming* API produces *events*, small atomic pieces of XML, for the application to consume one by one. The API treats these events as ephemeral, not storing them anywhere. The other kind of API is the *tree* API, where an XML document consists of *nodes*, again

small atomic pieces of XML, but with links to each other that correspond to the tree structure of XML. With a tree API, the nodes are usually kept in memory.

Streaming APIs for parsing exist in two forms, *push-style* and *pull-style*. In a push-style API the parser will usually have a single function called `parse` or the like that takes control of parsing and makes calls to application-registered callback functions for each event. In contrast, a pull-style parser API has a function to extract the next event and its relevant information, keeping the application completely in control.

The oldest XML API, designed alongside the finalization of the XML specification itself, is Simple API for XML (SAX) [Brownell, 2002], a push-style streaming API. One benefit of SAX compared to many of its successors is that it provides access to all parts of an XML document, including structured access to a possible DTD. Another, almost as old, is Document Object Model (DOM) [W3C, 2004b], a tree API specified by the W3C.

One failing of these two APIs is that they were designed before XML namespaces, so their namespace support has been included later, and because of backward compatibility they must also support the old-style namespace-unaware processing. DOM is also viewed as cumbersome because the API is specified in OMG IDL [OMG, 2004] to make it language-independent, but this also makes it unnatural for some programming languages.

The earliest pull-style streaming API was XmlPull [Slominski, 2004]. Having been designed after XML namespaces, its events, especially in serializing, are fully based on namespace URIs, and qualified names are not directly available to the application. XmlPull is significant in that kXML[1], the best-known XML implementation for mobile devices, implements the XmlPull API. On desktop systems, it has been superseded as the default pull-style API by its successor, Streaming API for XML (StAX) [BEA, 2003].

For tree APIs, Java programmers in particular have been active in trying to replace DOM. JDOM[2] is an attempt to make a pure Java API in that JDOM uses familiar Java classes and interfaces to represent the nodes and iterate over them. The newest well-

[1] `http://kxml.sourceforge.net/`
[2] `http://www.jdom.org/`

known tree API is XOM[3], in part a reaction to observed problems in all other XML APIs.

Considering these XML APIs in terms of data models, it is rare for an API to be precisely specified as corresponding to some data model. Rather, an API implicitly defines a data model for XML in that the model for a document is what the API's parser produces from that document. The XML APIs mentioned above are all defined for XML with namespaces, and do not include support for the type information of the PSVI or the XPath data model.

A completely different way of processing XML is provided by *XML data binding* [Sosnoski, 2003b], a process where code to convert between XML and programming language data is generated from a schema. This approach may make it easier to use XML in some applications, especially as a replacement for existing systems, but it is also limited in that some schema is required, invalid documents cannot be processed, and often processing some XML constructs like mixed content is cumbersome.

2.2 Web Services

To use XML for messaging, some form of infrastructure needs to be built, containing at least a syntax for messages and a description of the transfer protocol. Furthermore, various auxiliary specifications will be needed for different systems and services that can be built on top of messaging. XML-based messaging infrastructure is commonly called *Web services*.

We will here cover the SOAP-style "structured" approach to Web services. An alternate method of implementing Web services that has gained prominence in recent years is *Representational State Transfer (REST)* [Fielding, 2000], which is the architectural style of the WWW and its main protocol, Hypertext Transfer Protocol (HTTP). The main benefits of REST over SOAP are seen to be its requirement for stateless interaction and cacheability of communicated data.

Fundamentally, REST is based on *resources* that are each individually addressable and connect to each other through hypertext

[3] http://www.xom.nu/

links. In a REST application an interaction between a client and a server is accomplished by the client requesting and sending data to a series of resources, in contrast to the SOAP style where the client always sends messages to the same "resource" and the content of the message determines the state of the interaction.

2.2.1 XML Protocols

The first well-known use of XML for the interchange of programming language data was the XML-RPC [Winer, 2003] system of UserLand Software. This is a simple way to perform Remote Procedure Calls (RPCs) using XML over HTTP. It supports encoding the usual programming language types as well as structured data and arrays into XML.

While XML-RPC has been found suitable for a variety of applications, it lacks the kind of extensibility that is often required in distributed systems. To correct this state of affairs, Simple Object Access Protocol (SOAP) [W3C, 2000a] was devised. The main design was still to use XML as a data format for messages, but other considerations were relaxed; however, HTTP was still the only specified protocol.

The SOAP 1.1 specification also describes how to encode structured data, the so-called *SOAP encoding rules*, which define how to encode arbitrary data into XML, including cyclic structures. These rules are used in the also-specified SOAP for RPC.

The SOAP 1.1 specification was published as a Note of the W3C. After that, the W3C decided to work on XML-based protocols and formed the XML Protocol Activity, which was later transformed into the XML Protocol WG[4] of the Web Services Activity[5]. This WG produced version 1.2 of SOAP [W3C, 2007c], which relegates most of the areas specific to protocols and usage scenarios to its adjuncts [W3C, 2007d]. We shall focus exclusively on SOAP 1.2, even though SOAP 1.1 is still popular in existing systems.

The SOAP specification only defines the outer structure of a SOAP message, illustrated in Figure 2.6. This Figure shows the root element, `Envelope`, with its children, the optional `Header` and

[4] http://www.w3.org/2000/xp/Group/
[5] http://www.w3.org/2002/ws/

```
<soap:Envelope xmlns:soap='http://www.w3.org/2003/05/soap-envelope'>
  <soap:Header>
    <target soap:role='http://www.w3.org/2003/05/soap-envelope/role/next'
            soap:mustUnderstand='true'>
      ...
    </target>
    <priority soap:relay='true'>
      ...
    </priority>
    ...
  </soap:Header>
  <soap:Body>
    ...
  </soap:Body>
</soap:Envelope>
```

Figure 2.6: The SOAP message structure

the mandatory `Body`. The children of the `Header` element are called *header blocks*, and the example illustrates the common attributes that SOAP defines for header blocks. Note that SOAP itself does not define any header blocks, only common attributes for them.

The specified attributes for header blocks are used by the SOAP processing model. This model begins with the *initial sender* sending a message, the message passing through zero or more *intermediaries*, and finally being processed by the *ultimate receiver*. Collectively, these processors are called *SOAP nodes*. The `role` attribute specifies which nodes in this chain are intended to process the header block, the `mustUnderstand` attribute set to `true` specifies that if a node does not understand the header block, it must respond with an error message, and the `relay` attribute set to `true` specifies that the node is to retain the header block in the message instead of removing it.

The SOAP specification does not concern itself with the particulars of message transfer. It only defines a protocol framework that can be used to specify how an underlying protocol is used to transmit SOAP messages, and defines a protocol binding for HTTP. This binding allows both one-way and request-response messaging. Other protocol bindings have been specified for email [W3C, 2002c] and XMPP [Forno and Saint-Andre, 2005].

The XML Protocol WG has also produced some other specifications on message formats. These specifications were driven by the need to transmit binary data inside SOAP messages, a concern that was handled by SOAP with Attachments [W3C, 2000b] for SOAP 1.1. The desired characteristics of this attachment feature were first specified on an abstract level [W3C, 2004c].

The main issue solved by an attachment feature for SOAP is transmission of binary data, e.g., images. If embedded as such inside an XML document, they need to be base64-encoded [Freed and Borenstein, 1996a], which both takes significant processing time and increases the size of the data by one third. Further concerns were the ability to embed XML from other sources: a complete XML document is not embeddable inside XML, and even for fragments there are the questions of namespace prefix mappings and different character encodings. Finally, XML element delimiters can only be recognized by reading delimiters from the serialized form, so embedded binary data will create overhead as the parser will need to read every character in it.

The solution produced by the XML Protocol WG was XML-binary Optimized Packaging (XOP) [W3C, 2005g], a generic mechanism for including binary data in XML. XOP was intentionally limited to the case where the binary data is base64-encoded in the Infoset representation of the XML, and allows the separation and direct binary representation of such data. It requires that the XML document, along with any such binary data, be packaged inside a format such as Multipurpose Internet Mail Extensions (MIME) `multipart/related` [Levinson, 1998]. Any binary content inside the Infoset representation is then replaced with a pointer to the corresponding part in the package.

A way to use XOP to include binary data in SOAP messages is specified by SOAP Message Transmission Optimization Mechanism (MTOM) [W3C, 2005b]. This defines how a SOAP message is packaged in MIME format using XOP, and defines a feature for the SOAP HTTP binding to indicate that this optimization is being used. A later specification [W3C, 2005a] defines how the Internet media type [Freed and Borenstein, 1996b] of the binary data can be included also in the XML instead of just in the packaging.

2.2.2 Protocol Extensions

The SOAP processing model allows a very flexible way to define extensions to the protocol. An extension will specify one or more header blocks with names in its own namespace, and semantics for them. The standard attributes defined for the header blocks allow a robust manner of using the extensions, as even unaware processors are required to recognize what to do with these extension headers, even if they do not implement the actual extension.

The Web Services Activity includes an Addressing WG[6] chartered with defining how messages are addressed so that they can be delivered to their proper destinations, and responded to by their receivers. This work is based on a submission [W3C, 2004d] from a group of W3C members. The Addressing WG has produced Recommendations for the core principles [W3C, 2006d] and for a SOAP binding [W3C, 2006e].

The core Addressing specification defines an *endpoint reference* that can be used to describe a Web service message recipient. The specification further defines *addressing properties*, which allow correlation of messages, e.g., to indicate the destination of a message or to specify a request being responded to. These are all defined using an XML Infoset representation, which also allows extensibility. The SOAP binding for Addressing defines how a SOAP message can indicate that Addressing is in use, and how the abstract core concepts are mapped to SOAP headers.

An example of how Addressing could be used in a SOAP message is shown in Figure 2.7. The `MessageID` element denotes a unique identifier for the message, the `To` element denotes the endpoint that is the target of the message, the `ReplyTo` element directs replies to the message, and the `Action` element identifies the semantics of the message. In accordance with the architecture of the WWW [W3C, 2004a], all of these elements are URIs.

In addition to the W3C, Organization for the Advancement of Structured Information Standards (OASIS) has been very active in defining standards related to Web services. One of the main specifications of OASIS is the ebXML Message Service [OASIS, 2002a], which defines a messaging service on top of SOAP 1.1 to sup-

[6] `http://www.w3.org/2002/ws/addr/`

```
<soap:Envelope xmlns:soap='http://www.w3.org/2003/05/soap-envelope'
               xmlns:wsa='http://www.w3.org/2005/08/addressing'>
  <soap:Header>
    <wsa:MessageID>http://example.org/client/1</wsa:MessageID>
    <wsa:To>http://example.com/service</wsa:To>
    <wsa:ReplyTo>
      <wsa:Address>http://example.org/client</wsa:Address>
    </wsa:ReplyTo>
    <wsa:Action>http://example.com/purchase</wsa:Action>
  </soap:Header>
  <soap:Body>
  ...
  </soap:Body>
</soap:Envelope>
```

Figure 2.7: A WS-Addressing header

port secure and reliable messaging. These reliability and security features have since been further refined by OASIS into Web Services Reliability [OASIS, 2004a] and Web Services Security [OASIS, 2004b]. The latter will be covered in detail in subsection 2.3.4.

Web Services Reliability (WS-Reliability) is intended to provide reliability guarantees to SOAP messaging, including at-most-once, at-least-once, and exactly-once semantics, as well as ordered delivery of messages. These are handled by SOAP headers, in which the sender will include elements indicating its requirements.

2.3 Security

In the modern networked world, security, along with its related topics, trust and privacy, has become perhaps the most important concern for new systems. Broadly speaking, security has three components [Bishop, 2003]:

Confidentiality Concealment of information or even the fact that information exists

Integrity Verification of information and its origin

Availability Ability to access information

2.3.1 Security in Messaging

While all of the components of security listed above are needed in communication, a middleware messaging system does not need to concern itself with all aspects of each. In particular, availability is not usually a matter of systems features, but rather a matter of designing and implementing the system so that erroneous data does not cause crashes, and of administering the network so that Denial of Service (DoS) attacks can be detected and countered.

Confidentiality in messaging is achieved through a variety of *encryption* algorithms. These algorithms can be divided into *symmetric*, or private-key, systems, where the same secret key is used for both encrypting and decrypting, and *asymmetric*, or public-key, systems where encryption is performed using a publicly-available key but decryption is possible only with a secret key.

The invention of asymmetric cryptography [Diffie and Hellman, 1976; Rivest et al., 1978] was a true boon to communications. In symmetric systems, the most significant problem to solve is key distribution: since the key needs to be shared by the sender and receiver of encrypted messages, the sender needs to deliver it somehow to the receiver. This makes it difficult to communicate with a party that has not been previously encountered, a situation that is common in Internet-like public distributed systems.

Asymmetric cryptography solves this, since the receiver's public key is available to anyone, including the sender. The sender can therefore simply encrypt the messages with that key, and only the receiver can then decrypt them. In practice, asymmetric algorithms are orders of magnitude slower than symmetric algorithms, so the message is typically encrypted with a symmetric algorithm and only the key used for this is encrypted with an asymmetric algorithm, as is done in the popular Pretty Good Privacy (PGP) system [Garfinkel, 1994]. Assuming the use of secure algorithms, this achieves the same benefits as full asymmetric encryption of the whole message, but with much better performance.

Integrity in messaging typically uses two related technologies, *signatures* and *certificates*. A signature is computed over a message and included with it so that the receiver can ensure that the message has not been altered. Usually signatures rely on similar techniques as asymmetric cryptography [Elgamal, 1985; Rivest

et al., 1978] in that the sender computes the signature using its secret key, and verification of the signature is then possible using the public key. The standard way to compute a signature is to *hash* the signed content (e.g., with SHA-1 [Eastlake and Jones, 2001]) and then apply the signing algorithm to this hash value.

A cryptographic signature proves that the signer is in possession of the secret key used for the signature, and passing the signature check means that message integrity has not been compromised. However, signatures themselves do not address the other part of integrity, verification of origin when the signing party is previously unknown. Namely, most often parties are not identified by their cryptographic keys, though there are instances where this is the case, such as HIP [Moskowitz and Nikander, 2006] or the developers in the Debian project[7]. Therefore, cryptographic keys can be accompanied with certificates, which are essentially signatures linking the keys to some other identifier for the party, by which the certificate issuer certifies that the identified party is the possessor of the cryptographic key.

Use of certificates naturally raises the issue of how users then trust the certifiers. The approach in PGP is called *Web of trust*, which is based on each user rating the quality of other users as certifiers, and these quality ratings are then used to calculate how trustworthy an identification is. The more common approach, also used for secure communication on the WWW, is to rely on *trusted third parties*, who provide the service of signing public keys after receiving some proof of identity or authorization. Since trust in these parties is built in to all applications, this establishes the needed certification. Further discussion of these alternatives is provided in, e.g., [Garfinkel et al., 2005; Perrin, 2003].

A common method of achieving security in communication between two parties is to begin a session by establishing a *secure channel*, i.e., a communication path that is both encrypted and authenticated, and then using this channel for all communication. Establishment of such a channel is achieved by generating a symmetric encryption key using a key exchange protocol [Diffie and Hellman, 1976] and then using this symmetric key to encrypt all communication. This method forms the basis of the most popular security

[7] http://www.debian.org/

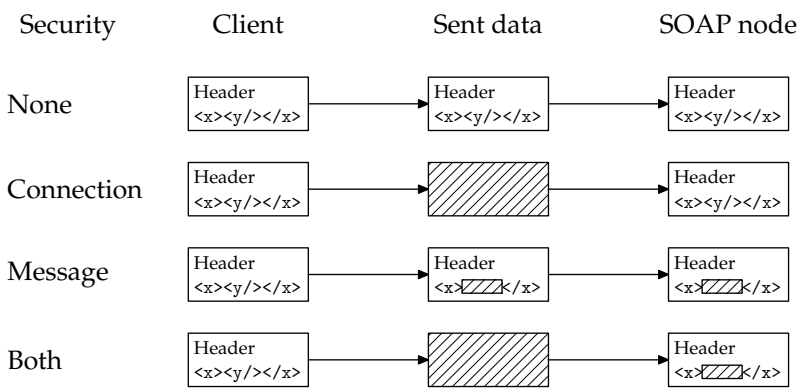


Figure 2.8: Message flow and content with different kinds of security

protocol on the Internet, Secure Sockets Layer (SSL) [Freier et al., 1996], standardized by the Internet Engineering Task Force (IETF) as Transport Layer Security (TLS) [Dierks and Allen, 1999].

2.3.2 XML-level Security

Current XML messaging systems typically use HTTP as the protocol, and use SSL to conceal and authenticate the communication. However, sometimes this *connection-level* security is not sufficient. For instance, in the SOAP processing model, a message can pass through several SOAP intermediaries, and with connection-level security, the full message will be visible to each of these intermediaries. Therefore, to achieve security between the initial sender and ultimate receiver, security needs to be applied at the *message level*, even at a granularity of individual XML elements.

Figure 2.8 illustrates connection-level and message-level security. Here the client is sending a SOAP message with a header and a body, and wishes to encrypt the content of the `x` element in the body. With connection-level security the content is protected while in transit but is fully visible at the first SOAP node that receives it. In contrast, with message-level security the element content is encrypted even at the first node. A combination of both approaches, encrypting the full message in transit but also encrypting the content, is also possible and needed in some use cases.

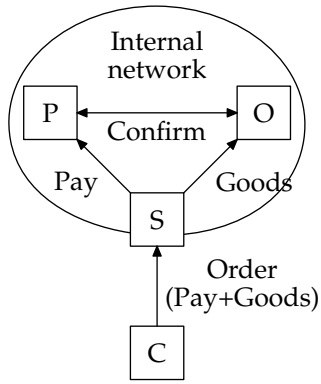


Figure 2.9: Security with an online retailer

We will consider two usage scenarios in more detail. In an online shopping system illustrated in Figure 2.9, the client C sends a purchase order to the merchant's outward-facing system S. S extracts the payment information (e.g., credit card number) for the payment processor P and the list of ordered goods for the order processor O. O will also need to communicate with P to verify that the payment succeeded. The client will encrypt the payment information to prevent it from falling into the wrong hands, and the merchant requires the client to sign the order for identification.

With just connection-level security, both client verification and decryption of payment information must happen at S. In contrast, with message-level security S merely needs to extract the relevant parts from the message and forward them inside the merchant's internal network. Therefore, with message-level security, S can be a simpler system that does not do any security processing, so compromising S will not be sufficient for an attacker to alter orders or extract payment information.

Similar concerns apply to workflow systems. In a workflow system, a series of computers are connected to each other, and messages pass through each of them. Each computer will do some processing on the message before passing it to the next system. Depending on the makeup of the system, there may be a need for securing the messages as they flow through the system.

The messages processed by workflow systems can be large, and each individual computer may need to process only a small part of

each. With connection-level security, each computer will need to decrypt and verify the complete messages, as well as encrypt and sign them. If security processing happened at the level of individual XML elements, each computer would need to process only the parts that it is interested in.

A further benefit in workflow systems is compartmentalization. Since each XML element is encrypted individually, they can be encrypted only to the computers in the system that need that information. In this way, computers that do not need some information will not even have access to that information, and signatures that have been made remain attached to their original signers and not to any intermediaries.

2.3.3 XML Security Specifications

The W3C has produced several specifications for encrypting and signing XML documents at the level of individual elements. The main specifications are XML Signature [W3C, 2002f] and XML Encryption [W3C, 2002e], and they are complemented by two canonicalization specifications [W3C, 2001, 2002b], as well as a method to interoperate when signed content is later encrypted [W3C, 2002a]. An example of an XML document with both signed and encrypted content is shown in Figure 2.10, with the actual content omitted.

The main XML element defined by XML Signature is the `Signature` element[8]. Such an element contains, at a minimum, a `SignedInfo` element and a `SignatureValue` element. It may also contain a `KeyInfo` element to identify the key used, either by including it directly into the document or by providing a known identifier for it, the latter being what the example in Figure 2.10 does. Finally, the `Signature` element may contain a number of `Object` elements that may carry arbitrary data.

The actual signature is carried in the `SignedInfo` and `SignatureValue` elements. The former of these contains a number of `Reference` elements, each of which identifies its corresponding content by URI (this content does not have to be XML). In the example of Figure 2.10, the `Reference` element refers to the `part`

[8] All elements defined by XML Signature are in the namespace `http://www.w3.org/2000/09/xmldsig#`.

```
<document xmlns:ds='http://www.w3.org/2000/09/xmldsig#'
          xmlns:xenc='http://www.w3.org/2001/04/xmlenc#'>
  <ds:Signature>
    <ds:SignedInfo>
      <ds:CanonicalizationMethod Algorithm='...'/>
      <ds:SignatureMethod Algorithm='...'/>
      <ds:Reference URI='#contract'>
        <ds:DigestMethod Algorithm='...'/>
        <ds:DigestValue>...</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>...</ds:SignatureValue>
    <ds:KeyInfo>
      <ds:KeyName>...</ds:KeyName>
    </ds:KeyInfo>
  </ds:Signature>
  <part id='contract'>
    <xenc:EncryptedData Type='...#Element'>
      <xenc:EncryptionMethod Algorithm='...'/>
      <ds:KeyInfo>
        <ds:KeyName>...</ds:KeyName>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
    ...
  </part>
</document>
```

Figure 2.10: XML encryption and signature example

element using a relative URI referring to the `id` attribute. A *digest*, i.e., a hash value, is computed over the referenced content and included in the `Reference` element. The content of the `SignatureValue` element is then a signed digest over the `SignedInfo` element. Both the digest and signature methods are selectable, with a few alternatives mandated by the specification to be available.

Digest computation is always performed on byte sequences, and if an XML document is processed by an intermediary, the serialized form may change. To permit signature verification despite this, the `Reference` element permits the use of *transforms* to the content to be specified, and the `SignedInfo` element mandates the presence of a *canonicalization method* to specify how the XML in the `SignedInfo` element was actually converted into bytes. If the content referenced by the `Reference` elements is XML, one of the specified transforms should usually be a canonicalization method.

The W3C has specified two canonicalization methods, the original Canonical XML [W3C, 2001], also called *inclusive*, and the later *exclusive* canonicalization [W3C, 2002b]. The difference between these is the treatment of in-scope namespaces in the canonicalization of subtrees. Figure 2.11(a) shows an XML document, of which the subtree rooted at `n2:b` is canonicalized inclusively in Figure 2.11(b) and exclusively in Figure 2.11(c). Inclusive canonicalization includes all in-scope namespaces in the `n2:b` tag while exclusive includes only the ones actually used in the subtree.

The example in Figure 2.11 also demonstrates some other features of canonicalization. Namely, attributes and namespace prefixes in a start tag are lexicographically sorted, with all namespace prefixes coming before any of the attributes. Also, empty elements are represented explicitly as start tag–end tag pairs and not as empty element tags.

The reason for exclusive canonicalization is applications such as SOAP where the signed content can be considered separable from its containing document. If this signed content is extracted and inserted into another document, and the set of in-scope namespaces changes because of this, a signature generated using inclusive canonicalization will not validate anymore. On the other hand, inclusive canonicalization is faster to perform, since it does not require going through the whole signed subtree and collecting the namespace prefix mappings that must be included.

```
<n1:a xmlns:n1="urn:y">
  <n2:b xmlns:n2="urn:x">
    <n2:c/>
    <n2:d  y="y" x="x">
      Text
    </n2:d>
  </n2:b>
</n1:a>
```

(a) Original

```
<n2:b xmlns:n1="urn:y"
      xmlns:n2="urn:x">
  <n2:c></n2:c>
  <n2:d x="x" y="y">
    Text
  </n2:d>
</n2:b>
```

(b) Inclusive

```
<n2:b xmlns:n2="urn:x">
  <n2:c></n2:c>
  <n2:d x="x" y="y">
    Text
  </n2:d>
</n2:b>
```

(c) Exclusive

Figure 2.11: Two XML canonicalization examples

The example of Figure 2.10 also shows the content of the `part` element having been encrypted. This is implemented by replacing the original content with an `EncryptedData` element as defined by XML Encryption[9]. XML Encryption also defines an `EncryptedKey` element to carry an encryption key, the possible content of which is a superset of that of `EncryptedData`.

An `EncryptedData` element includes, at a minimum, a `CipherData` element that can either reference the encrypted content or include it in a `CipherValue` element. It may also include the encryption method that was used as well as a `KeyInfo` element of XML Signature to identify the key used for encryption. The `Type` attribute can be used when the encrypted content is XML to denote whether it is an element or the content of an element.

An `EncryptedKey` element has the same possible child elements as an `EncryptedData` element, except the encrypted content in this case is an encryption key. The additional content it has is a list

[9] All elements defined by XML Encryption are in the namespace `http://www.w3.org/2001/04/xmlenc#`.

of DataReference and KeyReference elements that refer to EncryptedData and EncryptedKey elements, respectively, encrypted with the key contained in the element. As always, these references may point to either the same document or to external content.

2.3.4 Web Services Security

Web Services Security (WS-Security) [OASIS, 2006a] from OASIS defines how these XML security technologies are applied to SOAP messages. This specification appears to be what message security in Web services is converging towards.

WS-Security uses SOAP's standard extensibility mechanism by defining a *security header block* for SOAP messages. A security header block identifies those parts of a SOAP message that signatures or encryption have been applied to. It is possible for a message to contain more than one security header block, each targeted at a different SOAP intermediary, which provides support for the workflow scenario described above.

Identification of encryption and signature keys in WS-Security is typically accomplished through *security tokens*. A security token identifies, in some manner, the party responsible for encrypting or signing parts of the message. In principle, the token can identify the party in any manner understandable to both communication endpoints, but common methods include username-based tokens [OASIS, 2006b] and X.509 certificates [OASIS, 2006c]. The latter of these provides a certificate of identity authenticated by a trusted third party, and may therefore be usable even when no previous association exists.

An example of a security header block is shown in Figure 2.12. The element shown here would be a child of the SOAP Header element. The example also shows another feature of WS-Security, the *timestamp*. A timestamp gives a creation date and an expiration date, which are useful in protecting against *replay attacks* where an eavesdropper resends a previously-sent message in an attempt to cause the receiver to process it twice. A BinarySecurityToken can carry any type of security token encoded as a byte sequence; here the ValueType attribute identifies it as an X.509 token.

As can be seen from Figure 2.10 and Figure 2.12, adding a security header block to a SOAP message consumes a lot of space,

```
<wsse:Security
    xmlns:wsse='http://docs.oasis-open.org/wss/2004/01/oasis-
                200401-wss-wssecurity-secext-1.0.xsd'
    xmlns:wsu='http://docs.oasis-open.org/wss/2004/01/oasis-
               200401-wss-wssecurity-utility-1.0.xsd'
    xmlns:ds='...' xmlns:xenc='...'>
  <wsu:Timestamp wsu:Id='...'>
    <wsu:Created>...</wsu:Created>
    <wsu:Expires>...</wsu:Expires>
  </wsu:Timestamp>
  <wsse:BinarySecurityToken ValueType='...#X509v3'>
    ...
  </wsse:BinarySecurityToken>
  <xenc:EncryptedKey>
    ...
  </xenc:EncryptedKey>
  <ds:Signature>
    ...
  </ds:Signature>
</wsse:Security>
```

Figure 2.12: A Web Services Security SOAP header block

especially when considering that most of the attribute values we left out are URIs, which tend to be long. In our measurements on WS-Security performance [Kangasharju et al., 2006] we noted that the Apache WSS4J implementation[10] on default settings generates a security header block with size over 6 kilobytes. If the messages themselves are very small, adding security header blocks might be unacceptable overhead in many systems.

2.4 XML Performance

Performance is an ever-present concern in computer systems, and in XML-based systems the bottleneck has often been observed to

[10] http://ws.apache.org/wss4j/

be XML processing, sometimes parsing and sometimes higher layers of the XML stack. It is therefore necessary to investigate the performance of systems, identify the actual causes for poor performance, and attempt to improve the performance by different implementation techniques.

2.4.1 Existing Measurements

SOAP as a protocol has been adopted by the Grid [Foster and Kesselman, 2004] community for communicating scientific data. Therefore they have been especially active in charting the bottlenecks of XML messaging, with a focus on scientific computing. More relevant to our work has been the mobile computing community, but there the measurements usually include networking, so we cover them in subsection 3.3.2.

One of the oldest, and still perhaps the most famous, measurement of SOAP performance was published by Chiu et al. [2002] in the context of Grid computing. This work studied the structure of a SOAP implementation, identified performance issues in each component, and built a highly-optimized SOAP processor to address these issues. The optimized processor took advantage of schema-specific parsing, had special code to handle arrays, and eliminated all buffering between the application and the system Input/Output (I/O) layer.

In the end, Chiu et al. eliminated most of the issues in SOAP processing, leaving a problem inherent to XML. Namely, in the optimized processor, over 90 % of the processing time was spent in converting floating-point values between their internal representations and the text required by XML. Addressing this issue would require a binary encoding for data, which would then be no longer compatible with the standard XML encoding that every current SOAP system expects.

SOAP and XML have also been making inroads into the financial sector. In that field, there is an established text-based protocol called FIX [FIX, 2001]. Kohlhoff and Steele [2004] compare the performance of FIX to SOAP and to the binary Common Data Representation (CDR) used in CORBA [OMG, 2004]. From the result that SOAP performs poorly while the performance of FIX is close to that of CDR, Kohlhoff and Steele conclude that the poor

performance of SOAP is most likely due to poor implementations and not to any inherent weakness of a text-based format.

Both of the former measurements are quite old, from a time when SOAP implementations were not nearly as mature as they are today. In the context of Grid computing, Head et al. [2006] have proposed a benchmarking suite for evaluating the performance of SOAP processors. The SOAP messages in this benchmarking suite are designed so that they measure both the performance of different features of the SOAP processors and the performance in typical Grid computing applications.

Head et al. [2006] also evaluate several different SOAP processors against this benchmarking suite. Their conclusions are that when using C, the best parsers to use are gSOAP[11] and Expat[12]. For Java, they recommend Piccolo[13] instead of the better-known Apache Xerces[14]. However, our latest experiments [Kangasharju and Tarkoma, 2007] point in the direction that the latest release of Sun's Java implementation brings Xerces to the same level of performance as Piccolo, and another parser, Woodstox[15], is better than either. Though this is indicative of the changing landscape of XML parser performance, it must be noted that the full benchmark of Head et al. was more extensive than our simple measurement.

2.4.2 Efficiency in XML Processing

As the importance of XML processing in modern systems grows, so do naturally concerns over the performance of XML processors. Mostly, the focus has been on improving parser performance, as XML serialization is seen as quite trivial and not amenable to as much improvement as parsing. Many of the techniques rely on a schema of some form existing for the processed documents, either explicitly or implicitly.

Serialization of SOAP messages has been considered by Abu-Ghazaleh et al. [2004]. Their approach is based on noticing that

[11] http://www.cs.fsu.edu/~engelen/soap.html
[12] http://expat.sourceforge.net/
[13] http://piccolo.sourceforge.net/
[14] http://xerces.apache.org/
[15] http://woodstox.codehaus.org/

many applications send several messages to the same Web service endpoint, and therefore these messages bear close resemblance to each other. When serializing a message to a new Web service for the first time in this system, the serialized form is also saved into a buffer. When a later message shares some or all of the same structure, this pre-serialized form can be used to avoid serializing again the shared structure.

Later, Abu-Ghazaleh and Lewis [2005] applied the same technique to deserialization. This is based on creating *checkpoints* in the byte stream, essentially saving the parser state. Whenever the system notices that it is parsing bytes with the same semantics as data that was previously parsed, it can simply replay the results created before without needing to parse the bytes.

Takase et al. [2005] have applied the same technique in their implementation. The differences from the work of Abu-Ghazaleh and Lewis are that Takase et al. store the full byte sequences instead of just checksums, and use a deterministic finite automaton (DFA) that matches byte sequences and outputs SAX events.

These techniques work best when the processed XML documents are similar to each other, e.g., when they all conform to the same schema. However, these systems are not actually capable of using a schema. Rather, they extract common information from the documents during processing, essentially building a schema-derived data structure.

In contrast to such implicit schema use, other approaches have focused on explicit schema use. Both Chiu and Lu [2004] and van Engelen [2004] compile a schema into an automaton. The difference between these two approaches is that Chiu and Lu compile the automaton into executable code while van Engelen uses it as a data structure. In static cases, the former approach can be preferable, as all the message types can be known beforehand, and direct compilation into code may offer better performance. However, as Web services are typically very loosely coupled, a more dynamic system that can accommodate new types of messages at run time is usually preferred.

XML Screamer [Kostoulas et al., 2006] takes an integrated approach to parsing. The basic observation is that, like network protocols, XML processing is defined in specifications using a layered model, but like in network protocols, it is not sensible to mirror this

layering in the implementation [Watson and Mamrak, 1987]. Accordingly, XML Screamer integrates parsing, validation, and object construction into a single system.

The main benefit of XML Screamer is its ability to avoid reading data more than once. The integrated approach allows decoding the serialized bytes into characters and these characters further into XML names or typed values in a single pass over the data. At its heart, XML Screamer is a schema-based system, compiling a schema into the executable code of a high-performance schema-specific parser, thus sharing the disadvantages of the system of Chiu and Lu [2004]. However, XML Screamer can also be used with a fully permissive schema, and even in this mode it outperforms other high-performance XML parsers.

An interesting approach designed for multiprocessors has been designed by Lu et al. [2006]. Their parallel parsing system begins by *preparsing* the XML document to construct a skeleton version, i.e., a data structure containing only the tree structure of the document, none of the actual content. After the tree structure has been extracted, the document is partitioned equally for all processors, and can thus be parsed in parallel. As the preparsing phase does not need to perform many expensive well-formedness constraints, it is much faster than a full parse.

Many of the techniques described above are not suitable for mobile devices. For instance, the caching parser of Takase et al. [2005] requires megabytes of memory to store its memorized documents and the parallel parsing of Lu et al. [2006] requires a multiprocessor as well as reading the whole document into memory. Similarly, schema processing and compilation can be a prohibitive additional cost, even though the runtime costs of the schema compilation approaches are not excessive when only a few schemas are used.

2.4.3 Efficiency in XML Security

The XML security specifications are defined in terms of the Infoset model, which is a tree-based view that allows arbitrary traversals of the XML document. However, using an explicit tree data structure in an implementation of such a specification is often not a very good idea from the performance perspective. A common alternative is the attempt to process XML in a fully streaming manner,

which provides memory usage benefits as well as the ability to pipeline several processing stages together. However, if the processing semantics are defined in terms of a tree model, it may be difficult to avoid buffering.

Interest in efficient implementation of XML security has existed for as long as there has been XML security. An early effort by Imamura et al. [2002] implemented a streaming XML Encryption processor using special features of Apache Xerces. While there was little improvement in encryption performance, the streaming decryption was several times faster than a traditional implementation based on DOM.

On the XML Signature front, Shirasuna et al. [2004] noted that unless message-level security is required, SSL outperforms XML-level security mechanisms. Shirasuna et al. also measured that over 90 % of the signature generation time was spent on canonicalization. They thus conclude that optimizing the canonicalization phase is the most fruitful avenue for improvement.

The lessons of Shirasuna et al. [2004] were taken to heart by Lu et al. [2005] who integrated a streaming canonicalizer into the XML parser. This approach was over 6 times as fast as the comparison point, mostly due to the much faster canonicalization performance. In addition, as the system of Lu et al. is a streaming processor, its memory consumption remains constant. The problem with this approach is that it does not support all signed documents, only those where the signature precedes the signed content.

Earlier we saw examples of XML parsers that exploited commonalities between parsed documents and cached information to avoid expensive re-parsing. A similar approach was applied to WS-Security by Makino et al. [2005] who build templates for commonly occurring XML fragments and can then use these templates to directly process the same bytes again. This also helps in canonicalization, though some special processing needs to be performed for in-scope namespaces. This work is similar to [Takase et al., 2005] and, in fact, some of the same people were involved in both.

OASIS [2003] also worked on a minimal profile for WS-Security. The basic concept is that the sender of a WS-Security message can add assertions regarding the content of the message that help the receiver process it faster. For instance, one assertion specifies that the signed parts are transmitted in canonical form, obviating the

need for separate canonicalization at the receiver side. Unfortunately, this work never progressed past the draft stage.

2.5 Concluding Remarks

XML as the format for messages in a distributed system has the benefit of ubiquity. The popularity of XML means that there will nearly always be an existing implementation for an application's needs, and many issues that emerge have already been solved by some technology in the XML stack.

The message format described by SOAP has many benefits. The separation of the header from the body, with full namespace support, lets a system differentiate between message metadata and data, and specify types of metadata independently of any other uses of SOAP. The use of a language such as Web Services Description Language (WSDL) [W3C, 2007e] permits giving a schema definition for the messages as well. It is thus likely that SOAP messages can be associated with either complete or partial schema information, which comes from multiple sources.

Messaging applications are rarely interested in XML as such, but only as an interchange format. Such applications will therefore view XML only through some data model, most commonly the implicit one that is present in the API they use. Thus, it is necessary for the XML API to provide a convenient ability to convert between XML and application data, but on the other hand, current data binding approaches appear too rigid due to their insistence on a fully-correct schema.

An efficient XML processor implementation is a necessity in the world of mobile devices. Most of the research in XML processing efficiency appears to concentrate on large machines with abundant resources, as evidenced by the memory consumption requirements of the techniques described in subsection 2.4.2. Thus, it is not certain whether XML can be accepted as a suitable messaging format in mobile computing.

CHAPTER 3

Mobile Computing

> Whit a weerd bludy playce;
> ye wooldnae bileeve it.

Distributed systems become significantly different when wireless communication and mobile nodes are introduced. These differences are not merely quantitative in that the links and nodes have just worse performance, but also qualitative in that the characteristics also differ from fixed links and stationary computers. A deep understanding of the devices, networks, communication protocols, and programming support is crucial for developing future applications that function properly in the new environment.

3.1 Device and Network Characteristics

Mobile computing in this dissertation refers to computing on small mobile devices such as Personal Digital Assistants (PDAs) or mobile phones[1]. While laptop computers have seen use in mobility research as well, the convenience of the much smaller form factor and cheaper price has led to the ubiquity of mobile phones. The networking capabilities of modern phones have even led to

[1] Mobile phones with advanced capabilities, especially for programming, are also called *smartphones*.

Keshav [2005] claiming that the future of the Internet will be dominated by phones.

While form factor is an important enabler of the ubiquity of mobile phones, we shall not consider it further. Form factor has its main effect in user interface considerations, and our work is focused on the middleware level. Therefore our primary interest is in the technologies used in the devices and their properties.

3.1.1 Mobile Phones

We shall here focus solely on mobile phones, instead of PDAs or similar devices. The reasons are that phones are cheaper and therefore more widely used, and that the networking technologies on phones in general form a superset of those usually available in PDAs, with some of these technologies explicitly designed for mobility. Finally, mobile phones are in general weaker in capabilities than PDAs, so a system that works well on phones should also work well on PDAs.

The central processing unit (CPU) on a mobile phone is typically based on the ARM architecture due to the power efficiency of ARM CPUs. Clock frequencies on current models are usually around 200 MHz or slightly higher. Memory available to applications is usually a few megabytes, with recent high-end models having up to 20 megabytes.

Permanent storage on a mobile phone is typically based on flash memory. Usually a phone is shipped with a small memory card, possibly having up to 64 MiB available, but it is currently possible to acquire memory cards with 2 GiB of space, which is the limit supported by the old memory card interface (the newest models support even larger memory cards). High-end phones usually provide some form of a hierarchical file system view of the storage contents, but lower-end models just treat the storage as a simple key-value database.

The main concern with mobile phones is not their slower CPU speed or limited storage capabilities, but energy. As a phone needs to be mobile, its power source is a portable battery that can only contain a certain amount of energy. And unlike CPU speed and storage capacity, battery energy density has not followed an exponential growth curve [Paradiso and Starner, 2005].

When considered from the messaging point of view, the key question in energy consumption is how much energy do each of computation, storage use, and network use consume. This question has been studied in the context of security [Kangasharju et al., 2006; Karri and Mishra, 2003; Potlapally et al., 2006] and compression [Barr and Asanovic, 2006] where the algorithms require large amounts of CPU time, so it is necessary to know how much energy is consumed by the algorithms and how much by networking. The results indicate that even for modest amounts of communication, networking costs dominate, and that compression helps, but only when using simple efficient algorithms like DEFLATE [Deutsch, 1996a] instead of better but slower ones like bzip2[2] or Prediction by Partial Matching (PPM) [Cleary and Witten, 1984].

Mobile phones have a variety of different operating systems (OSs), the most common of which is Symbian[3]. Mobile ports of the desktop OSs Windows and Linux are also used on some phones, as well as manufacturer-specific ones. Each OS has a native programming language (C or C++) and a specific API for programmers to access the device's functionality.

In addition to the native language, most phones also include a Java Virtual Machine (JVM) for running Java programs. Different Java Micro Edition (JavaME) specifications define the capabilities of devices (called *configurations*) and the APIs available to programmers (called *profiles*). The most common profile is Mobile Information Device Profile (MIDP) [SM, 2002], which is based on the Connected Limited Device Configuration (CLDC) [Sun, 2003a]. A more expressive profile is Foundation Profile (FP) [Sun, 2002] based on the Connected Device Configuration (CDC) [Sun, 2001], found on some high-end devices. While the MIDP API is very different from the normal Java API, the FP contains basically the normal Java API at the level of version 1.1 with some extensions.

We chose to write our messaging system for the Java MIDP 2.0 API. This decision was made because programming in JavaME does not differ greatly from standard Java programming, whereas Symbian C++, the other option, includes a number of programming conventions and unusual classes because it lacks features

[2] http://www.bzip.org/
[3] http://www.symbian.com/

from standard C++ [Mikkonen, 2007]. While JavaME does not provide as much access to the phone's capabilities as Symbian C++, messaging middleware usually does not require the additional access, so MIDP 2.0 is sufficient[4].

3.1.2 Wireless Networks

One of the primary attractions of a mobile phone is its mobility, which allows it to be easily carried by the user and operated while mobile. Because of this, any networking technology used on the phone must be based on wireless operation. Users may accept a sporadic need to plug the phone in, e.g., for charging the battery, but they will not accept a requirement to be plugged in to connect to a network.

The main networking technology available on mobile phones is naturally the actual phone network. Nowadays, this is either a second generation (2G) digital technology such as Global System for Mobile communications (GSM) [Dettmer, 1991] or a third generation (3G) technology such as Universal Mobile Telecommunications System (UMTS) [O'Mahony, 1998]. While 3G networks provide better-quality connectivity, their availability is not yet as wide as that of 2G networks, so both kinds of technologies will likely be in use in the near future.

GSM itself is *circuit-switched* similarly to the landline phone network, i.e., whenever a call is placed, resources in the network itself are dedicated to maintaining a connection between the two endpoints. For data communication, *packet switching* is considered to be the superior approach [Roberts, 1978], in which each piece of data is split into packets, each routed to the destination independently of the others. This is an instance of the *end-to-end principle* [Saltzer et al., 1984], i.e., complex functionality should be placed at the edges of the network.

General Packet Radio Service (GPRS) [Cai and Goodman, 1997] is a packet-switching-based technology for data communication built on top of GSM. The main benefit to the user is that with GPRS the pricing structure of the operator is based on the amount of data

[4]MIDP 1.0 would not be sufficient due to its extremely limited networking capabilities.

Table 3.1: Theoretical data rates of mobile phone networks

Network	Max. data rate
GSM	57.6 kbps
GPRS	171.2 kbps
EDGE	474 kbps
UMTS	2000 kbps

sent and not on the time the connection is open. UMTS includes from the beginning both a circuit-switched and a packet-switched part to support both phone calls and data traffic.

Without switching to 3G technologies, the data rates in GSM networks can be increased by Enhanced Data rates for GSM Evolution (EDGE) [Furuskär et al., 1999]. Upgrading the core GSM network to support EDGE is not nearly as expensive as building a 3G infrastructure, so large operators usually support EDGE in their networks. Support in phones is less common, with only the higher-end models including EDGE support.

The maximum data rates of these technologies, including GSM for transmitting data over the circuit-switched network, are shown in Table 3.1. Note that these are theoretical maximums that are not achieved in regular use for a variety of causes, including user mobility and interference from other users located in the same cell.

The phone network is not the only available networking technology, especially on higher-end phones, even though it is the only one to support wide-area mobility. Commonly, phones support at least Bluetooth [BlueSIG, 2004] as well, but access to it may on some lower-end models be limited to existing applications without a programming interface. Nowadays, especially on phones intended for business use, Institute of Electrical and Electronic Engineers (IEEE) standard 802.11 [IEEE, 1999], also known as Wireless LAN (WLAN) or WiFi, is becoming available as well.

WLAN is commonly used in office environments to provide wireless connectivity to the Internet by placing a number of *base stations* around the office. In contrast to this *infrastructure* mode, it is also possible to use WLAN in an *ad-hoc* mode where the devices configure themselves with certain parameters that ensure they can all communicate with each other. While this mode can also be used

for network access if one device acts as a router for the others, it is more common to use it for less organized networking.

The attraction of WLAN is its high data rate. Of the two commonly-used versions, the older 802.11b provides a maximum data rate of 11 Mbps while the newer 802.11g provides 54 Mbps. However, to achieve these data rates in practice, the base station needs to be very close to the device, precluding wide-area mobility with high data rate.

Bluetooth is a very-short-range technology originally designed for replacing wired connections between a computer and its peripherals. However, it was soon applied to Personal Area Networks (PANs), that is, networks formed of a single person's computing devices. Because of its origins, Bluetooth includes the Service Discovery Protocol (SDP), which can be used to dynamically discover devices and the services they offer.

The main attraction of Bluetooth over other wireless technologies is its low energy consumption. A short-range transmission up to approximately 10 meters consumes only at most 2.5 mW of power. With a data rate of over 700 kbps this makes Bluetooth a very energy-efficient protocol for data transmission. According to Riva [2006], a request-response interaction with messages of size 100–200 bytes consumes approximately an order of magnitude less energy on Bluetooth than on ad-hoc WLAN, which in turn consumes an order of magnitude less than on UMTS. Note that the power consumption of WLAN is more than that of UMTS but the much smaller time spent in communication reduces the total energy consumption.

Two technologies are expected to feature prominently in the near future of mobile networking. Worldwide Interoperability for Microwave Access (WiMAX), IEEE standard 802.16 [IEEE, 2004b], is intended for Metropolitan Area Networks (MANs), i.e., city-sized networks, instead of office-sized networks like WLAN is. WiMAX promises a maximum data rate of 70 Mbps, though at its maximum range the data rate will be lower than that of 802.11b. A current problem with WiMAX deployment is the availability of radio spectrum in different parts of the world.

The other technology, expected to become Bluetooth's replacement for low-power short-range communication, is Ultra-Wideband (UWB) [Porcino and Hirt, 2003]. The major benefit of UWB

is that the use of a wide band for communication permits it to coexist with other radio systems on the same frequencies, so there is no need to allocate new areas of the spectrum just for UWB. However, an attempt to standardize a UWB technology at the IEEE has recently failed to reconcile two different approaches [Geer, 2006], making UWB in its current state unlikely to be adopted widely and across vendors.

3.2 Wireless Communication

The design of a communication protocol is highly dependent on the communication environment. As wireless networking and mobile computing have gained prominence only in the recent years, most existing protocols have been designed for a mostly-static network. However, the emergence of a new computing paradigm does not necessarily require a revolutionary change, but may be solvable through evolution.

There are two perspectives when designing protocols for mobile computing. One is that a fixed network such as the Internet is the main communication channel and mobile devices join this network through designated *access points*. In this style, the fixed network and the services provided by it form the *infrastructure* for the distributed applications. The other view, exemplified by research into *ad-hoc networks* [Perkins, 2001], is that networks can also consist solely of mobile devices. In this case, there is no fixed infrastructure, so service availability is not guaranteed, but depends on the proximity of devices providing those services.

3.2.1 Internet Protocols

The basis for Internet communication is its network layer protocol, Internet Protocol (IP) [Deering and Hinden, 1998; Postel, 1981a]. As the current trend even in the mobile world appears to be away from wireless-specific protocols and towards direct Internet connectivity, we will make the assumption that IP is the protocol used when communicating with larger communities.

According to some views, mobility should be the province of the network layer, as only that layer should care about the location

of communicating nodes. Thus, there exists Mobile IP [Johnson et al., 2004; Perkins, 2002] that provides a way to support mobile nodes with IP. This is done by each mobile node having a permanent *home address* where IP packets are sent. The home address is in reality monitored by the mobile node's *home agent* that will forward packets to the mobile node's current *care-of address*. The mobile node will simply send packets normally, but uses its home address as the sender IP address.

This method leads to a problem called *triangular routing*: since the mobile node is identified by its home address, all packets addressed to it must first go to the home agent and only from there to the mobile node. Mobile IP for IP version 6 includes *route optimization* support to mitigate this issue, but the fundamental problem is that on the Internet IP addresses are used as both identifiers of nodes (*identities*) as well as routing targets (*locators*).

This need to separate between identities and locators of network nodes has long been known (Saltzer [1993] provides a good overview). A recent proposal currently going through the IETF is HIP [Moskowitz and Nikander, 2006], which adds a layer between IP and the transport layer that provides cryptographic identifiers that serve as identities, thus leaving IP addresses only as locators. This solves problems in both mobility and multihoming without sacrificing security [Nikander et al., 2003].

On the Internet there are two main protocols on the transport layer: Transmission Control Protocol (TCP) [Postel, 1981b] and User Datagram Protocol (UDP) [Postel, 1980]. TCP is a reliable byte-stream protocol and UDP an unreliable datagram protocol. Of the two, TCP is much more commonly used due to its reliability, which removes the need for applications to implement their own flow control and retransmission policy. However, UDP is simpler and has a smaller per-packet overhead, which makes it more attractive for mobile devices.

TCP has been the subject of a multitude of performance measurements and enhancements. In wireless networks TCP has two main problems. One is that it assumes all packet losses to be due to network congestion and thus reduces its sending rate, whereas over a wireless link packet losses can often be due to corruption when no rate-limiting would be needed. The other is that the slow-start algorithm that TCP uses to probe the available bandwidth

begins by allowing only a very small number of sent but not yet acknowledged TCP segments, which leads to a long period of underutilization over high-latency networks.

The main ways to improve TCP performance while still honoring TCP's end-to-end semantics are to improve the algorithms for estimating network latency (to avoid spurious retransmissions) and bandwidth (to avoid needlessly long slow-start phase). Allman and Paxson [1999] evaluate a number of algorithms for both cases. More recently, Capone et al. [2004] have developed a bandwidth-estimation algorithm that is shown to improve on a normal TCP implementation without overestimating the bandwidth, unlike some other proposed improvements.

Of interest to our topic are measurements of TCP performance in mobile phone networks. An early work by Meyer [1999] uses simulation to determine the specific problems that TCP encounters over GPRS connections. According to Meyer, packet loss is a negligible problem, unlike over WLAN, due to retransmission functionality built into GPRS. In the simulation, TCP took on the order of 10 s to adjust its retransmission timer after a change in network conditions.

More recently, Benetazzo et al. [2003] have analyzed TCP performance over GPRS from the point of view of a network operator. In the measurements, a laboratory experiment in perfect conditions observed a mean round-trip time (RTT) of 1.5 s, while in a trace of real Internet traffic the RTT values were concentrated around 4 s, with a minor concentration around 6 s. Vacirca et al. [2005] provide similar measurements for both GPRS and UMTS, acquiring a median RTT of 1.5 s for GPRS and 350 ms for UMTS, though the UMTS distribution had a noticeably heavier tail.

3.2.2 Splitting Connections with Gateways

The end-to-end design principle [Saltzer et al., 1984] is usually considered to be a fundamental principle of the Internet, and the scalability of packet switching also seems to indicate that communication state should not be maintained in the network. However, with wireless communication there is a clear division of a connection between the wireless part and the wired part, and thus many people see benefits in the approach where a *gateway* (also called a

proxy or bridge) is placed at the edge of the wired network to split communication into a wireless part and a wired part.

Border et al. [2001] provide an overview of techniques used for improving TCP performance with gateways, with an emphasis on their effects on TCP end-to-end semantics. Border et al. focus on gateways that function at the transport and application layers, as the operation of gateways on lower levels is usually transparent to the transport layer. A transport-layer gateway can be implemented either as a split-connection gateway, which establishes separate TCP connections to both ends, or transparently, based only on modifying traffic on the connection. Application-layer gateways are by necessity split-connection.

Balakrishnan et al. [1997] evaluate three kinds of TCP improvements: end-to-end schemes that modify the TCP algorithms only on the end hosts, link-layer schemes that retransmits lost packets locally over the wireless link to avoid TCP noticing them, and split-connection schemes. The experiments were run on a two-hop network with the wireless link being WLAN, with the conclusion that a TCP-aware link-layer scheme performs best. Such an approach also has the benefit that it does not violate TCP end-to-end semantics, unlike most split-connection schemes.

Wireless Application Protocol (WAP) [WAP, 2001a] is a gateway-based architecture that uses its own protocols and formats over wireless links on all layers from the transport layer upwards. WAP is designed for mobile WWW usage, and originally used its own XML-based markup language, Wireless Markup Language (WML) [WAP, 2001b], but has now moved to a profile of Extensible Hypertext Markup Language (XHTML) [W3C, 2002d]. For transmission over wireless links, WML content is also compressed with a WAP-specific method [W3C, 1999].

WAP also includes a gateway-based security solution, Wireless Transport Layer Security (WTLS) [WAP, 2001c]. As the WTLS connection is formed between the mobile device and the WAP gateway, there is no end-to-end security, even if the connection from the gateway onwards were secured as well. Essentially, this requires a mobile device user to trust the gateway operator, since the gateway acts as a *man-in-the-middle* in any attempted secure communication with a server.

The WAP approach of replacing the whole protocol stack from

message syntax to the transport layer also drew some early criticism [Khare, 1999]. Nowadays, though, it appears that WWW access on mobile devices will use standard Internet protocols, with the protocols improved to take into account the variability of Internet connectivity. A common technique currently is to determine on the WWW server whether the client is a mobile device and send more appropriate content in that case.

3.2.3 Pervasive Computing

The Internet is not the only reason for communication. The recent rise of pervasive [Satyanarayanan, 2001] computing (similar concepts are ubiquitous [Weiser, 1993] and nomadic [Bagrodia et al., 1995] computing) has brought to the forefront issues with communicating with nearby devices.

The main concept of pervasive computing is that devices capable of computation are everywhere and they also possess communication capabilities. Thus a mobile device can, by communicating with other nearby devices, potentially be able both to understand its current environment better and to offload some computation to other devices. The vision is that these capabilities will lead to applications that adapt to the user and her situation for a smoother user experience.

The main technological enabler of pervasive computing is a variety of wireless networking technologies. Both Bluetooth and WLAN can be used to communicate with nearby devices, and mobile phone networks, or possibly WLAN, can be used to access remote servers on the fixed network. Thus, pervasive computing combines access to infrastructure services with the more dynamic current service environment of the mobile device.

Many pervasive computing applications function only in the one-hop radius, i.e., devices communicate only with devices that are in their immediate vicinity. However, forming the devices into actual networks might bring additional benefits, especially in heavily-trafficked areas. Such *mobile ad-hoc networks* [Perkins, 2001] are currently an active research topic, and many of the issues in messaging are essentially the same in both ad-hoc networks and wireless Internet connections.

We will not cover ad-hoc networks specifically here. Essentially,

we will assume point-to-point communication that can be either because of the link layer or because there is an underlying network layer that handles routing. This lets us avoid questions like routing [Perkins and Royer, 1999] and device mobility pattern, which can have a significant effect on communication [Camp et al., 2002].

3.3 Middleware and Mobility

Developing distributed applications is not an easy task, and writing one directly on top of the transport layer, say, using the sockets API [Stevens, 1997], requires a lot of care. Furthermore, most of the issues and required functionality are common across many such applications. Therefore, there exist a variety of *middleware* platforms [Aiken et al., 2000] to abstract away the common difficult parts, allowing developers to focus on application-specific functionality.

As middleware has been a popular concept in distributed applications, there already exist several deployed systems for the fixed network. Therefore an attractive option to include support for mobile computing in a middleware platform is to extend an existing system in some manner to provide it. Another alternative is to design the whole system from the beginning with the requirements of mobile computing in mind.

3.3.1 Mobility Extensions

Remote Procedure Call (RPC) [Birrell and Nelson, 1984] is a common style for many middleware APIs. In RPC communication with a remote system is made to look exactly the same as a local procedure call in the program. Usually this implies that the caller will block until the remote procedure returns with a value, as this is the semantics with the local procedure call. This synchronicity is usually not desired in wireless communication due to the fine granularity of procedure calls and high latencies of the networks.

There are ways to make RPC work asynchronously. With *promises* [Liskov and Shrira, 1988], the return value of an RPC invocation is only a stub object, and the call itself is performed asynchronously. When the application needs the return value, it will need

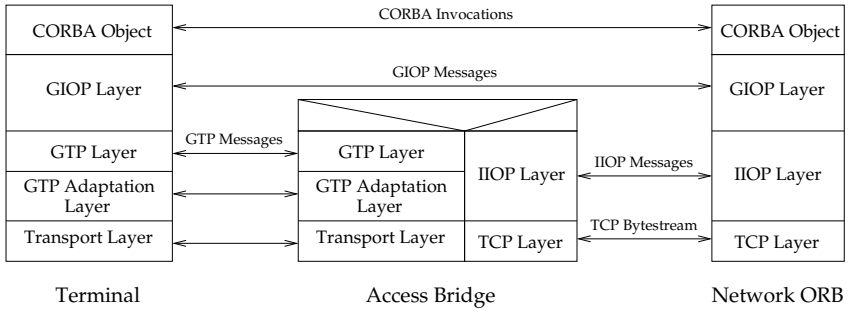


Figure 3.1: The Wireless CORBA protocol architecture

to explicitly *claim* the promise. The *futures* of Multilisp [Halstead, 1985] take this even further by not requiring an explicit claim operation at all, but Multilisp is a system for parallel programming, not distributed.

A well-known and widely-used middleware platform based on RPC semantics is Common Object Request Broker Architecture (CORBA) [OMG, 2004], a distributed object platform from the Object Management Group (OMG) that supports several programming languages. Nowadays, CORBA also supports asynchronous operation through its Messaging component. This supports both a promise-like *polling* model as well as a *callback* model where the application registers a callback object that the system invokes when the operation's result arrives.

There are two basic problems for a system based on RPC such as CORBA in mobile computing. First, wireless networks are not as efficient as wired ones, so invocation latency might not be sufficiently low. Second, if invocation targets reside on mobile nodes, there needs to be a way to locate them.

These problems are both solved, to some extent, by the Wireless CORBA specification [OMG, 2005]. It is based on the gateway architecture, shown in Figure 3.1, where the logical connection between the objects on the *mobile terminal* and fixed network is split by the *Access Bridge* in the middle. Wireless CORBA permits the use of more efficient protocols over the wireless link through *adaptation layers*, and it specifies four such layers, for each of TCP, UDP, WAP, and Bluetooth.

Locating objects on mobile nodes in Wireless CORBA is solved

in a manner similar to GSM. Namely, a mobile terminal may have a *Home Location Agent* in its administrative home network that keeps track of the Access Bridge through which the terminal is currently connected. Any locators for objects on the terminal point to the Home Location Agent, which redirects invocations to the proper Access Bridge. A terminal may also be *homeless*, in which case the locators point to an Access Bridge.

A well-known CORBA-like distributed object system is Java Remote Method Invocation (RMI) [Sun, 2004], which is specific to Java. Due to its similarities with CORBA, though, there exists an explicit interoperability specification using the Internet Inter-ORB Protocol (IIOP) of CORBA as the remote invocation protocol of RMI. RMI has also been extended by Wall and Cahill [2001] to support mobile objects. This approach is based on Architecture for Location Independent CORBA Environments (ALICE) [Haahr et al., 1999], which closely resembles the Wireless CORBA design.

A notable point with Java RMI is that the protocol requires a large amount of round trips, making it unsuitable for a high-latency network. Campadello et al. [2000] solve this issue with a gateway on the fixed network side that takes care of most of the communication needed in an RMI invocation, substantially reducing the number of round trips needed over the wireless link. Furthermore, Campadello et al. note that the original implementation wrote data one byte at a time, increasing the number of round trips even more due to TCP's slow start algorithm.

An alternative to RPC in middleware design that many consider better-suited for scalable distributed systems, is *publish/subscribe (P/S)* [Eugster et al., 2003]. P/S is a many-to-many asynchronous communication model that decouples senders from receivers through the P/S service API, shown in Figure 3.2. Here a *filter* is a way for subscribers to express their interest in a subset of messages, and is essentially a matching function over the set of possible messages. Some P/S systems support *advertisements* through which publishers can express a subset of messages in which all their published messages belong.

In principle, message targets in a P/S system are located based only on their subscriptions, so there should be no issue with mobility. However, the implementation of a wide-area P/S system is usually decentralized by using an overlay network of *brokers* [Car-

```
interface Publish {
    void publish (Message message);

    void advertise (Filter filter); // optional
    void unadvertise (Filter filter); // optional
}
```

```
interface Subscribe {
    void subscribe (Filter filter);
    void unsubscribe (Filter filter);
}
```

Figure 3.2: The generic publish/subscribe API

zaniga et al., 2001], so whenever a node moves, it may attach itself to a different broker, requiring an update to the routing tables.

A P/S system can be extended to support mobility using only the P/S API [Caporuscio et al., 2003], but this method incurs a large messaging cost. A better approach is to extend the brokers directly with special operations for a node to signal that it has moved. Usually in such treatments only subscriber mobility is considered [Fiege et al., 2003], as publisher mobility is considered to be rarer, and special support for it is needed only when advertisements are used. However, publisher mobility can be treated essentially in the same manner as subscriber mobility [Tarkoma and Kangasharju, 2007].

3.3.2 XML and Web Services

Web services can also be seen as a middleware platform. While their common usage does not require any specific consideration for mobility, as HTTP is a stateless protocol that uses IP addresses or Domain Name System (DNS) names for addressing, communication over wireless networks has an effect on their operation.

Originally, SOAP was positioned as a replacement for CORBA and Java RMI. Early measurements by Elfwing et al. [2002] show a performance ratio of 400 in favor of the CORBA implementation. The reasons for this are analyzed and optimizations proposed that, based on analytical considerations, would improve the ratio to 7.

Elfwing et al. note that, at the time, this was essentially the limit that would have been achievable with then-current technology.

Later work [Hericko et al., 2003; Juric et al., 2004] considers the serialization and deserialization of objects, to a binary form or to XML using Java Architecture for XML Binding (JAXB) [Sun, 2003b]. The conclusions are that, in total, SOAP invocations are an order of magnitude slower than RMI invocations. In addition, when considering purely the serialization and deserialization time, XML is again 5 to 10 times slower.

Laukkanen and Helin [2003] measure SOAP performance over a GSM network, and note that on a mobile phone over 90 % of a Web service invocation time is spent over the network. Tian et al. [2004] perform a similar experiment over a GPRS network, and conclude that generic compression provides a clear benefit despite the increased processing time on the mobile device. While Tian et al. do not analyze this benefit closely, a likely cause is the reduced number of TCP segments needed to transmit the messages.

All of these measurements treat SOAP and Web services as essentially an RPC system. This view has drawn criticism from the modern Web service community [Vogels, 2003], as Web services are nowadays seen as a messaging system instead of a replacement for distributed object systems. Indeed, by designing a remote processing application based on messaging, the performance of Web services can be increased to be better than naïve RMI [Cook and Barfield, 2006].

However, common practice in the distributed object community is to design interfaces by taking the network into account, similarly to what Cook and Barfield do for Web services. This breaks the abstraction of a remote object as a local object, since such interface design would not be typically practiced in local usage. And, despite appearances to the contrary, this has been long known in the distributed computing community [Waldo et al., 1994].

Apart from Web services, there are few examples of XML-based middleware. The usual concern, especially in the mobile community, is the increased processing and data transmission requirements incurred by XML compared to a special-purpose binary format. However, extensibility in the protocol is often beneficial, and that is rare with binary protocols, but is natively supported by XML, which makes an XML-compatible system attractive.

XMIDDLE [Mascolo et al., 2002] is a prominent example of an XML-based middleware platform. Its model consists of shared XML documents, accessed using standard XML technologies such as DOM and XPath. The model resembles tuple spaces [Gelernter, 1985] somewhat, except that the use of XML gives hierarchy to the data, making it more similar to file synchronizers [Balasubramaniam and Pierce, 1998; Lindholm, 2003], since the hierarchy of an XML document can be compared to a hierarchical file system.

3.3.3 Designing for Mobility

Designing a middleware platform from scratch for pervasive computing is very different from designing one for the fixed network. In a fixed network with stationary computers, it may be assumed that the network conditions remain stable and a computer's capabilities rarely change. In contrast, in pervasive computing, devices are mobile, network conditions change between fast access and no access, and the user's and device's contexts become significant in determining what actions are appropriate.

The key requirement, as identified by Raatikainen et al. [2002], is adaptability to the changing environment, consisting of the abilities to make available nearby peripherals, to divide execution of applications appropriately to computing systems in the environment, to tolerate non-availability of services, and to monitor the environment to discover new services and determine what computing capabilities are available. In addition to this, Raatikainen et al. also require a unified information base for each user, available everywhere independently of network access.

This adaptability to the changing situation is more often called *context-awareness* [Dey, 2001], and it has received a large amount of attention in the recent years. While each application has its own conception of what context is meaningful, the actual gathering and processing of context information is a more generic process, and therefore a variety of middleware platforms to support context-awareness have been developed.

It is naturally possible to design a middleware platform for mobile computing from the beginning while still retaining compatibility with an existing platform. Such an approach is provided by Yau and Karim [2004] whose RCSM is compatible with the CORBA

architecture, and uses an extension of CORBA's Interface Definition Language (IDL) to define context-sensitive objects. However, the communication protocol is not described, so it is uncertain whether CORBA's General Inter-ORB Protocol (GIOP) is used.

Hong and Landay [2001] propose an infrastructure approach to context-awareness, with the goal being independence of specific hardware and possibility for piecewise evolution of the system. This raises to the forefront issues such as data format and protocol design, where Hong and Landay see promise in XML and SOAP.

The *one.world* system [Grimm et al., 2004] follows three guiding principles for pervasive applications: context changes are explicitly visible, ad-hoc composition of components is encouraged, and data sharing is considered default behavior. Data in *one.world* are tuples, and nesting is permitted, which makes the data model hierarchical. Grimm et al. [2004] note that XML would be superior, but identify issues with the programming interfaces and verbosity that preclude XML usage in pervasive computing, the same concerns that have been a driving force of this dissertation.

MundoCore [Aitenbichler et al., 2005] is a pervasive computing middleware that provides both a CORBA-like distributed object programming interface as well as a P/S interface. Aitenbichler et al. [2005] note that many programmers gravitate towards the distributed object paradigm whereas the P/S paradigm is more often the better approach for pervasive computing applications. However, they note that debugging a P/S application is harder than debugging synchronized RPC.

3.4 Concluding Remarks

Modern mobile devices appear to be sufficiently powerful and versatile to participate properly in distributed computing. However, taking full advantage of the capabilities requires deep knowledge of what is available. A modern device supports several different kinds of networking technologies, but these all differ in range, data rate, latency, error profile, and cost, and what is appropriate to communicate through one might not be through another.

Selecting a proper communication protocol also requires care. In many current environments it may be assumed that IP connec-

tivity is available, and the Internet protocol stack can therefore act as a building block. But in many cases using the ubiquitous Internet protocols can add too much overhead, especially if there is little data to communicate.

Energy consumption is the most important concern for mobile devices, but taking it properly into account in application development is not an easy task. Both communication and local computation consume energy, but the ratios are very device- and network-dependent so that what is acceptable communication on one network and device can be completely unacceptable on another.

Development of distributed applications is often helped considerably by a suitable middleware platform. However, traditional middleware platforms can be poorly suited for mobile devices. In particular, the overarching concern for energy efficiency needs to permeate the design of the whole middleware [Riva and Kangasharju, 2007]. The middleware also needs to provide suitable programming abstractions so that the most efficient communication patterns are natural to use.

Part III

XML Messaging for Mobile Devices

CHAPTER 4

Messaging System and Protocol

> . . . , and from rubble may we *build*.

The fundamental design of our messaging system was based on the depiction of the components shown in Figure 1.2. However, while the Figure is useful as guidance, it depicts the components as if they depended precisely on the components below them. In the real design, such strict dependencies are not completely followed.

Due to the differing characteristics of wireless networks and mobile devices compared to fixed networks and stationary computers, communication designed for the mobile side often needs to take a different perspective from traditional distributed systems. This affects both the design of an application and the API that it uses for communication as well as the actual protocol that is used to transfer data from one device to another.

4.1 Messaging System Architecture

In designing the architecture, we endeavored to make the components shown in Figure 1.2 as independent from each other as possible. The design eventually resulted in four basic components,

each of which required new considerations to take into account the needs of mobile computing:

Messaging system The APIs and high-level functionality needed for applications to send and receive messages.

Communication protocol The protocol for sending and receiving serialized messages between devices.

XML API The API and system for processing XML data.

Serialization format The actual method of serializing XML data into bytes and parsing it back.

Each of these components was designed to be independently usable, with connections made through defining and implementing suitable interfaces, as befits an object-oriented design.

The way that these components integrate to form a complete messaging system is shown in Figure 4.1. In this Figure, the endpoints and messages are the API and functionality of the central messaging component. The communication protocol handles the connections between devices, and is made completely independent of the messages, in that it does not require messages to be XML, just byte sequences. XML processing is handled with the XAS system, potentially using the Xebu serialization format, described in chapters 5 and 6, respectively.

The messages passed around by the system are represented by the `Message` class. This is a class that knows how to serialize and parse itself as a SOAP message using the XML API. The actual serialization format can be selected at run time depending on the capabilities of the communicating peers. As the Figure shows, the message functionality is completely separated from the endpoints, addresses, and connections, by having the `Message` class implement an interface that allows for serializing messages[1].

The application data in the `Message` class is currently just a set of name-value pairs, with generic values. As the rest of the messaging system is not dependent on this particular class, other forms of messages would be possible to integrate into the system. Other

[1]This capability was also useful in testing the protocol, as it allowed a very simple alternative message class to be used.

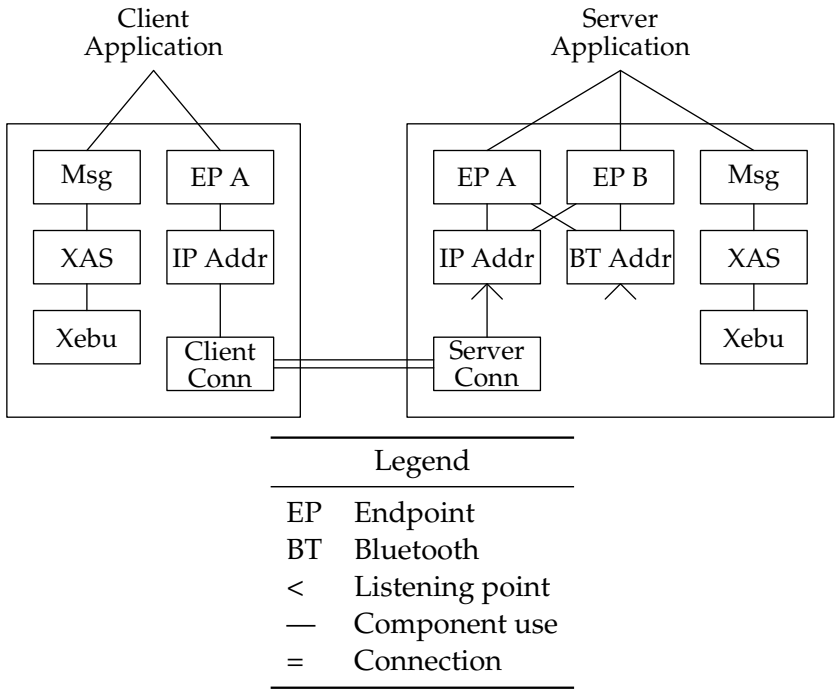


Figure 4.1: The messaging system architecture

data available to applications includes the source of the message, a unique identifier for the message, and, if the message is a response, the unique identifier of the message it is a response to.

4.1.1 Endpoints and Addresses

The basic concept in the actual messaging system API is the *endpoint*. Conceptually, an endpoint represents a target (or source) of messages. Endpoints are divided into two classes: *local* endpoints are listening points for receiver objects of the application and *remote* endpoints are targets to which messages can be sent. A message can also be sent to a local endpoint, which gets converted into a direct method invocation of the registered receiver object.

The contents of an endpoint are a set of *addresses* and a *target* name. The addresses serve to identify the host on which the endpoint resides and the target name identifies more specifically the actual endpoint on that host. This is similar to the HTTP Uniform Resource Locator (URL) syntax [Berners-Lee et al., 2005; Fielding

et al., 1999], with the addresses acting as a combination of the `http` scheme and host name, and the target name acting as the path. The differences are that endpoints allow several addresses and that the target names are not given any structure of their own.

The reason for an endpoint having many addresses is that modern mobile devices are able to use a variety of networking technologies, as described in section 3.1. Therefore a single target running on a single host can be reached through several different addresses. Furthermore, the environment of a mobile application will change so that different networking technologies are appropriate at different times. Allowing an endpoint to contain multiple addresses lets the messaging system select an appropriate technology without the application being aware of these changes. However, we have not yet implemented any method for automatically changing the network access technology in the messaging system.

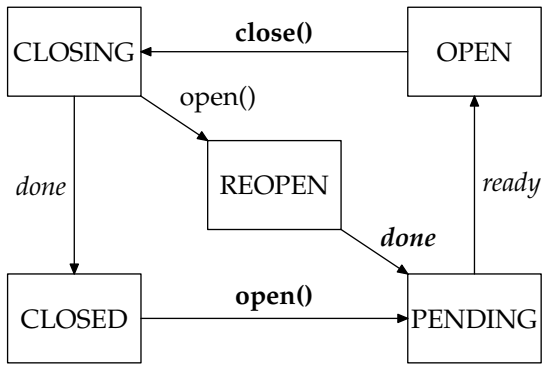
Like endpoints, addresses are divided into two classes, local and remote. Every address contained in an endpoint must be of the same type as the endpoint itself, and in fact, the class of the addresses is what determines the class of the endpoint.

4.1.2 Connections for Messaging

To send messages, the system must first open a *messaging connection*, which has `open()` and `close()` methods to manipulate the connection state. As latencies in wireless networks can be high, all connection manipulation is handled asynchronously, using a dedicated thread for communication with the remote node.

The state diagram for a connection is shown in Figure 4.2. In normal use, both the `open()` and `close()` methods will trigger sending data to the other side to indicate what is being done to the connection, with the connection state changing as appropriate. When the response is received, the state is again transitioned, with the OPEN state being the one where messages can be sent. If a connection is being closed, an `open()` call at that time transitions the state to the REOPEN state, in which the response to the close message triggers an immediate sending of the open message, transitioning the state to PENDING.

Applications do not normally see this connection management at all. Currently, the endpoint interface provides applications the



Parentheses () indicate local method calls, **bold** indicates sending data, *italic* indicates receiving data

Figure 4.2: Connection states and transitions

possibility to close the endpoint, which will call `close()` on all messaging connections of the endpoint. This is intended as a resource-saving device, allowing applications to discard endpoints they are no longer using. In the future, we would also like to include functionality for automatically determining when messaging connections should be closed.

The main way that applications interact with the messaging system is through invoking an endpoint object's `send()` method. This will pick a suitable address from the endpoint, and if needed, open a connection to that address. If the connection was not yet in the OPEN state, the message will be buffered until the response to the open message comes back from the other side.

4.2 Application Communication Model

The differences required for mobile computing are not confined to just the protocol level, below message syntax. Rather, they permeate the whole design of the application. This is most evident in client-server systems, where the most common current pattern is the *synchronous request-response (SRR)*, also the model of HTTP.

The problem with the SRR pattern is that the client application will block waiting for the server's response. On a mobile phone network, where application-level latency can reach several

seconds, this wastes time that the application could use for other processing. Therefore a better pattern for this would be the *asynchronous request-response (ARR)*, to allow the client application to continue processing while waiting for the server's response.

However, switching to a different communication pattern is not achieved by simply changing SRR invocations to ARR invocations. Usually SRR applications are designed to take immediate advantage of the received response. Converting such an application directly to ARR would just entail that the application would still block waiting for the server's response immediately after sending the request. Therefore the applications need to be redesigned and not just brought over directly from the fixed network side.

4.2.1 Message Exchange Patterns

The formalization of a distributed application's communication model is called a Message Exchange Pattern (MEP), and there are several in existence, each best suited for certain applications. As we focus on a basic messaging system, we consider only point-to-point MEPs, not more complex ones such as P/S that we mentioned in subsection 3.3.1.

A few basic MEPs are shown in Figure 4.3. We do not separate the application-level programming model for these into synchronous and asynchronous yet. The simplest MEP is the *one-way* pattern where the sender sends one message to the receiver without waiting for a response. This is also a general pattern, in that it is possible to implement other MEPs on top of it. However, we consider it the responsibility of the messaging system to provide the most common MEPs to aid in application development.

The one-way pattern also invites certain extensions. As there is no communication in the other direction, the sender cannot know, in the basic form of the pattern, whether the message ever reached the receiver. Because of this, it is useful for the messaging system to provide delivery guarantees even for the one-way pattern. These guarantees can range from always providing exactly-once semantics to providing a variety of options to the level of reliability desired, such as in CORBA Messaging [OMG, 2004].

The simplest two-way pattern is the *request-response*. In this pattern, the sender sends a single request to the receiver, which sends

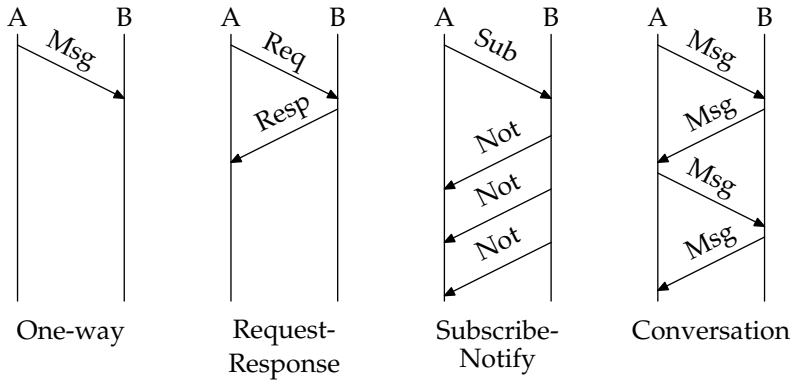


Figure 4.3: Different Message Exchange Patterns

a single response back. In theory, reliability for this pattern can be provided by the application, since the response indicates that the receiver processed the message. However, it is more cost-effective to provide reliability guarantees at the messaging system level, since detection of duplicate messages caused by retransmissions is not something the application should concern itself with.

Two other commonly-seen two-way patterns are *subscribe-notify* and *conversation*. In subscribe-notify the first message is a *subscription* message. After this, whenever a condition expressed in the subscription message is triggered, a *notification* message is sent to the originator. Conversation is an extension of request-response where every message can be responded to.

While the one-way pattern is fundamentally asynchronous, request-response can be either synchronous, where the sender will have to block waiting for a response, or asynchronous, where the sender is notified of a response in some other manner. Thus, its synchronicity depends on the system API.

4.2.2 Messaging System API

A rough interface to a messaging system that supports both synchronous and asynchronous styles is shown in Figure 4.4. The `MessagePromise` interface represents a promise that acts as a container for a not-yet-received response. The promise can be queried as to whether it already contains the response (to avoid blocking),

```
interface MessagePromise {
    boolean isReady ();
    Message claim ();
}
```

```
interface Receiver {
    void receive (Message
        response);
}
```

```
interface MessageSystem {
    Message requestS (Message request); /*
        synchronous */
    MessagePromise requestP (Message request); /*
        asynchronous poll */
    void requestC (Message request, Receiver
        receiver); /* asynchronous callback */
}
```

Figure 4.4: Different request-response APIs

and the response can be extracted, possibly blocking until it is available, by claiming the promise. The `Receiver` interface acts as a general receiver for messages of any kind.

The `MessageSystem` interface in Figure 4.4 has one method for synchronous invocation and two for asynchronous, based on *polling* and *callbacks*. The synchronous `requestS` method will return the response as its value, blocking until it arrives. The asynchronous polling method `requestP` returns a promise that can be later claimed by the sender. Finally, the asynchronous callback method `requestC` requires the sender to provide an implementation of the `Receiver` interface. When the messaging system receives the response, it will invoke this callback object's `receive` method.

Of these styles, the asynchronous polling style can be implemented using the asynchronous callback style, and the synchronous style using the asynchronous polling style. Namely, the `MessagePromise` object that is returned by `requestP` is made to implement the `Receiver` interface and is registered as the callback object in a `requestC` invocation. The `requestS` method is implemented by invoking `requestP` and immediately invoking the returned promise's `claim` method. Thus we can say that the asynchronous callback style is the most general interface of these three.

To implement the conversation pattern using the interfaces of Figure 4.4, we change the `receive` method of the `Receiver` inter-

face by making it return a `Message` object. The semantics of this return value is that if it is not `null`, it will be returned as a response to the received message. The original receiver can also be implemented using the `Receiver` interface, unifying the two sides and allowing a natural implementation of the conversation pattern.

The subscribe-notify pattern cannot be directly implemented by just the messaging system APIs. The reason for this is that the sending times of the notify messages sent by the notifier are not controlled by the messaging system. Therefore to support this pattern in the messaging system the endpoint of the sender can be extracted from a received `Message` object. It is naturally the application's responsibility to perform this extraction and determine when to send notify messages to it.

4.3 Protocol Requirements

Improving the processing of application messages helps only to the extent that processing is the bottleneck of the system. A messaging system needs to consider also the protocol used for transferring messages. We noted originally [Kangasharju et al., 2003] that the default manner of using HTTP in conjunction with SOAP is significantly suboptimal, especially in wireless networks. For this reason our messaging system also includes an improved protocol.

The basic unit in the protocol should be the message, and not a stream of bytes or characters. We made the decision that the protocol should not provide the needed MEPs itself, but these should be implemented in the messaging system using SOAP headers, as is done in WS-Addressing [W3C, 2006d], though we refuse to use URLs due to their prohibitive length. Therefore the basic protocol should only provide one-way messaging as a primitive.

If the protocol is connection-oriented, this connection should not limit which side can send messages at which time. While a connection will always have client and server roles based on who initiated the connection, these roles should not reflect on the communication. TCP is an example that satisfies this requirement whereas the request-response interaction of HTTP is not directly suitable.

The messaging system will also need some reliability guarantees from the protocol. At-most-once semantics is clearly desir-

able. This can further be extended to exactly-once semantics when we can assume that connectivity for sending a message is available infinitely often. Messages should not be garbled in transit, but for this it should be sufficient to rely on lower layers. Ordered delivery is a nice feature to have, especially considering that messages will be sent asynchronously before replies to previous messages have been received. However, messaging itself does not place this as a requirement, so it can be dropped if need be.

It is noteworthy that normal reliable transport protocols such as TCP do not provide sufficient reliability in a mobile environment. When a device moves to a new access network, it will also acquire a new network address, which breaks all existing TCP connections, meaning that messages may be lost if they are sent during mobility. Reliability options in this case include unchanging addresses at the network layer through Mobile IP or implementing the needed reliability in the protocol itself.

As noted in section 3.1, modern mobile devices have a variety of networking technologies available to them. While some, such as GPRS and WLAN, are designed to support Internet connectivity, others, such as Bluetooth, are only usable for short-range peer-to-peer communication. Despite the current trend of everything moving to run on top of IP, there is still a need for the messaging protocol to support a number of *underlying protocols*.

Furthermore, even if the whole network stack runs on top of IP with only one protocol at each layer, there is still a need to adapt to the current network. For instance, with a WLAN connection to the Internet, bandwidth, latency, and monetary cost are rarely an issue, whereas they are with GPRS and UMTS, and with Bluetooth even the set of available peers is restricted. The pervasive applications of the future will require access to such information to better configure their behavior to be appropriate to the current context, and therefore the protocol layer will need to provide access to some of the network characteristics.

4.4 Basic Protocol

Our implemented protocol, which we call *Abstract Mobile Message Exchange (AMME)*, consists of two layers, the *Transfer layer* and the

```
Content-Type: application/x-ebu
Content-Length: 1245
...

<body data>
```

Figure 4.5: The BEEP message syntax

Mobility layer. The Transfer layer provides a very simple uniform messaging semantics, and each underlying protocol has a separate Transfer layer implementation. The Mobility layer is composed of modules that can be independently composed to provide features that the underlying protocol lacks. Since the Transfer layer provides a common interface and unified semantics, the modules of the Mobility layer are independent of any underlying protocol.

The basic purpose of a messaging protocol is to be sufficiently versatile to accommodate a variety of messaging styles. As we noted in section 4.2, the callback-style interface of our messaging system directly supports a variety of MEPs. Implementing these should not be too contrived using whatever protocol is used for message transfer.

4.4.1 Original Protocol Design

The initial version of our system was implemented and tested on laptop computers, to better gauge the usefulness of the ideas. The main intent was to use existing protocols as much as possible to determine the most useful ways to implement functionality required for mobile messaging.

The original version of AMME was built on top of Blocks Extensible Exchange Protocol (BEEP) [Rose, 2001]. BEEP is a generic application-layer protocol designed for the creation of more specific protocols. A BEEP message consists of headers and a body, similarly to HTTP, as illustrated in Figure 4.5. In the BEEP communication model, a *session* is opened between two peers. Such a session is further divided into *channels* for actual communication, each of which can be opened by either side of the connection.

At the time we chose to use BEEP for our initial exploration,

there seemed to be some interest in it, including a standardized SOAP binding [O'Tuathail and Rose, 2002]. However, we did not find implementations that would have been truly robust, and interest in BEEP seems to have waned. A possible cause is its complexity: a generic protocol is by necessity more complex than each application-specific protocol, even if adopting the generic protocol would save work in the long run. This waned interest is somewhat regrettable, since in our opinion BEEP is a well-designed protocol with many applications.

Of course, BEEP was not usable in the final version of our messaging system, since none of the available implementations were runnable on mobile phones. Furthermore, we originally targeted MIDP version 1.0, which only supports HTTP for communication, so we needed a solution usable for that protocol as well.

However, BEEP does have interesting features, which we found useful and adopted for our design. The first of these is the splitting of a session into multiple channels, which can be used to provide multiplexing in the protocol itself. The second was that the name-value headers provide a generic metadata mechanism. We used this in our original protocol to provide features not available in BEEP and our current design improves upon this idea.

One of the purposes to which generic metadata is put to in both BEEP and HTTP is *content negotiation*. This refers to each message carrying with it its Internet media type so that the receiver can understand how to process it, and to including the acceptable media types in connection initialization messages. In our system we envision the message to be an abstract representation of an XML document that can have several different concrete representations. Based on the media type carried in the metadata the system can then direct incoming messages to the correct parsers, which we consider to include both XML and binary format parsers.

4.4.2 Transfer Layer Semantics

As one requirement for the protocol was the ability to accommodate different underlying protocols, we chose to design the Transfer layer in an abstract manner. That is, there is a common API and certain behavioral guarantees associated with each operation in the API. This needs to be sufficient to implement the Mobil-

ity layer in a generic manner, requiring only an implementation of the Transfer layer, called a Transfer layer *mapping*, when a new underlying protocol needs to be supported. By keeping the Transfer layer semantics sufficiently unconstraining, the work required for this can be kept to a minimum.

On an abstract level, we designed the message syntax of AMME to be the same as that of HTTP and BEEP: a header consisting of name-value pairs and a body of opaque binary data. However, in AMME this is merely an abstract model, with the header names serving as unique and memorable identifiers, not necessarily as something that would get sent in the message. Noting that in a typical application, the size of the HTTP header part can reach several kilobytes, we came to the obvious conclusion that the metadata in the protocol needs to be represented compactly.

The Transfer layer is based on point-to-point *connections*. Inspired by BEEP, each connection is divided into *pipes* that carry the actual messages. In principle there can be an arbitrary number of pipes, and they do not need to be reflected in any manner in a Transfer layer mapping, allowing efficient multiplexing of a connection. While there are client and server roles, this is visible only when opening a connection, and afterwards either party may send messages on any pipe at any time.

The communication abstraction provided by the Transfer layer is extremely simple. Each pipe is *full-duplex*, as noted above, *one-way*, meaning that there are no response or acknowledgement messages, and *unreliable*, meaning that messages may be lost or may arrive out of order. Messages are assumed to not be corrupted during transit, so if corruption is possible with the underlying protocol, the mapping must protect against it.

The Transfer layer interface is shown in Figure 4.6. A `TransferConnection` is used to send messages. Headers can be sent either individually through the two header methods or a `TransferHeader` can be requested from the connection with the `newHeader` method, filled in, and passed at message sending time. For receiving messages, the application must implement the `TransferAcceptor` interface. When a message is received, the system will first call the two header methods to pass all the headers of the message, and then the `receive` method for the actual message. This style was designed to support streaming at the receiver end.

```
interface TransferConnection {
    void send (TransferHeader header, byte[]
       message);
    TransferHeader newHeader ();
    void numberHeader (int type, long value);
    void generalHeader (int type, Object value);
}
```

```
interface TransferAcceptor {
    void receive (TransferConnection connection,
       InputStream in);
    void numberHeader (TransferConnection connection,
        int type, long value);
    void generalHeader (TransferConnection
       connection, int type, Object value);
}
```

Figure 4.6: The Transfer layer interface

Besides implementing the interfaces of Figure 4.6, a Transfer layer mapping also needs to provide a header mapping. On the abstract level, each header is just an integer in the interface, with an encoded type: one of number, string, array of numbers, or array of strings. Number headers are treated specially in the interface to avoid conversion between primitive values and objects in Java. Currently, the header mappings in our Transfer layer mappings are static, in that each mapping knows all the possible header values and contains predetermined values for them.

4.4.3 Transfer Layer Mappings

Our system includes three Transfer layer mappings, for each of TCP, Bluetooth, and HTTP. Of these, we consider the TCP mapping to be the best choice if possible, with Bluetooth used for ad-hoc communication when an Internet protocol stack is not available, and HTTP used when the network infrastructure does not allow anything else.

We built the Bluetooth mapping on top of the RFCOMM protocol, which provides a stream interface. This allows the Bluetooth mapping to share code with the TCP mapping, as TCP is also a

stream protocol. We needed to include some special processing in the Bluetooth mapping, isolated in specific stream classes that are passed to the generic stream mapping implementation.

In the stream mapping, the message header is represented fully in binary. The `type` argument of the header methods of Figure 4.6 is simply output as such to the stream. The built-in typing in the headers is sufficient for the receiving end to decode the header even if it did not recognize the identifier, which allows for future extensibility, since all unrecognized headers are simply passed on to the higher layers of processing.

The HTTP mapping is much more complex than the stream mapping, since HTTP's single-request-response model is not very suitable for general messaging. While real-world considerations necessitate supporting HTTP, and the mapping's design has some interesting points, we do not consider HTTP to be very useful for general messaging.

In the HTTP mapping, headers are mapped directly to HTTP headers, with numbers encoded as strings with possible separators. We chose this representation to be compatible with existing HTTP systems. However, we do not use the full names of the headers but map each header name to a two-character identifier that is still unique and memorable without excessively adding to the size of a message. This mapping is not extensible in the same way as the stream mapping as unknown header types do not currently have a representation.

In the HTTP mapping, a Transfer layer connection is initialized by sending an HTTP request to a stable URL of the server. The server will then generate a unique URL for the opened connection, which it sends back to the client in an HTTP response. All further HTTP requests from the client are sent using this unique URL.

The connection itself is implemented by running two threads on the client side, the *message thread* and the *token thread*, illustrated in Figure 4.7. At the beginning of the connection, the token thread sends an empty HTTP request, a *token*, to the other side. Whenever the other side has a message to send, it can send it as a response to the token request. When the client has a message to send, it is sent in an HTTP request by the message thread. The response to such a message will then be an empty message as well, as the Transfer layer has no concept of responses or acknowledgements.

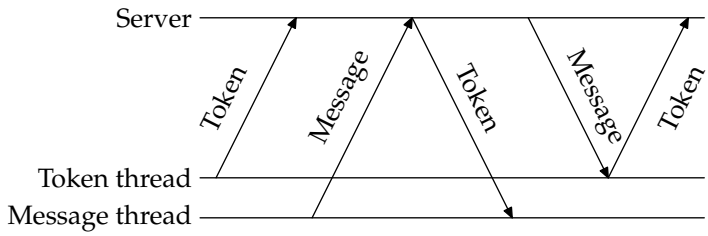


Figure 4.7: Token and data messages in HTTP Transfer mapping

Table 4.1: Code line counts for the protocol components

Component	Lines
Transfer	40
Stream	354
TCP	99
Bluetooth	307
HTTP	422
Mobility	522

4.5 Protocol Extension Modules

Apart from the most basic functionality, the Transfer layer does not satisfy the requirements we placed on the protocol in section 4.3. As we noted, this was a design decision made to keep the amount of code required for each underlying protocol to the minimum. Instead, the requirements are implemented by the Mobility layer, which is independent of the underlying protocol due to the uniform interface and guarantees offered by the Transfer layer.

The decision to separate the Mobility layer, and to generify the stream part of the stream-based protocols, is validated by the code line counts shown in Table 4.1 (computed using the `sloccount` tool by Wheeler [2002]). Here the Transfer component denotes the APIs provided by the Transfer layer, and the Stream component is the common stream-based code used by both the TCP and Bluetooth mappings. The large size of the Bluetooth component is caused by its inclusion of dynamic discovery of devices.

To implement the needed functionality, the Mobility layer defines a number of headers for the messages. Furthermore, we

noted that most of the required pieces of functionality are reasonably independent of one another, so we split the Mobility layer into a number of *modules*. Each module defines a few headers, which are inserted by the sender component of the module, and read and interpreted by the receiver component of the module.

We originally believed that the module interface could be made generic, with the possibility of composing only the modules required by each Transfer layer mapping. For instance, if the underlying protocol were to provide truly persistent connections and addressing, the persistence module would not be needed. However, the functionality of the modules is not completely transparent, so in the end it was necessary to compose the modules in quite a precise order and manner. For some modules it might be possible to enable or disable them at will, but we did not explore this further.

4.5.1 Sequence Number Module

Since the Transfer layer does not provide any guarantees for reliable or ordered delivery of messages, we needed to implement a sequence numbering system. Such a system cannot be avoided even if the underlying protocol provides reliability, like, e.g., TCP does. This is because we also target mobile clients, and during mobility TCP connections will break. Any new connection established afterwards will not share the old connection, so TCP's reliability does not extend to such situations.

When using this module every message contains a SEQUENCE-NUMBER header, the value of which starts at 0 and increases by one for each message. Acknowledgements are of two kinds. A CONSECUTIVE-ACKNOWLEDGEMENT header's value is a single number indicating that all messages up to that sequence number have been received (and can therefore be deleted from any buffers). An INDIVIDUAL-ACKNOWLEDGEMENT header contains a list of sequence numbers for messages that have arrived out of sequence.

An individual acknowledgement typically indicates lost messages, so upon reception the receiver should resend all unacknowledged messages. Since the system only has a single thread for each connection, and uses queues for message sending, it should not be possible for messages to arrive out of order, but we emphasize that the Transfer layer need not guarantee this. For instance, we exper-

imented with having more than one message and token thread in the HTTP mapping and we did see out-of-order arrival then.

The Mobility layer also passes all received messages to the application in the order of their sequence numbers. The messaging system preserves this order, giving applications a guarantee of ordered delivery. Furthermore, the messaging system guarantees that response messages will be delivered back in the same order as the requests came, as long as the application processes and responds to the messages in a single-threaded fashion.

4.5.2 Connection Persistence Module

The Mobility layer also provides more direct support for mobility with persistent connections. On first opening of a connection the server will return a CONNECTION-IDENTIFIER header containing a unique identifier for the connection. If the client later wishes to continue a previous connection, it will send the connection's identifier in a CONNECTION-IDENTIFIER header when reopening the connection. Thus the connection can be logically continued even across mobility.

Naturally the server cannot remember every connection from every client indefinitely. Therefore the server's response also includes a CONNECTION-PRESERVE header, giving the time that the server is willing to retain the state after a connection has been dropped. The client can also provide this header to request a certain value, but the server's provided value is authoritative.

This feature is also useful to applications, for two reasons. The first is that applications, both at the client and server, will see a unique persistent identifier for any communicating peer, and can use this identifier instead of using an address, which may change when the other end is mobile. The second benefit is that we are able to retain the state of our format processor, which means the messages are smaller, even when connections break.

4.5.3 Message Compaction Modules

The Mobility layer also contains modules to reduce the amount of data that is transmitted. The most significant of these in high-frequency messaging is the ability to bundle several application

messages into a single AMME message. To do this, the Mobility layer can insert a MESSAGE-BUNDLE header, the value of which is a list of numbers. Each of these numbers is a byte-based index into the message body, and indicates where a new application message starts. These individual messages are then separated by the receiver and passed to the application as individual messages.

Another feature is the ability to specify types of messages and to allow default values to be omitted. At the Transfer connection opening, both parties will send, in an ACCEPT-TYPE header, a list of message types that they understand. The intent is that these types are alternate ways to serialize the same message. Later, if a message's type is the same as the first one in the receiver's understood list, the CONTENT-TYPE header marking the type can be omitted; the receiver will then default to its preferred type.

4.5.4 Measuring Round-Trip Time

As we have noted, applications need reliable information on the device's context to be able to adapt their behavior, and the messaging system is best placed to determine the characteristics of the current network. Thus far, we have implemented a module to measure RTT using a method that does not require ping messages, i.e., immediate responses to specific messages. Our method is similar to PinPoint [Youssef et al., 2006], except that our system requires the clocks on both sides to advance at the same rate.

At connection opening both parties will inform the other of their local time in milliseconds in an OWN-TIMESTAMP header. After that, each message may contain a new OWN-TIMESTAMP header updating this value, and a PEER-TIMESTAMP header, giving the time that the sender believes the receiver to have. By subtracting the received PEER-TIMESTAMP value from its actual time, the receiver will get an estimate of the RTT.

The precise formulas used in calculating timestamps are

$$\begin{aligned} sot &= ct \\ spt &= not + (ct - npt) \\ rtt &= ct - rpt \\ not &= rot - (ct - npt) \end{aligned}$$

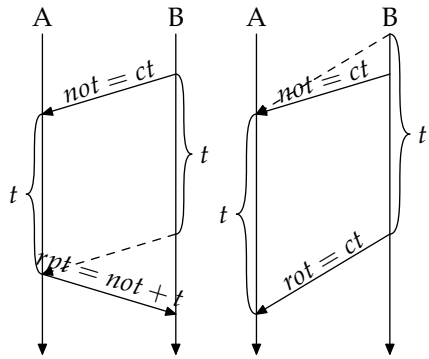


Figure 4.8: Computing round-trip times in AMME

where *sot*, *spt*, *not*, *npt*, *rot*, *rpt*, *ct*, and *rtt* denote, respectively, the OWN-TIMESTAMP and PEER-TIMESTAMP values to send in a message, the original received OWN-TIMESTAMP value and the local time at that value's reception, the OWN-TIMESTAMP and PEER-TIMESTAMP values received in a message, the current time, and the calculated RTT.

An illustration of how these equations work to compute the RTT is given in Figure 4.8(a). Here we see B sending the original message at time $not = ct$. After time t has passed, A sends a message (this can be independent of the message sent by B), containing the PEER-TIMESTAMP value of $spt = not + t$. When this message arrives at B, the time that has actually passed from *not* is t plus the RTT. Hence a subtraction of the received value from the current time gives the RTT.

RTT consists of two individual times: the time for a message to reach the recipient and the time for the recipient's response to come back. In the calculation of Figure 4.8(a) the first of these components will always be the time that the initial message took. Since changing network conditions, especially during mobility, will affect RTT, the OWN-TIMESTAMP value can be updated to provide more current information.

Figure 4.8(b) shows how this works. The second message sent by B contains its current time in an OWN-TIMESTAMP header. A

will then recompute its new *not* value to be $rot - t$. The new value of *not* will affect the later calculations so that the initial message is perceived to have taken the time that the latest message containing an OWN-TIMESTAMP header took.

4.6 Summary

The design of our messaging system follows the abstract model of Figure 1.2 and good object-oriented design principles by separating concerns. This is not the best design style for mobile devices [Hartikainen et al., 2006], but for a prototype research implementation we consider it suitable, especially as one purpose is to experiment with different implementations of the components.

The messaging system itself is a simple implementation, derived in a straightforward manner from the requirements of mobile messaging. Its most distinguishing feature, the use of the endpoint as an address container, is similar to how Interoperable Object References (IORs) in CORBA work, except that the endpoints contain much less data and are not intended to be transferable.

As mentioned above, many of the ideas in the protocol layer come from BEEP, namely the ability to use a single physical connection for multiple logical connections and the header-body division in messages. At the time, the method for measuring RTT was novel (it first appeared at a student workshop [Kangasharju, 2004]), but as mentioned, the PinPoint system improves on it by not even requiring the clocks to advance at the same rate.

We found the module-based system for implementing extensions very comfortable compared to implementing them all in a single class. The explicit design of the interfaces to support efficient message processing made this approach feasible even from the efficiency perspective. We believe that at least some useful functionality could also be implemented as generic modules; our current modules need very close coupling to the message sending layer to function properly.

CHAPTER 5

XML Processing with XAS

> Let's hear it for the vague blur!

The traditional view of XML comes from its roots as a document markup language. According to this *document-oriented* view, an XML document is mostly composed of text, is intended to be read and modified by people and therefore has descriptive names, and element content is often mixed. Furthermore, XML is processed by applications as XML, and commonly the whole document, the size of which can be quite large, is kept in memory.

The emerging *data-oriented* view that we are concerned with treats XML as a standard data interchange format. The actual data is kept in an application-specific form inside the system, and therefore XML is visible only to programs, not people, though the text-based nature of XML helps debugging the applications. Elements are typically rigidly structured, and contain either only other elements or a stringified representation of some programming language data value. Traditional XML processing APIs are not always very well suitable for processing such documents easily and efficiently, and therefore part of our work focused on designing an XML processing system and API that would support the data-oriented view, including the possibility to use alternate serialization formats conveniently.

5.1 The Basic XAS API

Our original XML API [Kangasharju and Lindholm, 2005] was essentially a small extension to the XmlPull API to support typed content better. However, experience with this API [Kangasharju et al., 2006; Lindholm et al., 2005] revealed that it was limited both in extensibility and in providing advanced functionality for efficient XML processing in unconventional ways. We therefore decided to redesign the whole processing system to better support our target applications.

Our new design was partially inspired by the design principles of XOM [Harold, 2006], though speed and memory usage need to be accorded more importance with mobile devices. In particular, since we already had experience with XML applications, the design principle of implementing only what is required by some application was viable to adopt. Also, the idea of enforcing correctness as required by XML already at construction time makes the internal code much simpler, as the necessary constraints do not need to be rechecked.

We retained the name XAS for the new system as well[1]. From the beginning XAS was intended to be an integral part of Fuego middleware, through which all XML processing would take place. Thus the design would need to take into account not only the needs of the messaging system, but also those of the synchronization system, which processes XML as XML and does not convert it to an application-specific format. Our experiences with the earlier XAS API were valuable in understanding the precise needs of all the components of Fuego middleware.

5.1.1 Item Sequences

The basic concept of the XAS system is the *item*, which corresponds to an atomic piece of XML syntax. XAS comes with a few items for representing XML in the normal manner. These *core items* are shown in Table 5.1, including how each would be represented in an XML document. The `ED` item represents the end of the document and is introduced for symmetry with the `SD` item. The `T` item

[1] XAS used to be an acronym, but is no longer. It is not simply SAX reversed.

Table 5.1: The core item types of XAS

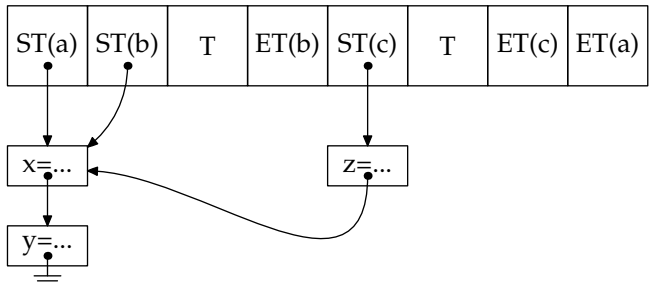
Name	Item	XML
Start document	SD	<?xml version="1.0"?>
End document	ED	
Document type	DTD	<!DOCTYPE foo SYSTEM "...">
Start tag	ST	<p:foo xmlns:p="..." at="...">
End tag	ET	</p:foo>
Text	T	...
Entity reference	ER	&ent;
Processing instruction	PI	<?bar baz?>
Comment	C	<!- ... ->

Table 5.2: Content of XAS core items

Item	Content
SD	empty
ED	empty
DTD	public and system identifier, text
ST	qname, prefix map, attributes, parent
ET	qname
T	text
ER	name
PI	target, text
C	text

represents normal text content. The DTD item represents the full DTD as text, unlike in, say SAX, where the content of a DTD is fully parsed. We do not consider it useful for an XML document to have an embedded schema, but would prefer all schemas to be external, so XAS does not support DTDs very well.

The content of each of these items, shown in Table 5.2, is derived from the correspondence with XML shown in Table 5.1. The qname type represents an XML name, i.e., a pair consisting of a namespace URI and a local name. The parent of an ST item is the ST item for that element's parent in the XML document. The names of the other types are based on the XML specification [W3C, 2006a], and should be mostly self-explanatory.




```
<a xmlns:y='...' xmlns:x='...'>
  <b>...</b>
  <c xmlns:z='...'>...</c>
</a>
```

Figure 5.1: Representation of namespace prefix mappings

The `ST` item contains both a mapping between namespace URIs and prefixes as well as the attributes of that element. The prefix map is semantically a *functional map* [Okasaki, 1999] in that it is derived from the parent's prefix map by adding the prefix mappings of the `ST` item, but the parent's prefix map must also remain intact. As we expect there to be only a few prefix mappings in a single start tag, we use a simple linked list as shown in Figure 5.1, with the last node in a start tag's list linking to its parent's list. Both the prefix map list as well as the list of attributes are kept in sorted order inside an `ST` item so that canonicalization [W3C, 2001] is efficient, but note that since the list can be sorted only within a single `ST` item, canonicalization of a subtree will require merging the list of the subtree's root with its parent's list.

For the attributes there is only one way to iterate over them, in list order. For the prefixes, there are three different ways. *Local* iteration goes over only the prefix mappings explicitly present in the `ST` item and is used, e.g., when serializing the whole document. *Global* iteration, on the other hand, goes over all in-scope mappings in list order, and is used, e.g., to locate the namespace URI mapping to a given prefix. Finally, *detached* iteration goes over all in-scope prefix mappings dropping duplicate prefixes. This is needed when serializing only a subtree, since any prefix mapping

in scope may need to be included in the root element, but including a mapping for the same prefix twice is an error.

The XAS model deviates from XmlPull by including slightly more structure into the `DTD` and `PI` items, and from both XmlPull and SAX by including both namespace prefix mappings and attributes into the `ST` item. Furthermore, the `ST` item also contains a reference to its parent `ST` item in the XML document, but it is possible to *detach* such an item as well as *attach* a detached `ST` item to another `ST` item. Our original XAS model was closer to the XmlPull model, but actual use revealed that the current model is easier for applications to handle.

The item concept is implemented as a Java class `Item`, an abstract superclass for all item types, which are implemented as subclasses of `Item`. The `Item` class itself has no functionality, only a `type` field to make it possible to test the type of an item without needing potentially expensive `instanceof` operations (or the even more expensive Visitor pattern [Gamma et al., 1995], which would be the "correct" object-oriented solution).

5.1.2 XAS Fragments

A sequence of XAS items is a *fragment* if it is one of

1. A `C` item or a `PI` item,
2. A sequence of `T` and `ER` items,
3. A sequence of fragments surrounded by an `ST` item and an `ET` item with the same name, where there are no two consecutive fragments of type 2 or any fragments of type 4, or
4. A sequence of fragments surrounded by an `SD` item and an `ED` item, optionally starting with a `DTD` item and having exactly one fragment of type 3 and no fragments of types 2 or 4.

These rules are a direct translation of the XML grammar. A complete XML document is a fragment of type 4. The intent behind these definitions is that if a sequence of items is a fragment, its subfragments are uniquely determined by the rules 1–4.

At the lowest layer, processing of item sequences with XAS uses a streaming model. There are *item sources* and *item targets*, the

```
interface ItemSource {
    Item next ();
}
```

```
interface ItemTarget {
    void append (Item item);
}
```

Figure 5.2: XAS item source and target interfaces

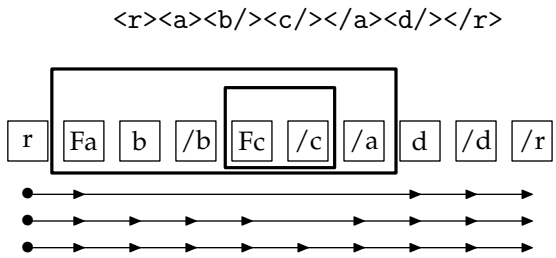


Figure 5.3: A XAS fragment with different iterations

interfaces of which are shown in Figure 5.2. The full processing model follows the Pipes and Filters design pattern [Buschmann et al., 1996], permitting applications to insert their own *item transformers* on top of supplied sources or targets.

While for messaging applications it is usually sufficient to have streaming parsers and serializers, other applications may need an in-memory representation. In our work we have observed that for many such applications there is in the end no need to process XML as a tree, so the basic in-memory model of XAS is an array of items. We chose this form because it will save space compared to the additional pointers that building a tree structure would require.

However, it is still possible to use XAS through a tree view. This is enabled by the use of *fragment items*. Such an item is a core part of XAS and represents a fragment as defined above. A fragment item replaces an `ST` or a `T` item and essentially contains a pointer to the item after the fragment, which is a standard method for representing a tree in a list or array [Knuth, 1997a]. The ability to selectively convert fragments into tree form and iterate or skip them is illustrated in Figure 5.3 where the `a` and `c` elements have been converted into fragment items but other elements have not.

The three possible iteration sequences in Figure 5.3 show varying degrees of skipping. In each case the arrowheads indicate the items that are processed in the iteration. At the fragment items the

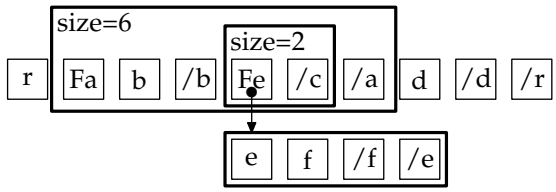


Figure 5.4: A modified XAS fragment

iteration can either skip over the whole fragment or proceed inside it. Note that since there are only two fragment items in the example, the three iteration sequences shown are the only ones possible.

Efficient modification of XML documents is made possible by separating two fragment-related concepts, *length* and *size*. The length of a fragment is the number of items that it contains, and is generally useful information. When a fragment item is contained in a larger fragment, its size tells how many items it takes up space in its parent fragment. For example, the size of the fragment item representing element `c` in Figure 5.3 is 2 and that of element `a` is 6. Size is used by iterators to know the item logically following a fragment item, since the physically following item can be a part of the fragment.

An example of this is shown in Figure 5.4 where the fragment of Figure 5.3 has been modified by replacing the `c` element with an `e` element containing an `f` element. Here the fragment of the new `e` element still takes up only two items in its parent, so its size remains 2. However, now its list of items points to an external list, and its length is therefore the length of that list, or 4. The `ET` item for the `c` element remains in the main fragment, but any processing of the fragment will skip over it by using the size of the fragment now containing the `e` element.

5.2 XAS Extensibility and Advanced Features

The design of XAS included an explicit requirement for extensibility. Our earlier work [Kangasharju and Lindholm, 2005] had integrated typed content into the low-level model, but even then,

explicit extensibility through the ability to define application-specific items was quickly added. For the redesign of the API we decided to keep the basic API as simple as possible and only model XML as such, with extensibility considered from the start and as much functionality as possible implemented as extensions.

5.2.1 Extensibility API

Extending XAS is done simply by defining a new subclass for the `Item` class. Such a class will need to define its own item type that is unique across all types used in the application. While ensuring this uniqueness is not really feasible when assuming a number of separate extensions, we structured the item type field by giving meaning to its individual bytes to make clashes less likely.

Any special processing that an application wishes to perform for its own item types can be inserted into the Pipes and Filters pattern by defining an application-specific transform that recognizes the item type. Here, a target may transform such special-purpose items into some XML content whereas a source may introduce such items when it finds suitable content in the XML document. Furthermore, when processing an XML document in memory, the application can recognize its own item types.

As additional support for extension items, XAS includes two concepts called *appendable item* and *serializable item*, represented by the Java interfaces shown in Figure 5.5. Both of these are supposed to be recognized by any serializer that actually outputs bytes, but the former might be recognized by other transformers as well. The semantics of them are that an appendable item has a serialized form consisting of a sequence of (simpler) items whereas a serializable item has a serialized form consisting of bytes, so it cannot be transformed unless there is byte output stream underneath. An example of these item types is provided by the typed data handling described in section 5.3.

The `type` parameter of the `serialize` method denotes the Internet media type [Freed and Borenstein, 1996b] of the underlying stream, and exists to support alternate serialization formats. For instance, a binary format usually permits direct embedding of binary data in a serialized document whereas such content must be represented as base64 in XML. Thus, a serializable item may need

```
interface AppendableItem {
    void appendTo (ItemTarget target);
}
```

```
interface SerializableItem {
    void serialize (String type, SerializerTarget
        target);
}
```

Figure 5.5: Interfaces for appendable and serializable items

```
interface ParserSource extends ItemSource {
    InputStream getInputStream ();
    String getEncoding ();
    StartTag getContext ();
}
```

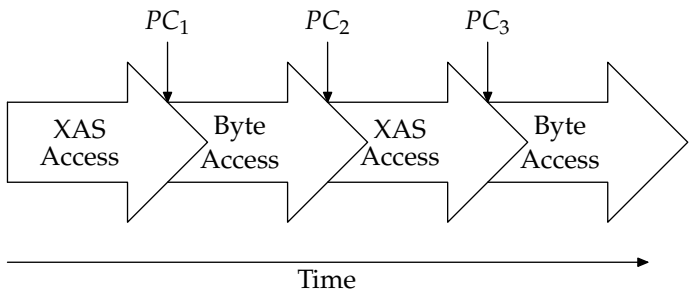
```
interface SerializerTarget extends ItemTarget {
    OutputStream getOutputStream ();
    String getEncoding ();
    StartTag getContext ();
}
```

Figure 5.6: Parser source and serializer target interfaces

to be prepared to serialize itself in a variety of formats, though ultimately it is the application's responsibility to make sure it does not use media types for which it lacks serializers.

5.2.2 Direct Byte Stream Access

The `SerializerTarget` interface referenced in Figure 5.5 is an extension of the `ItemTarget` interface of Figure 5.2. There is also a corresponding `ParserSource` interface as an extension of `ItemSource`. These interfaces, shown in Figure 5.6, provide direct access to the underlying output or input stream, making many applications more efficient. For instance, copying a part of an XML document as such is much more efficient when it can be implemented as a direct byte copy than it would be if parsed and re-serialized.



Constraint: $PC_1 \equiv PC_2$

Figure 5.7: Preservation of processing context (PC) in XAS

In addition to providing access to the byte streams, these extended source and target also provide other information required to fully use the streams. First, they provide the character encoding used in the document so that the application can understand encoded characters correctly. Second, and more importantly, they provide the current *processing context*, which consists of the stack of `ST` items whose corresponding `ET` items have not yet been seen.

The formalization of the processing context is a fundamental one in XAS and what in the end makes its direct stream access so useful. The processing context contains all visible namespace prefix mappings as well as attributes such as `xml:space` that may affect processing. This is required information when, for instance, there is a need to serialize or interpret XML qualified names that are serialized using only the prefix.

The processing context can also be used to formalize what an application is allowed to do with the direct byte stream access. The precise requirement is that the processing context must be the same at the time when the application requests the byte stream and at the time when the application next performs a higher-level XML processing operation. This ensures that any state kept by the low-level XML processor is still correct (with the exception of any position indicators that, e.g., XML parsers commonly maintain to be able to point to erroneous locations). An illustration of such a mixed byte-stream and regular XML processing is shown in Figure 5.7, with the associated constraint on the processing contexts. Here, XAS Access denotes using the item-based parts of XAS.

5.3 Typed Data Extension for XAS

As we noted above, handling typed data is a necessity in an XML messaging application. Typed data usually refers to programming language data types and the encoding and decoding (also called marshaling and unmarshaling) of them to and from bytes. As we also noted, we decided to implement typed data handling as a pure extension for XAS, in part to ensure that XAS extensibility is sufficiently usable for real applications.

Our previous experiences in typed data handling in XML led us to divide typed data into two classes. *Primitive* typed data consists of types whose encoded form in XML is formed of pure text, i.e., not containing any markup. On the other hand, *complex* typed data is structured at the programming language level, and its structure is encoded as XML structure. The intent is that the data contained in complex typed data is also typed data in its own right and encoded as such.

The reason for this separation is our use of alternate XML serialization formats. Such formats usually provide a more efficient representation of primitive typed data, e.g., integers, dates, and floating point numbers. However, it is also often desirable to represent the structure of application data as structure in the produced XML. Thus, an encoder for complex typed data will usually be independent of the underlying serialization format whereas an encoder for primitive typed data is closely coupled with it.

The typed data extension consists of two extension items, the *typed item* and the *parsed primitive*, the former of which represents complex typed data and the latter primitive. The typed item is an appendable item and the parsed primitive is serializable, and both contain an `Object` as a generic value and a `Qname` to denote the type of the value. The type system in XAS is based on XML names as in XML Schema [W3C, 2004g] instead of Java `Class` objects to make it better suited for XML usage.

The encoding and decoding processes for typed data are fully generic. There are *codec* classes (encoder–decoder pair) for both complex (`ValueCodec`) and primitive (`PrimitiveCodec`) data, with interfaces shown in Figure 5.8. An application may register its own codecs with a singleton `Codec` class, and the encoding in the typed item and parsed primitive look for such registered codecs for their

```
interface PrimitiveCodec {
    String getType ();
    boolean isKnown (Qname type);
    void encode (Qname type, Object value,
       SerializerTarget target);
    Object decode (Qname type, byte[] value, int
       offset, int length, String encoding, StartTag
        context);
}
```

```
interface ValueCodec {
    boolean isKnown (Qname type);
    void encode (Qname type, Object value, ItemTarget
        target, StartTag context);
    Object decode (Qname type, ItemSource source);
}
```

Figure 5.8: XAS codec interfaces

type. This class also provides a way to map Java classes to XAS type names for the case when an application only has a generic `Object` and it needs to create a typed data item from it.

For the encoding side things are simple, since the typed data extension items can simply know how to encode themselves. For the decoding side things are not so simple. There exist therefore two source transformers, `PrimitiveConverter` that decodes primitive typed data and `Decoder` that decodes complex typed data. The implementation of `Decoder` is somewhat complex, as it needs to buffer items prior to decoding, provide the items in the buffer as an item source to the codec, potentially recursively as complex typed data may contain other complex typed data, and to determine whether to provide to the application the buffered items or a successfully-decoded typed item.

5.4 XML Security with XAS

One reason for the new design of XAS was to make it easier to implement the XML security specifications. As with the typing extension, our XML security implementation is a pure XAS extension.

The core XAS functionality is fully generic, but its features were selected partially to provide good support for an efficient XML security implementation.

5.4.1 Requirements on the XML API

To ensure both that our implementation technique would not be specific to XAS and that the required functionality in XAS would not be specific to the security implementation, we designed our security implementation in a generic manner. Our starting point was the previous work by Imamura et al. [2002] and Lu et al. [2005]. Based on this, we built a basic design, identifying the generic functionality that an XML API needs to support it.

The intention of the basic design was to promote efficiency by processing security features in a streaming manner whenever possible, and by avoiding re-serialization of data. Our previous proof-of-concept implementation on top of the old XAS system [Kangasharju et al., 2006] had to make several passes through the document, sometimes serializing the same piece several times. While performance was still dominated by the asymmetric cryptography operations, we knew that it was possible to improve the performance of the XML part quite a bit.

We decided to build our implementation on top of an in-memory model of XML, since it does not seem feasible to have a purely streaming implementation for XML Signatures. The reason is that the signature element will include references to other content, and its serialized form depends on that other content. So if the signature element comes first in the document, as is commonly the case, it will need to be buffered in any case, so streaming provides no benefits. Furthermore, it is not clear what is the best way to address pieces of an XML document when the document itself is ephemeral, and this would need to be solved for both signatures and encryption.

We assume that the in-memory model is based on *nodes* of some form. These nodes could be, e.g., the information items of XML Infoset or the nodes of DOM; in our implementation they are XAS fragments defined in subsection 5.1.2. We expect the node division to have sufficient granularity so that individual XML elements and content are represented as nodes. This requirement is usually sat-

isfied by any node-based API. We also require the ability to define new application-specific nodes and a way to add recognition of them into the standard processing.

There needs to be a way to refer to individual nodes using regular variables of the programming language. We shall use the term *pointer* to denote such a reference. The API needs to provide a way to acquire these pointers, usually through a query interface of some sort. Furthermore, it must be possible to modify the XML document through these pointers by replacing nodes with others.

Our last requirement is not usually supported by XML processing APIs. During serialization, our implementation technique requires the ability to access the serialization parameters, such as character encoding, as well as the underlying output stream for direct writing of bytes. This is the main reason that this stream access exists in XAS, though we specified it in a fully general manner and have already used it in other cases. However, we do not believe that the addition of this feature is an onerous requirement, so it might be possible to extend some existing API with it as well.

While these are the only actual requirements as such, XAS also has some features that help in further improving the efficiency of the security processing. One is the above-mentioned maintaining of attribute and prefix lists in sorted order, which makes canonicalization very efficient. As Shirasuna et al. [2004] had noted before, canonicalization can be a bottleneck in XML Signature processing.

XAS also includes the direct byte stream access for the parser side. This could be used to implement streaming parsing of encrypted data by inserting a streaming decrypter on top of the byte stream and then an XML parser on top of that. However, in XML it is not possible to determine the end of element content except by encountering the end tag, and our byte stream access does not currently support the lookahead that this would require. Despite this lack, we see no fundamental obstacle to implementing this feature at some later time.

5.4.2 Implementation Technique

We use the assumed node-based XML API that permits extension nodes and their special-purpose processing for our efficient implementation. The smallest building block of our XML Signature

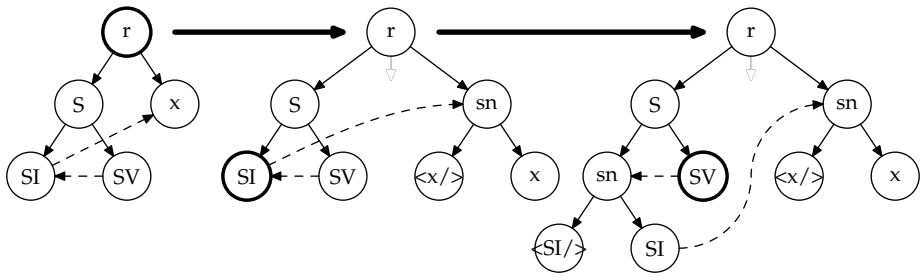
implementation is the *digest node*. Such a node contains a pointer to content that is to be digested. When a digest node is being serialized, it will access the pointed-to content, serialize it into canonical form, and compute a value for itself. It will also replace the pointed-to content with a *serialized node* that contains these serialized bytes as well as a pointer to the original content.

After this replacement, when the serialization process reaches the serialized node, it will compare the encoding used for serialization with UTF-8 that is used for canonicalization. If these are the same, it will simply write the bytes into the output stream, which is accessible per our requirements. This avoids the need to serialize the data twice, with some potential overhead in document size, since, e.g., empty elements are not represented in the compact form in canonicalized documents. Note that this process will serialize content twice if it is the target of two overlapping signatures, but this is required, since the canonical form of a subtree is not the same as it is when serialized as a part of a larger tree.

The `Signature` element is represented by a *signature node*. A signature node contains a digest node for each piece of content to be signed. These nodes are inside a `SignedInfo` element, which is represented using regular nodes. The signature node also contains a *signed digest node*, which is an extension of the digest node that computes a signature as its value from the computed digest.

An example of how the serialization process for an XML document with signatures works is shown in Figure 5.9. The progress of the serialization process is shown by circling the current node in bold. When the process comes to the `SignedInfo` element, it will convert the pointed-to node `x` into a serialized node, with both the bytes and the original node. Processing of the `SignedInfo` node is actually special-cased, since we know it will always be pointed to by the signed digest node of `SignatureValue`, so it will always be replaced with a serialized node. Hence, when the process comes to the `SignatureValue` node, it will simply compute the signed digest over the serialized bytes.

A notable point is that the serialized nodes replace the original nodes so that the pointers are also converted to point to the serialized nodes. This is needed, since the direct writing of the serialized bytes works only if the serialization is using UTF-8. If UTF-8 is not used, the serialization needs to happen using the correct encoding.



Legend	
r, x	element nodes
S	`Signature`
SI	`SignedInfo`
SV	`SignatureValue`
sn	serialized node
<. . . />	element as serialized bytes
⟶	child relation
- - -▸	pointer to node

Figure 5.9: XML signature processing example

Furthermore, our implementation supports the re-serialization of the same document, so that we cannot know whether retaining the serialized bytes is useful or whether retaining the original node is necessary at the first serialization.

For encryption, our implementation contains an *encrypted data node* that replaces the node to be encrypted. When serialized, this node will first serialize the original content and then serialize its own value as an `EncryptedData` element containing the encrypted version of the node's content. Our encrypted data node implementation supports different Internet media types for serialization as well as allows specifying gzip [Deutsch, 1996b] compression prior to encryption, both of which are indicated in the attributes of the `EncryptedData` element.

Our implementation also includes an *encrypted key node*, which corresponds to an `EncryptedKey` element. Such a node contains a number of pointers to content that is to be encrypted as well as a key to use in encryption. When being serialized, an encrypted key node will replace all pointed-to nodes with encrypted data nodes,

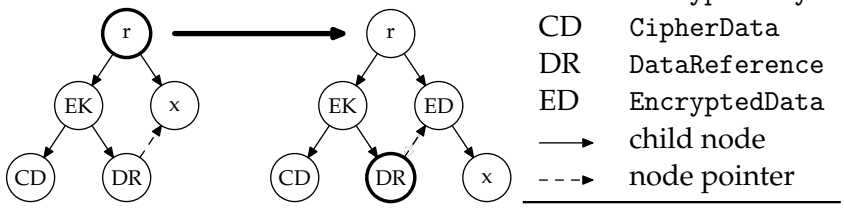


Figure 5.10: `EncryptedKey` processing example

and serialize itself as an `EncryptedKey` element, containing the encrypted version of the key used for encrypting the actual content. An example of this process is shown in Figure 5.10, similarly to Figure 5.9. Note that serialization of the encrypted data happens only when the encrypted data node itself is actually processed.

5.4.3 Extensions to XML Encryption

As we have noted, the main problem of XML on mobile devices is its verbosity. In normal communication this is usually solved by compression, but when encrypting data, compression needs to be applied prior to encryption, as encrypted data is not supposed to be compressible [Schneier, 1990]. However, the XML Encryption specification provides no standardized way to implement this sensible policy.

Because of this, our implementation provides two extensions to regular XML Encryption. First, we make use of the `MimeType` attribute of the `EncryptedData` element to denote the type of the serialized form, namely whether it is XML or a binary format. This saves the regular `Type` attribute to denote whether the encrypted content is an element or the content of an element.

The other extension is a completely new attribute for the `EncryptedData` element that we call `ContentEncoding`. The semantics of this is intended to be the same as those of the `Content-Encoding` HTTP header [Fielding et al., 1999], namely to indicate any encoding that has been applied to the content after it was serialized but prior to encryption. Our system currently recognizes only one

Tree	DOM, JDOM, XOM	
Event list		
Event stream	SAX, StAX	XAS
Byte list	File	
Byte stream	Stream	

Figure 5.11: API support for different views of XML

value for this attribute, `gzip`, to indicate that the content has been compressed with gzip.

There is an existing attribute called `Encoding` in the `EncryptedData` element, but, like with the `Type` attribute, we could not use it. The reason is that the semantics of the `Encoding` attribute appear to be the same as that of the `Content-Transfer-Encoding` MIME header so it denotes only the manner in which 8-bit bytes have been encoded for transmission and does not allow more elaborate encoding methods.

5.5 Summary

There exists very little research into XML processing APIs; usually the goal of XML API design is to just enable generic XML processing without considering the more elaborate needs of some applications. In contrast, the design of XAS began by explicitly determining what kinds of special features would enable efficient XML processing applications and then developing generic functionality to support such features.

One of the distinguishing features of XAS is its consideration of XML at various levels of abstraction. As shown in Figure 5.11, the usual pattern is to have a separate API for each type of access. XAS, on the other hand, is built on the idea that all these different kinds of access are worthwhile and it should be possible to seamlessly combine them in the same application.

Of the features described here, the most prominent one is the XML security implementation. While special-purpose streaming implementations had been written before, XAS differs from these

in that its implementation works also in cases where streaming processing is not possible. Furthermore, the idea of using the bytes acquired during canonicalization as the serialized form as well does not seem to have appeared before.

We have also implemented an XML indexing feature by recording the byte offset and processing context at the indexed points in a document. We also implemented a repositionable parser that can take an offset and its corresponding processing context and continue parsing correctly at the indicated offset. These features were useful in implementing an on-demand construction of an XML document in memory, which forms the basis of our XML editor that can process gigabyte-size XML files on a mobile phone [Lindholm and Kangasharju, 2008]. This is also an improvement over an existing special-purpose implementation of the concept [Fernandes and Raghavachari, 2005]. Indexing, however, is not so useful in messaging, as it requires the whole XML document to be available, so we did not cover it in detail above.

CHAPTER 6

The Xebu Serialization Format

> The evil is young, barely three days old.

The idea of an alternate serialization format for XML is not a new one. As one design principle of XML, listed in section 2.1, was "Terseness [...] is of minimal importance", there have been several attempts to reduce the amount of space that an XML document takes. We will below cover the most important XML compression ideas, and then move on to binary serialization formats and the work done at the W3C in that area. The rest of this Chapter then presents our own serialization format.

6.1 XML Compression

XML documents have much textual redundancy, so they compress very well with generic text compressors. However, XML has structure beyond the linear one exploited by a typical compressor. For instance, it could be expected that elements with the same name (e.g., two `occupation` elements) would have more similar content than just consecutive elements (such as `occupation` and `born`).

In the early days of XML there was much interest in XML-specific compression. The main interest was in getting better results

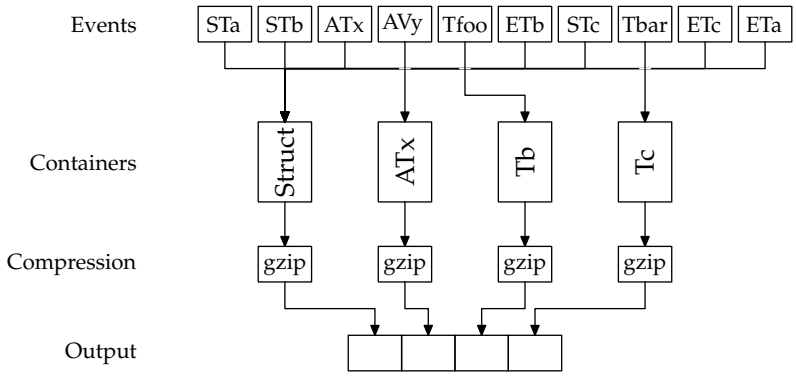


Figure 6.1: The XMill transform

than the popular general-purpose compressor gzip[1], which implements the well-known Lempel–Ziv compression algorithm [Ziv and Lempel, 1977].

One of the early XML-specific compressors was XMill [Liefke and Suciu, 2000]. The basic principles of how XMill works are separation of structure (tags) and data (text content), grouping related data items (e.g., elements with the same name), and using different compressors for different groups. XMill is a very flexible system, allowing these principles to be used to different extents.

The *XMill transform*, shown in Figure 6.1, reads an XML document using SAX and splits the generated events into different *containers*. There is one container for the structure (tags), and a number of data containers. A user can specify the names of elements that are included in each data container. Default containers are then constructed individually for each element name that was not included in the user's definitions. The structure container also contains pointers to the data containers so that the XML document can be reconstructed after the transform.

XMill allows the user to specify *semantic compressors* for data containers. For example, a user could specify that the content of some specified element was always a date value, so the semantic compressor could represent these in an efficient binary format. Semantic compressors can also match regular expression templates

[1] http://www.gzip.org/

against the data value to eliminate common parts directly.

In the final phase, gzip is applied individually to each container (to the data containers after semantic compression), and the containers concatenated to form the final document. Measurements on XMill performed by Liefke and Suciu [2000] indicate that XMill performs better than gzip on many kinds of XML data, and that an original text document converted to XML and compressed with XMill can be smaller than the original document compressed with gzip. Timing measurements indicate that XMill is approximately as fast as gzip, both compressing and decompressing.

While XMill performs well when combined with gzip, there exist better algorithms for textual data compression. A currently-popular one is bzip2[2], which applies the Burrows–Wheeler transform [Burrows and Wheeler, 1994] to preprocess the data into a more compressible form. Bzip2 achieves a compression ratio close to that of state-of-the-art compressors while being much faster.

Since the XMill algorithm only transforms the data to be compressed, it can be used with any compression algorithm. It would therefore be conceivable that using, e.g., bzip2 as XMill's compression algorithm would yield even better compression. However, this has been observed not to be the case; in fact, applying the XMill transform to an XML document can worsen the performance of state-of-the-art compressors [Cheney, 2001].

After noticing that XMill's modeling of XML data is not sufficient, Cheney [2001] proposed a technique called *Multiplexed Hierarchical Modeling (MHM)*, based on the well-known PPM technique [Cleary and Witten, 1984]. The idea behind MHM is roughly similar to XMill: split the XML into different streams based on the item type, and model each of these streams independently.

The MHM algorithm is performed on an *Encoded SAX* stream, which is essentially the sequence of events produced by a SAX parser from an XML document. It builds different models for document structure and various kinds of names and text content. An additional improvement to *inject* element start symbols at various places inside the element improves the models even further. This process has been implemented in the XMLPPM[3] tool.

[2] http://www.bzip.org/
[3] http://xmlppm.sourceforge.net/

Based on investigating the activity (development, mailing list discussion, etc.), both XMill and XMLPPM seem to be very unused. In particular, XMill appears to have been abandoned after publication, and its authors have moved on. The situation is even worse for the many commercial tools that existed five to six years ago, as they have completely disappeared.

Our main concern, though, is with XML messaging. Here typical XML documents are small and contain much structural information instead of text. The methods described above are all generic XML compressors, so it seems believable that there could exist messaging-specific ways to compress XML better.

Considering messages to a single destination, there will very probably be a large amount of similarity among them. Especially in the case of SOAP there will always be the SOAP framing, and possibly some common extension headers. If we can assume a session between two messaging applications, we could use differential encoding techniques that have proved useful for Internet protocols [Casner and Jacobson, 1999; Jacobson, 1990].

Even if we do not assume a session, there may still be a WSDL description of a service endpoint, possibly including a message schema. This description can be used to create a template message, to which differential encoding is applied [Werner et al., 2004]. However, it appears that this technique does not yet provide substantial benefits, nor is the XML differencing and patching technology used sufficiently robust to run automatically.

6.2 XML Binary Serialization

For use in the resource-constrained environment of the wireless world, XML compression methods are of little benefit. The goal there is not merely to reduce the size of the documents but also to reduce processing time and memory consumption in serialization and parsing. An additional compression step, while beneficial for bandwidth usage, only exacerbates these other concerns.

What is needed is an XML representation format that can be directly read and written in a streaming manner. This need is the origin for many *binary XML* formats, as first exemplified by WAP Binary XML (WBXML) [W3C, 1999]. *Binary XML* as a term usually

refers to a binary serialization format that is designed to be compatible with XML and can be written and read directly without going through XML in between[4].

Binary XML techniques can roughly be divided into Infoset-based and schema-based [Pericas-Geertsen, 2003]. Of these, the former is suitable for any XML data while the latter may require information on a schema that documents conform to. We must handle general XML in our messaging system, so the basic format needs to be Infoset-based. However, often a complete or partial schema for the messages is available, so schema-based optimizations should be included if possible.

6.2.1 Tokenization Techniques

One basic concept of binary XML formats that has been used by many existing formats is called *tokenization*. This is similar to what generic compressors like gzip do in that a recurring string in the data is replaced by a short integer token. This provides both increased compactness, as the string is shortened to often only one or two bytes, and improved processing speed, as there is no need to perform as much string processing on the parser side.

While generic compression requires much processing power, the tokenization performed by binary XML formats is much more efficient. This is because tokenization does not consider every substring of the serialized form to be tokenizable, only the names in XML items. For instance, of an element name, a binary XML tokenizer tokenizes only the namespace and local name instead of considering all possible substrings of the full qualified name.

The oldest format, WBXML [W3C, 1999], is a simple tokenizer. Its tokens come from a space of 65536 (2^{16}) available values, and at each point of a WBXML document there is a *current code page*, which gives 8 bits of this value, allowing a token to be represented in a single byte, yet enabling a large space of possible tokens. Code pages are switched with special tokens; obviously the placement

[4]Strictly speaking, the term *binary XML* is an oxymoron, since XML is always text. However, it is a common term, and there is no widely-used short alternative, so we shall continue using it.

of tokens into code pages needs to be done with care to avoid too many code page switches.

While WBXML is an old and established format, it is poorly suited to the XML messaging world. Its largest deficit is that it only works for the specific format used with WAP, and any modification to this would require a round of standardization. However, even if this would be remedied, it would still leave the problem of namespaces, which are not at all supported by WBXML.

Millau [Girardot and Sundaresan, 2000] extends the WBXML format by splitting the document into a *structure stream* and a *content stream*. This allows separation of structure from content as well as separate compression of content. Millau also extends WBXML to permit binary encoding of common data types such as bytes, integers, or floating point values. Finally, the Millau implementation provides binary versions of the SAX and DOM APIs, which were measured to have a positive effect on application performance. However, like WBXML, Millau does not support namespaces, so it cannot be considered a modern format suitable for our purposes.

The best-known modern general-purpose format is indubitably Fast Infoset [Sandoz et al., 2004]. This format represents the information items of XML Infoset in an Abstract Syntax Notation One (ASN.1) schema [ITU, 2002b]. Then, it is possible to use the well-established *encoding rules* of ASN.1 [ITU, 2002a,c] to serialize a document represented as an Infoset into a more compact form.

The main benefit of Fast Infoset comes from the indexing of strings and qualified names, i.e., tokenization. Another benefit, which is also common to most binary formats, is the ability to embed binary content directly into an XML document without encoding it in base64. It is also possible to preserve the state of the indexing from one document to another, which is very useful for message streams containing similar messages.

A somewhat similar general-purpose format is XBIS [Sosnoski, 2003a]. XBIS is designed to be one-to-one compatible with Canonical XML, which is a deviation from most other binary formats that consider some more abstract data model. This makes XBIS a very stream-oriented format.

The basic concept in XBIS, as in Fast Infoset, is tokenization. Names of elements and attributes are always tokenized, while tokenizing text and attribute values is optional. A document is seri-

alized as a sequence of *nodes*, each of which represents some piece of XML data. The serialization format of nodes has been chosen so that more commonly used types of nodes, e.g., element start nodes, are serialized in a smaller number of bytes than, e.g., processing instructions.

In contrast to the use of qualified names in Fast Infoset, XBIS tokens always reference the actual namespace URIs. As all element and attribute names of a single namespace will simply reference the first instance of that namespace (which should be a namespace declaration in namespace-well-formed XML), this does not consume additional space. It also makes the XBIS format somewhat more independent of the actual namespace prefix mappings.

In contrast to WBXML and Millau, Fast Infoset and XBIS do not limit the space of available tokens in any manner. Instead, they define ways to encode arbitrary integers, and this encoding is also used for the tokens. This makes these formats more widely applicable, as the tokenization does not degrade for any documents, but it can cause an increase in the sizes of documents, since larger token values will take more space in serialized form. The memory consumption of the implementation will also grow without bounds, as every token ever assigned needs to be remembered indefinitely. This limits the use of these formats for streams of messages while retaining state between messages.

Chiu et al. [2005] have designed the Binary XML for Scientific Applications (BXSA) format explicitly based on the requirements of scientific computing. BXSA extends the XML Infoset model with typed data, especially arrays, but is capable of representing arbitrary XML. In contrast to most other general-purpose formats, BXSA uses a type-length-value encoding for elements, similarly to ASN.1 Basic Encoding Rules (BER) [ITU, 2002a], where the value of an element can contain similar type-length-value frames for contained elements. In comparison with established scientific data formats, Chiu et al. measure BXSA to perform well, but there is less consideration of the general XML world.

6.2.2 Using Schemas to Improve Compactness

In SOAP messaging we can say that there is always partial schema information available, namely the high-level SOAP message struc-

ture presented in section 2.2. Furthermore, in many cases there will be schema information on some header blocks and the message body. It can therefore be useful to allow the binary format to take advantage of available schemas. However, since a schema for messages can be a composite of several independent schemas, the format needs to be flexible enough to allow partial schema information to also have benefits.

Existing formats that can take advantage of schema information include BiM of MPEG-7 [Niedermeier et al., 2002], Fast Web Services [Sandoz et al., 2003], Xenia [Werner et al., 2006], and XML Schema-based Binary Compression (XSBC) [Serin, 2003]. There is also a schema extension for Millau [Sundaresan and Moussa, 2001]. Unlike with general-purpose techniques, there is more diversity in schema-based techniques.

XSBC [Serin, 2003] is a very simple format. Its approach is basically tokenizing the names in a schema beforehand, and encoding typed data specially, determining the correct encoding from schema information. Each element gets a unique token based on the XPath expression that points to it. This is necessary so that elements with the same name but differently-typed content can be distinguished from each other.

Performance measurements on XSBC [Bayer, 2005] indicate that XSBC achieves approximately the same serialized form size as Fast Infoset. This is expected, since the tokenization technique used is principally the same, and binary encoding of primitive typed data often does not reduce the size. Furthermore, parse times for XSBC are clearly worse than for Fast Infoset.

The Millau extension of Sundaresan and Moussa [2001] is based on DTDs. The mechanism of the schema optimization is to perform as a validator against a DTD by traversing both the XML document and the DTD simultaneously. There is only a need to produce some structure information when the DTD allows several choices as to the next item.

The measurements of Sundaresan and Moussa [2001] are performed only for content-heavy XML. This is puzzling, since this schema optimization is very slow and does not perform any content compression, so the measurements indicate it being a very poor choice. Furthermore, the presence of DTD operators deep in the tree is a significant cause of poor performance for this opti-

mization, requiring that the DTDs used with this technique do not have too many choices available.

Fast Web Services [Sandoz et al., 2003], like its sister technology Fast Infoset, is based on ASN.1. Here, however, instead of defining an ASN.1 schema for the XML data model, a mapping from XML Schema to ASN.1 schema [ITU, 2004] is specified. Then, XML instances conforming to a given schema can be transformed into ASN.1 instances of the mapped schema. A standard ASN.1 encoding, such as Packed Encoding Rules (PER) [ITU, 2002c], is then used to produce the serialized form.

The performance of Fast Web Services appears to be better than that of the Millau extension. The measurements of Sandoz et al. [2003] indicate that over 60 % of total time in a Web service invocation is spent on processing the SOAP body, and that Fast Web Services can cut this time down to one tenth. This factor is measured to increase with document size; the reported result is for a 50-kilobyte XML message, which is 10 kilobytes encoded in the Fast Web Services format.

However, if the complete schema for a message is available, the ASN.1-based technique of Fast Web Services can perform significantly better. Measurements on a large corpus of XML messages [Cokus and Winkowski, 2002] indicate that ASN.1 PER can achieve up to 50-fold improvement in document size compared to XML. However, Cokus and Winkowski do not present timing measurements.

The BiM format [Niedermeier et al., 2002] was designed for use with the MPEG-7 metadata format [Avaro and Salembier, 2001] of Moving Picture Experts Group (MPEG)[5] used to represent audiovisual content. The basis of BiM is generation of automata from either a DTD or an XML Schema. The serialization automaton is driven by the items of the XML document and produces the serialized form directly. The parsing automaton performs the reverse transformation.

The automata of BiM allow a very compact serialized form to be generated. With the kind of constrained schema that BiM is designed for, there are typically only a very small number of possible following items at each point, and the BiM automaton transi-

[5] http://www.chiariglione.org/mpeg/

tions can then output the minimal number of bits required to distinguish between these alternatives. Measurements [Cokus and Winkowski, 2002] indicate that BiM is capable of achieving over 10-fold reduction in document size.

A technique similar to BiM was developed by Werner et al. [2006] for a format called Xenia. This work is an extension of BiM in that it makes explicit how recursive element definitions in a schema are handled. Namely, Xenia uses pushdown automata, giving it the ability to recurse during processing while maintaining sufficient state to resume processing correctly after the recursive element has been processed.

None of these formats permit deviations from the schema, but XSBC could easily be extended to support them. We see this rigidity as a liability, since in real use it is not uncommon that documents are produced without ensuring their validity, or that schemas evolve and are no longer the same everywhere. Furthermore, different use cases may require different schemas to be applied to the same XML document at various times.

6.2.3 Binary XML Standardization

The W3C, as the keepers of the XML specification, has also followed the binary XML developments, and in September 2003 organized a workshop on Efficient Interchange of XML Information Item Sets [W3C, 2003b]. Several participants in this workshop presented their own binary formats, and as a result, the W3C chartered the XML Binary Characterization (XBC) WG[6] (the author of this dissertation participated in this WG representing the University of Helsinki). The WG's purpose was to determine use cases for an alternate serialization format, to find out why XML is not suitable for these use cases, and to provide a recommendation on whether the W3C should continue work in this area.

The XBC WG concluded its work at the end of March 2005 with the publication of its findings [W3C, 2005c], supported by use cases [W3C, 2005f], required format properties derived from the use cases [W3C, 2005e], and ways to measure the properties [W3C, 2005d]. The findings were that a binary format that supports the

[6] http://www.w3.org/XML/Binary/

use cases is feasible to build and that the W3C should standardize such a format. Based on this recommendation, the W3C chartered the Efficient XML Interchange (EXI) WG[7] to

> define an alternative encoding of the XML Information Set that **addresses at least the minimum requirements identified by the XML Binary Characterization Working Group**.

The EXI WG charter[8] defines the design goals of XML (see section 2.1) to also be design goals of EXI, but lists the following exceptions:

- The interchange format must be compatible with the XML Information Set instead of being "compatible with SGML" (XML goal 3),
- For performance reasons, the format is not required to be "human-legible and reasonably clear" (XML goal 6),
- Terseness in efficient interchange is important (XML goal 10).

The EXI WG began its work at the beginning of 2006. It initially issued a call for contributions to the industry and XML community, asking for existing binary format implementations. The intent was to evaluate each of these against the requirements set to the group, in part to determine whether it is even feasible to satisfy all the requirements.

The group received a total of eight submissions that included implementations. Each submission was also represented by a participant in the WG, though this was not a requirement for a submission to be considered. The submissions and their submitters were

- X.694 [ITU, 2004] with PER [ITU, 2002c], submitted by Paul Thorpe from OSS Nokalva,
- X.891 (Fast Infoset) [ITU, 2005], submitted by Paul Thorpe from OSS Nokalva and Paul Sandoz from Sun Microsystems,

[7] http://www.w3.org/XML/EXI/
[8] http://www.w3.org/2005/09/exi-charter-final.html

- Xebu [Kangasharju et al., 2005b], submitted by Jaakko Kangasharju from the University of Helsinki,
- X.694 [ITU, 2004] with BER [ITU, 2002a], submitted by Ed Day from Objective Systems, Inc.,
- Efficient XML [Schneider, 2003], submitted by John Schneider from AgileDelta,
- XSBC [Serin, 2003], submitted by Don Brutzman, Don McGregor, and Alan Hudson from the Web3D Consortium,
- FXDI, submitted by Takuki Kamiya from Fujitsu, and
- esXML [Williams, 2003], submitted by Stephen Williams from High Performance Technologies, Inc.

The EXI WG ran measurements on all of the submitted candidates, mainly on how much compression they achieved and how fast the submitted implementations were. Based on these measurements, the WG selected the Efficient XML submission as the basis for their current development of a binary format. This process and the measurements are covered in more detail in chapter 8.

6.2.4 Efficient XML Interchange

The first draft of the EXI format was published in July 2007 [W3C, 2007a]. At the basic level, EXI uses tokenization like many other formats. However, it also includes learning of content models, called *EXI grammars*, which gives it an advantage over other formats. This learning applies to the content of each element, so that EXI will serialize repeating content even more compactly if it appears in an already-encountered context.

In more detail, EXI is based on the streaming model of XML, and it uses a stack of grammars, with each grammar describing the possible event sequences for a single element. As start tags and end tags are encountered in the stream, grammars will be pushed into and popped out of the stack, respectively. In the EXI specification, all of the grammars are *right-linear LL(1)* grammars (see, e.g., [Lewis and Papadimitriou, 1998]), i.e., all productions are of the form

$$N_1 \rightarrow TN_2$$

where N_1 and N_2 are non-terminals and T is a terminal representing an event, but the general technique is applicable to at least any context-free grammar.

Each grammar production is assigned an *event code*, with more likely productions getting codes representable in smaller space. The serialization process can be seen as serializing the sequence of event codes that encode a leftmost derivation of the document in the EXI grammar. Elements and attributes are treated generically at first, i.e., there is a production that matches any start tag or attribute in appropriate places, and the name of the element or attribute is serialized after the corresponding event code. However, after this first appearance, the grammar is modified to include a production with specifically the encountered start tag or attribute, which gains benefits if the added production repeats later.

The schema optimization of EXI is based on the same principle as its standard technique. Namely, a given schema is compiled into a group of EXI grammars, which provide beforehand a more accurate version than the dynamic learning in the non-schema-using case. In addition, the grammars include events not permitted by the schema, which gives EXI the ability to tolerate arbitrary schema deviations while still achieving much improved compactness for valid documents. The grammar modification is not applied to the non-terminals derived from a schema, but only the ones encountered at deviations.

Finally, EXI includes its own compression technique, which performs better than simply running a generic compressor over the EXI document. This technique is basically the same as the XMill transform shown in Figure 6.1, with the differences that in EXI the structure stream contains less explicit structure, some of the content streams may be combined if they do not contain a sufficient number of material, and the document is compressed in chunks to permit streaming.

6.3 The Basic Xebu Format

Our format, Xebu [Kangasharju et al., 2005b], is based on a minor extension of the XmlPull [Slominski, 2004] API for XML parsing. Each of the serialization methods shown in Figure 6.2 corresponds

```
interface XmlSerializer {
    void startDocument (String encoding, boolean
       standAlone);
    void endDocument ();
    void setPrefix (String prefix, String namespace);
    XmlSerializer startTag (String namespace, String
       name);
    XmlSerializer attribute (String namespace, String
        name, String value);
    XmlSerializer endTag (String namespace, String
       name);
    XmlSerializer text (String text);
    XmlSerializer text (char[] ch, int start, int
       length);
    XmlSerializer typedContent (Object data, String
       namespace, String name);
    void entityRef (String name);
    void processingInstruction (String text);
    void comment (String text);
    void docdecl (String text);
}
```

Figure 6.2: Xebu serializer interface

to a *Xebu event* as shown in Table 6.1[9]. A Xebu event is serialized as a one-byte *type token* that contains the event's type and some flags to indicate how the rest of the event is to be processed, followed by the content of the event.

Each string in an event's content is serialized either as a one-byte token or as a length-prefixed string. If Xebu has been set to *tokenize dynamically*, the latter form also includes a one-byte token for later appearances of the same string. Tokenization can happen either only for namespaces and names or for all strings in an event's content.

Xebu includes four separate *token mappings*, for each of namespaces, names, values, and text. Namespaces are simply the name-

[9]Note that the events and their names differ somewhat from the XAS items. This is intentional in the sense that XAS and Xebu are designed as independent components, so their designs do not need to be aligned.

Table 6.1: Events in Xebu serialization

Event	Abbr.	Method	Data
DOCUMENT START	SD	startDocument	none
DOCUMENT END	ED	endDocument	none
PREFIX MAPPING	PM	setPrefix	namespace, prefix
ELEMENT START	ES	startTag	qname
ELEMENT END	EE	endTag	qname
ATTRIBUTE	AT	attribute	qname, value
CONTENT	CO	text	text
TYPED CONTENT	TC	typedContent	qname, data
COMMENT	CM	comment	text
PROCESSING INSTRUCTION	PI	processing-Instruction	text
ENTITY REFERENCE	ER	entityRef	name
DOCUMENT TYPE	DTD	docdecl	text

space URIs. Names consist of pairs of a namespace and a local name. Values denote attribute values and have a namespace, a local name, and a value. Finally, text is simply text content. By tokenizing complete names instead of each component separately, Xebu achieves additional size reduction.

An approach to improve document size even further is to preload the token mappings, which we call *pre-tokenization* and explicitly support in the Xebu serializer and parser. Our main use of this feature is in a messaging connection where the messaging system preserves the token mappings from one message to the next over the lifetime of the connection. In such a case, only the first message over a connection needs to contain the common names explicitly.

We chose to use only one byte for a token, since we believe that the number of actually-common strings will be quite small for each separate communication channel. Allowing more tokens would have either wasted space (both in the messages to represent the values and in memory to store larger token mappings) or complicated processing. For example, the code pages of WBXML are usable for the very static case that it considers, but would be extremely complex to implement for the more dynamic document sets that Xebu considers.

The second design decision was to include token values explicitly in the serialized form. This does waste space in comparison with the approach of having them be selected implicitly. However, since the token space is limited in size, the implicit approach would require the eviction policy of expired tokens to be specified for interoperability. In our approach the serializer can select its token replacement policy freely, and can even vary it dynamically without synchronization problems.

We have considered a number of different token replacement policies in our work. The current implementation uses the Least Recently Used (LRU) policy to determine which token to evict. However, when considering the names in XML messages, we note that some names are repeated in many messages while others are very rarely present. Therefore, a technique like Adaptive Replacement Cache (ARC) [Megiddo and Mocha, 2003] that provides two classes of tokens, persistent and temporary, could be beneficial.

Another Xebu feature, also common in other binary XML formats, is the binary encoding of known data types. This uses the TYPED CONTENT event of Xebu, which denotes primitive typed data as described in section 5.3. The content of a TYPED CONTENT event consists of a generic `Object` value and a qname identifying the data type in the manner of XML Schema.

The design of Xebu does not include tightly integrated type information in the serialized form, so the parser cannot, upon encountering a TYPED CONTENT event, determine the actual type of the content. This is why the serialized form of a TYPED CONTENT event also includes its length in bytes (the flag bits are used for this, so sufficiently short serialized forms are not penalized). The Xebu parser then provides only an array of bytes to the higher level, which is then responsible for providing the correct type to the decoder. This is left completely to the application, so the determination of type can be hard-coded or it can come from schema knowledge or the presence of an `xsi:type` attribute [W3C, 2004f].

6.4 Schema Optimizations in Xebu

As noted above, a binary format for XML messaging should include optimizations in the case when a complete or partial schema

for the messages is available. Both the pre-tokenization and data type encoding described above can be seen as schema-based techniques too, but they still function at the level of individual events and cannot take advantage of the structure described by a schema. Therefore Xebu also includes one technique that is based on exploiting the structure as well.

6.4.1 Schema Optimization Design

Our approach to schema-based optimization is similar to that of BiM with its automata. We construct a *Codec Omission Automaton (COA)* as a pair of automata, *Encoding Omission Automaton (EOA)* for the serializer side and *Decoding Omission Automaton (DOA)* for the parser side. Their input and output are both sequences of Xebu events, in contrast with BiM where the output of the serializer side and the input of the parser side are bit sequences.

By outputting event sequences instead of the final serialized format we make our schema optimization more independent of the underlying format. We note that it will not be completely independent as the transformed event sequence may not obey the rules that the XML definition places on the form of an event sequence representing an XML document. A sufficient requirement for the format is that the serialization of an event is *contextless*, i.e., a serialized event can be correctly identified and read without knowing any of the preceding or following events. Here reading an event refers to extracting the bytes that comprise the event, not necessarily a full understanding of it, which is not possible in isolation with Xebu's dynamic tokenization.

XML itself is not contextless as we have defined the term. One reason is that recognizing an attribute as an attribute requires first the processing of the attribute's start tag, and therefore an attribute cannot be distinguished from text content if its ELEMENT START event is not present. Of the formats covered above, we believe that at least XSBC satisfies the requirements for contextlessness.

The schema optimization we perform is simply the omission of events from the input sequence of the EOA. Since we perform only a transformation to another event sequence, there is little else to be done. We do not see any other feasible actual improvements that could be made while still producing event sequences.

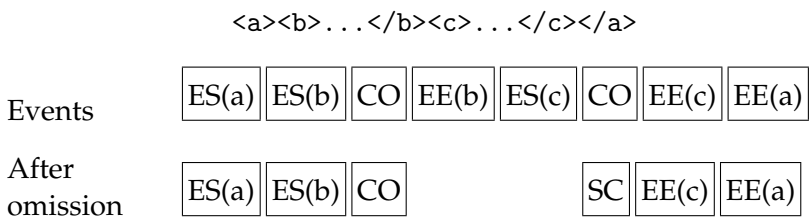


Figure 6.3: The need for the SEPARATE CONTENT event

The omission of events introduces issues that are not covered by the model introduced in section 6.3, but will need to be handled by the serialization. An issue that could break the system is the coalescence of CONTENT events. The model allows an element's text content to be represented as multiple consecutive CONTENT events to support efficient streaming. However, if events are omitted so that two separate text contents become adjacent, as shown in Figure 6.3 where the `ET(b)` and `ST(c)` are omitted, the parser side will need to recognize where the content of `b` ends and the content of `c` begins.

Our solution is to introduce the SEPARATE CONTENT event that functions as a CONTENT event, except that its content explicitly does not belong together with any preceding CONTENT event's. This is serialized with a *separation flag* in a Xebu CONTENT event. In Figure 6.3 this is shown as the second CONTENT event becoming a SEPARATE CONTENT event after the intervening events have been omitted.

The TYPED CONTENT event provides a possibility for some further size improvement. The event omission may in some cases cause two TYPED CONTENT events to become consecutive, and in cases where data structures are serialized as XML, this situation might be quite common. Therefore we ensure that each kind of primitive data is serialized in Xebu so that it is decodable without length information, and serialize the consecutive TYPED CONTENT events as a single TYPED CONTENT event at the Xebu level. This is especially beneficial when serializing a sequence of small integers, since then each integer takes only one byte instead of the two that would be consumed if each integer were serialized as a separate TYPED CONTENT event.

As our schema language we chose RELAX NG, mainly because

it has, in addition to XML syntax, a standardized compact syntax [OASIS, 2002b] more resembling traditional programming languages. This compact syntax is both easier for humans to handle and more amenable to traditional parsing techniques.

Our language of choice for implementing the COA generator was Standard ML [Milner et al., 1997], whose features are a good match for implementing compilers [Appel, 1998]. As we were not sure what subset of RELAX NG would be supported, a flexible parsing system was necessary. The powerful structured data manipulation capabilities of Standard ML made evolution of the generator easy. To build our parser, we used the *combinator* technique [Fokker, 1995], which is well suited for implementing understandable easily extensible parsers for simple languages like the RELAX NG compact syntax.

The parser implementation that we wrote to construct abstract syntax trees for RELAX NG eventually ended up parsing the complete RELAX NG compact syntax, as there were unexpected dependencies and conveniences in parts that we originally thought would be safe to discard. However, for automaton generation we omitted two, perhaps central, features.

RELAX NG supports the *interleave* operator which takes a set of sequences and allows these sequences to be interleaved with each other. Each component sequence, however, must match as some subsequence of the combined sequence. This operator is responsible for much of the power of RELAX NG, but we did not manage to find satisfactory semantics in our event omission model that would allow concrete improvements for interleaved sequences. Our automaton construction therefore does not process the interleave operator in any manner.

The other feature we left out were recursive definitions. Like all schema languages, RELAX NG allows naming of schema rules and referring to these named rules even within the same rule. Our choice to use finite automata as such precluded the use of recursion, though. In our most central use cases the messages are encodings of non-recursive data, so this omission was not as crucial as it could be in a more general context. We have briefly considered adding a stack of states to the COA to allow the possibility of recursing in the automata, but have not yet begun the design.

6.4.2 Codec Omission Automaton

We next give a description of how the COA operates. Both the EOA and the DOA are event-driven automata: their input is an event sequence, and their transitions on these events have specifications on what events to output. In both automata transitions also have, in addition to an event, a *type* that determines (some of) the processing to perform on that transition.

The event of a transition may be either a *wildcard* event or a concrete event. In the case of a concrete event, some of its components may be wildcards. The set of *matching transitions* for an input event is selected by collecting all the transitions whose event matches the input event according to the following rules:

1. A COMMENT or a PROCESSING INSTRUCTION input event does not match any event

2. A wildcard event matches any other input event

3. A non-wildcard event matches the input event if they have equal non-wildcard components

After the set of matching transitions is collected, the *most specific* of these is selected. Basically, this means the transition whose event has the fewest wildcards. If the set of matching transitions is empty, the *default transition* is taken. This default transition does not change the state that the automaton is in; we will cover below what processing happens for each of EOA and DOA.

In the EOA transitions can be of two types, `out` and `del`. Of these, the `out` transition specifies that the transition outputs the event that triggered the transition. The `del` transition specifies that no output is produced. In both cases the input event is consumed from the sequence. The default transition is an `out` transition, i.e., it outputs the input event without changing state.

The DOA has two kinds of transitions: `read` and `peek`. However, these are not the main part of the transitions in the DOA. In addition to the event and type, each transition in the DOA also has two lists, the `push` and `queue` lists. These lists contain events that were omitted by the EOA; the transition semantics provides for their insertion into the DOA's output sequence.

When the DOA makes any transition, it begins by outputting the transition's `push` list. If the transition is a `read` transition, it will then output the event that triggered the transition. And, independently of the type of the transition, it will then output the transition's `queue` list. The default transition is a `read` transition with empty `push` and `queue` lists, i.e., the default transition produces exactly the input event in its output.

The semantics of the `peek` transition is otherwise the same as that of the `read` transition except that the input event is not consumed from the input sequence and the DOA does not output it. This provides a way for the DOA to perform one-event lookahead. The main uses of the `peek` transition in our implementation are for wildcard names: the transition's event will have a type, but no name, so that it matches any event of that type. Our implementation is constructed so that the DOA never contains cycles consisting only of `peek` transitions, which ensures that processing will always terminate.

An example RELAX NG schema and its associated generated COA are given in Figure 6.4. The schema (a) says that a `person` element is a sequence of elements `name`, whose content is a string, and `age`, whose content is an integer. The legend (b) provides some abbreviations for the EOA in (c) and the DOA in (d).

From Figure 6.4 we can clearly see how an element with typed content is converted to a COA. On the EOA side the ELEMENT START and ELEMENT END events are omitted as is the ATTRIBUTE event giving the type of the content. On the DOA side a `peek` transition first inserts the ATTRIBUTE event, and after this, a `read` transition on the TYPED CONTENT event inserts the rest of the element.

In the Figure the `read` transitions for the TYPED CONTENT event have the omitted events in their `queue` list, since they get inserted back *after* the read TYPED CONTENT event. In this case we do not see the possibility of both `push` and `queue` lists being non-empty. Such a situation could happen if the element content was just a CONTENT event. In this case it would be sufficient to have a `read` transition on the CONTENT event that had the ELEMENT START event in its `push` list and the ELEMENT END event in its `queue` list.

An example of how the COA shown in Figure 6.4 works for a document only partially conforming to the schema is given in Figure 6.5. Here the original document consists of an element contain-

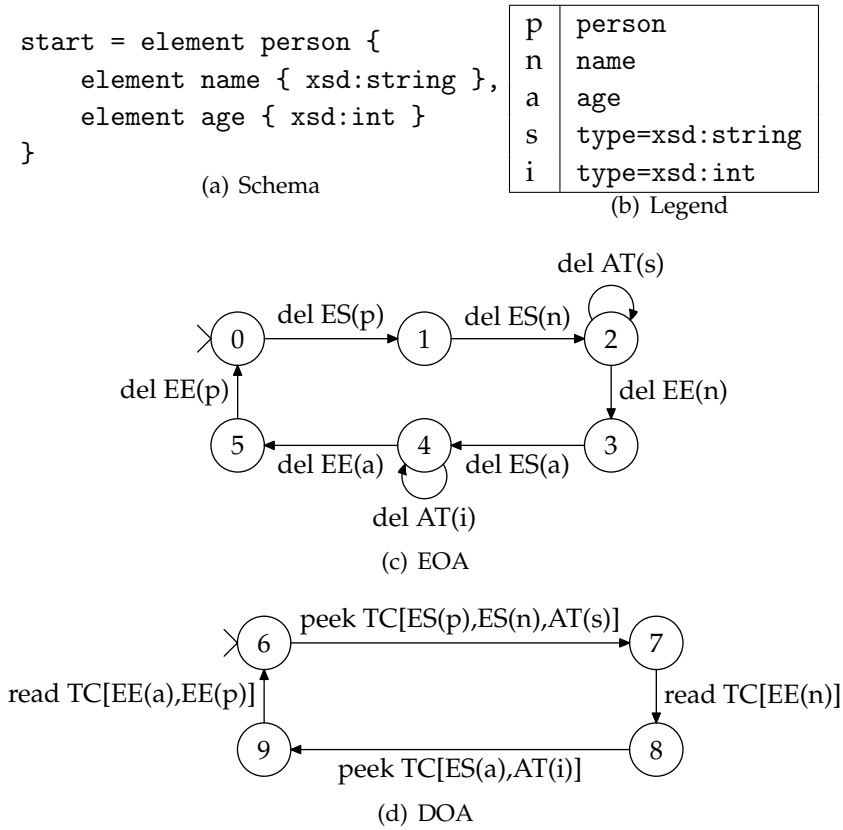


Figure 6.4: An example COA

ing three elements, two of them with typed content. The `b` element is an added element that is not described in the schema, the other names are as in Figure 6.4(b). Since the `b` element is not described by the schema, its content also will not be typed, but just text.

The EOA in Figure 6.4(c) begins its processing in state 0, reading the two `ES` events and the `AT` event, omitting them. In state 2 it will make the default transition and output the `TC` event. After that it transitions to state 3, omitting the `EE(n)` event. Now, since the following events denote the `b` element, the EOA remains in state 3, making default transitions and outputting the full element. After that, it will then proceed as expected through states 4 and 5, omitting the structure and outputting the `TC` event.

The DOA in Figure 6.4(d) begins in state 6. Since the first event

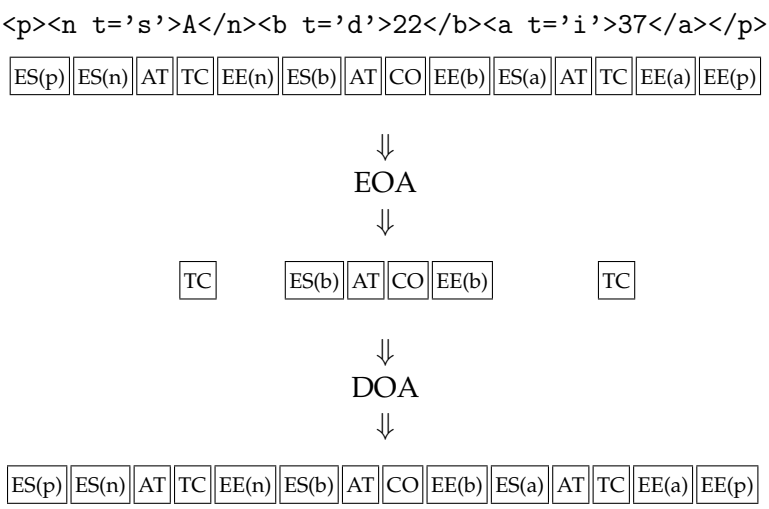


Figure 6.5: Example of Xebu COA usage

it encounters is a `TC` event, it inserts the events that precede it, moving to state 7. From there, actually reading the `TC` event causes it to move to state 8, producing also the `EE(n)` event. Then, since none of the following events match what the transition is expecting, the DOA remains in state 8, making default transitions and producing only the events that are present in its input. Then, after the `b` element has been completed, the DOA will again transition through state 9 to state 6, producing the structure around the `TC` event.

6.4.3 Schema Optimization Implementation

Our RELAX NG parser constructs an abstract syntax tree from its input schema. The implementation then performs some of the simplifications specified by RELAX NG [OASIS, 2001]; as we are not implementing a RELAX NG validator, we only implemented such simplifications that were useful, including some that were our own invention. These simplifications were easily implemented with the *catamorphism* technique [Augusteijn, 1998] that transforms a recursively-defined structure by recursing on it and applying a node-specific function to the results on substructures.

After simplifying the RELAX NG abstract syntax tree, we generate the COA from it. This transformation recurses on the RE-

```
element name { xsd:string }
  element age { xsd:int }
element person { name, age }
element data { name | age }
```

Figure 6.6: Selecting whether to enter a subautomaton

LAX NG structure using again the catamorphism technique. We implemented the catamorphism by specifying trivial processing for every piece of RELAX NG syntax and then replacing these as the implementation progressed. This made it easy to gradually develop the system and to leave out the processing of the interleave operator without affecting anything else. We call the intermediate results of this process *subautomata*.

The main construct to process for the automaton generator is the `element` construct. After all, elements are the most common pieces of XML syntax, and the regularity of their placement offers the most benefits for our event omission semantics. The processing of the grouping constructs did prove interesting, as they necessitated the addition of new semantics for the intermediate form of the constructed COA.

In general, it is not possible to determine, when transforming a language construct into a subautomaton, whether entry to that subautomaton happens always or only sometimes. For example, if an element is the second item in a `group` construct, it will always be present, but if it is a part of a `choice` construct, it might not appear in the processed document. Therefore the decision of whether to omit an initial event cannot be made yet when processing the piece of syntax that produces that event.

An example of this is illustrated in Figure 6.6. Here the subautomata for `name` and `age` are always used inside the `person` element, but only one of them is used inside the `data` element. Thus, in the former case it is possible to omit the ELEMENT START and ELEMENT END events of both `name` and `age` elements, but in the latter case it is not possible to omit the ELEMENT START events.

A subautomaton will need entry and exit points that are used to attach it to the higher level constructs that get created. Because of the issue described above, we implemented two entry and exit points for each subautomaton, the *known* and *unknown* points. The

```
                element pair { seq, seq }
seq = element seq { element item { xsd:int }* }
```

Figure 6.7: A problematic use of the star construct

known points will be used when it is known that the subautomaton itself will be used; otherwise the unknown points are used.

The entry and exit points in the EOA are states whereas in the DOA they are transitions. Use of states was simpler, but the more complex process of DOA construction could not be implemented with states. Recently, we have also implemented the EOA construction using transitions as the entry and exit points [Kangasharju and Koskimies, 2008]. This did prove more complex, necessitating the inclusion of `peek` transitions to the EOA as well.

Using states as entry and exit points introduced the problem of chaining the subautomata. To solve this, we define, at build time, equivalences between states, e.g., when two subautomata are grouped consecutively, we mark the first one's exit point as equivalent with the second one's entry point. After the complete automaton is constructed we *collapse* each set of equivalent states into a single state. We also reduce the constructed automata to the start state's strongly connected component, i.e., to those states which are mutually reachable from the start state.

Repetition constructs also have some interesting points. An example is provided in Figure 6.7, which shows two consecutive elements both containing a sequence of indeterminate length composed of the same elements. In this case it is known that these subautomata will be used, so naïve processing would omit all ELEMENT START and ELEMENT END events, thus destroying the information of where the boundary between the sequences was.

For this reason, we added the concept of *open* subautomata. An open subautomaton is one whose length is determinable only by the presence of an ELEMENT END event in its containing element, and not by anything internal. For the repetition constructs we build such an open subautomaton, and the builder of the element subautomaton will always construct the known exit point identically to the unknown exit point (note that the beginning is not indeterminable, so the known entry point can still be different from the unknown entry point).

This concept could also be used to provide additional schema evolvability. Marking a subautomaton as an open one would allow the addition of new content at the end of the corresponding element's content, since the default transitions would let all content through until the ELEMENT END event. While there is no direct support for such specification in our current implementation, its addition would only require local modification to recognize it and no modification of other processing.

Finally, we need to have special processing of optional components in a `group` construct on the DOA side. Normally, a `group` construct will chain its subautomata, connecting each exit point to the next subautomaton's entry point. However, in the presence of optional components, a connection also needs to be made to the subautomaton following the optional component. To handle this case, we mark the subautomata of optional components specially in DOA construction and handle them when constructing a subautomaton from the `group` construct. On the EOA side there is no need for this, as we just mark the entry and exit points of the optional component to be equivalent.

6.4.4 Automaton Build Rules for RELAX NG Constructs

Above we have covered on a general level the building of the COA from a RELAX NG schema. To provide some concreteness to our description, we next go over some of the RELAX NG constructs and show how they are converted into a COA. In these examples an M (possibly with a subscript) denotes either a part of a schema or a subautomaton constructed from that schema.

The automata we present will also show whether their known or unknown entry and exit points are used. These are indicated with a k or a u, respectively. We adopt the convention that entry points are always at the left and exit points at the right. Furthermore, we also mark the exit point of an open subautomaton with an o and that of an optional construct on the DOA side with a q. These markings appear only when introduced in the construction.

We begin by showing the `element` case in Figure 6.8. In all Figures, we shorten event names, transition types, etc., to single letters whose meaning should be clear from the context. We show the normal case and the case where the subautomaton is an open

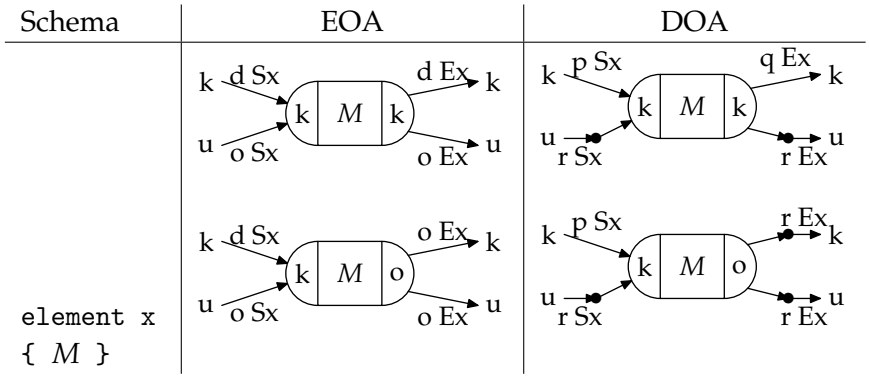


Figure 6.8: Subautomaton construction for `element`

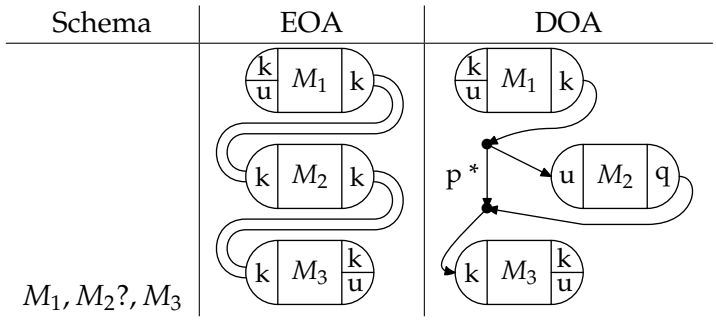


Figure 6.9: Subautomaton construction for `group`

one. Note that in the case of an open subautomaton the known exit point is constructed in the same way as the unknown one.

Most of the constructs in RELAX NG only take subschemas as arguments, so they will not produce events in the transitions. Apart from the `element` construct, only the `attribute` and `data` constructs produce events for transitions; the others may transform existing transitions, but will not produce new ones.

The next case we cover is `group` in Figure 6.9. On the EOA we have marked a double line to indicate that one subautomaton's exit point is marked equivalent to the next subautomaton's entry point. These equivalent states will then be collapsed to a single one at the end. The constructed automaton will have its known and unknown entry points be the same as the first subautomaton's, and analogously with the exit points and the last subautomaton.

On the DOA side we see that the M_2 subautomaton is marked

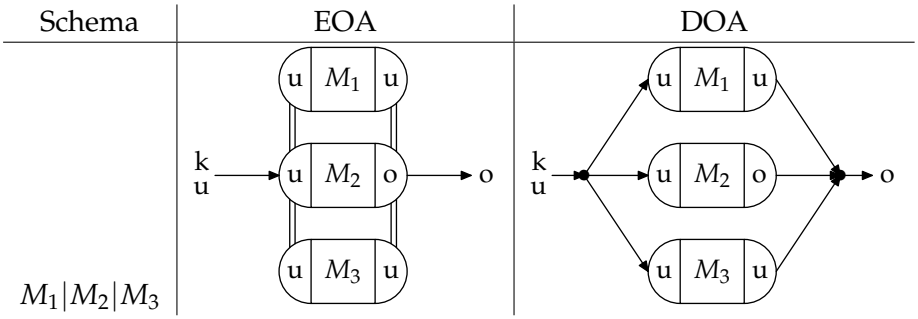


Figure 6.10: Subautomaton construction for `choice`

optional. This is not shown in the EOA construction, but the result is that in the EOA M_2's entry and exit points get marked equivalent, and thus collapsed at the end of automaton construction.

As we see from the DOA construction, the grouping here creates two additional states. Using the unknown entry point for M_2 ensures that it will be recognized if it is present. The `peek` transition between the two new states will be taken if M_2 is not entered, so the processing can continue with M_3. Note that since the most specific transition is always selected, the `peek` transition can be made only if M_2 is not entered.

In full, the DOA-side processing of the `group` construct is very complex. In our implementation it takes nearly 100 lines of code whereas the next largest, `element` processing for either the EOA or the DOA, only takes 30 lines. Our example can only capture a part of this complexity, since it is the result of needing to handle several different cases depending on the types of the subautomata.

The final interesting subautomaton construction is the `choice` construct in Figure 6.10. On the EOA side we need to select the unknown entry and exit points for each subautomaton, as we cannot know which option in the choice is taken by the document. As can be seen, the entry and exit points of the subautomata are collapsed. Furthermore, both known and unknown points of the constructed automaton are the same, and the constructed automaton is an open one if even one of the alternatives in the choice is.

The DOA is very similar to the EOA, except that since its entry and exit points are transitions instead of states, the construction will create a new state to scatter the entry points and to gather the

exit points. Again, as in the EOA, the entry point selects all the unknown entry points, and the exit point selects the unknown exit points, and is open if even one of the subautomata is.

6.5 Summary

The design of Xebu always followed the requirement to run well on mobile devices. Among well-known formats with published specifications, Xebu would appear to be the only one possessing all of the following features:

- small-footprint implementation,
- efficient in both time and memory usage,
- ability to process any XML, and
- ability to use schema information to different extents.

The EXI format may turn out to have these properties as well, but it is too early to tell whether this is the case.

The main feature that distinguishes the basic Xebu format from other formats is its bounded memory usage, accomplished using a bounded space of possible tokens. The presence of the token values permits less coupling between serializer and parser, but also leads to larger document sizes. This could be mitigated for small documents by not including the token value until the token space has been completely used.

The schema usage in Xebu is also a novel approach, as most schema optimizations require perfect adherence to the schema. Its utility is somewhat lessened, though, since we have no precise knowledge of the permitted deviations. Furthermore, the COA has some limitations that make it less usable on some real-world schemas, namely ones with mixed content or recursive elements.

Despite the potential future prominence of the EXI format, we still believe that Xebu is not completely obsoleted by it. The main benefit of Xebu is an available implementation with well-understood properties under a F/OSS license. It may take some time before a suitably efficient and small EXI implementation for mobile devices materializes, and at least until that happens, we consider Xebu to be a usable alternative.

Part IV

Measurements and Analysis

CHAPTER 7

Messaging System Measurements

> You know nothing, Jon Snow.

We performed several experiments on the messaging system. First, we measured the actual messaging performance under a variety of parameters. Second, we measured the performance of our security implementation separately, as adding security features to the main measurements would have obscured the actual messaging performance. Third, we measured the performance of the Xebu format on its own, as we consider that to be the most likely component to be adopted on its own. Finally, we measured the overhead of the AMME protocol and its RTT measurement accuracy.

We use two principal methods for reporting the results. When there is sufficiently little data to present, only two or three parameters that vary, results are shown with bar charts or graphs. When applicable, error bars are marked at two standard deviations, representing the 95 % confidence interval. In other cases, where there is more data to present, or when the presented results inherently require more than two dimensions to present conveniently, tables are used. In these tables we simply present the computed averages for each measurement, which are in some cases fully accurate (e.g., when measuring sizes), but in some cases may have minor variance.

7.1 Component Sizes

As application footprint is important on mobile devices, we began by measuring the sizes of the compiled class files. Our deployment process for the system followed the standard steps [Knudsen and Li, 2005]:

1. *Compilation*: Compiling the source code into Java bytecode with a Java compiler using the MIDP class libraries,
2. *Preverification*: Performing some of the security checks required by Java in the compilation environment and adding information on these into the class files, to reduce the checking the device has to do at run time, and
3. *Obfuscation*: Removing unused classes, methods, and fields, and shortening class, method, and field names to one or two letters to reduce code size.

We used the Java tools from Sun Microsystems (JDK 6 and Wireless Toolkit 2.2) for compilation and preverification, and Proguard[1] for obfuscation.

The full breakdown of the sizes of all the components is shown in Table 7.1, grouped with related components together. In the XML group, the XAS-base consists of only the basic XAS described in section 5.1 while XAS-ext includes all of XAS. The Xebu component includes, in addition to the actual parser and serializer, also the code to integrate Xebu into XAS such as the typed data codecs.

The Messaging group is the main system and protocol. MTS-base is the user-visible part described in section 4.2, AMME is the full protocol part, and WSS is our Web Services Security implementation. As most of the functionality needed for WSS is provided by the XML security implementation in the XAS-ext component, the specific WSS component is quite small.

In the AMME Transfer layer mappings, the Stream component is common to the BT and TCP mappings, while the HTTP mapping is independent of these. This group is included in the AMME component of the Messaging group as well; the rest of the code in that component is the Mobility layer.

[1] http://proguard.sourceforge.net/

Table 7.1: Sizes of messaging system components

XML		Messaging		AMME Transfer	
Name	Size (B)	Name	Size (B)	Name	Size (B)
XAS-base	48566	MTS-base	17138	Stream	8940
XAS-ext	106370	AMME	50164	TCP	3581
Xebu	30771	WSS	5887	BT	11083
				HTTP	13263

External		Totals	
Name	Size (B)	Name	Size (B)
kXML	23714	Total	506148
BC	111022	JAR	447728
JZlib	68147	JAR obf.	302761

We use some external components in the system as well. Regular XML parsing and serialization is handled by kXML[2], low-level security functionality by Bouncy Castle (BC)[3], and gzip compression and decompression by JZlib[4]. These are the libraries usually used for these purposes on mobile devices, but we note that the sizes of Bouncy Castle and JZlib are quite large.

Finally, the total sizes include the total combined size of all the class files, and the size of the JAR file. Since the JAR file is a compressed archive, its size is naturally smaller than the total size of all the classes. We show the JAR size both partially obfuscated and fully obfuscated; the difference is that in the former the names of classes, methods, and fields were not altered, but all unused classes, methods, and fields were still removed.

7.2 Platforms

In our experimentation, we used a variety of real devices and networks, with the main purpose being to get an impression of the

[2] http://kxml.sourceforge.net
[3] http://www.bouncycastle.org/
[4] http://www.jcraft.com/jzlib/

Table 7.2: The devices used in the experiments

Name	Roles	CPU	Description
E61	Client	219 MHz ARM9	A Nokia E61 smartphone
9500	Client, Server	163 MHz ARM9	A Nokia 9500 Communicator
brattain	Server	AMD Athlon XP 1700+	A desktop computer

factors that affect a messaging system. We used both very recent technology to provide an anchor point for the current performance as well as slightly older technology so that it might be possible to extrapolate some trends towards the future.

7.2.1 Devices and Networks

The devices we used in our experiments are shown in Table 7.2. Both the E61 and the 9500 are "business" phones, the 9500 being from the year 2004 and the E61 from the year 2006. Both support MIDP 2.0 and CLDC 1.1, which are required for our current implementation. The 9500 also supports the more capable FP, but as our target platform is the more prevalent MIDP, our measurement applications were also written for that profile.

The server machine brattain was an oldish desktop computer, appropriated as a test server after reaching its end of life as a desktop system. We did measure processing time spent on that machine, but noted it to be negligible compared to the communication time and processing time on the phone, so it will not be considered. Neither will we show processing times on the 9500 server, as the application using it is symmetric, so the processing times should be equal on both sides of the connection.

The network technologies supported by the devices we used are shown in Table 7.3. Except for UMTS on the Nokia 9500, all of these are supported on both phones. Two of the networks are ad-hoc technologies, so the server is also a phone, and three are Internet connectivity technologies, for which we use brattain as the server.

Table 7.3: Networks used in the experiments

Name	Clients	Server	Description
BT	9500, E61	9500	Bluetooth
Ad-hoc	9500, E61	9500	An ad-hoc WLAN
Infra	9500, E61	brattain	An infrastructure WLAN connection to the Internet
GPRS	9500, E61	brattain	A regular consumer-provided GPRS connection
UMTS	E61	brattain	A regular consumer-provided UMTS connection

Table 7.4: Names for network-protocol combinations

Name	Net	Proto	Name	Net	Proto
BT	BT	BT	aT	Ad-hoc	TCP
iT	Infra	TCP	iH	Infra	HTTP
gT	GPRS	TCP	gH	GPRS	HTTP
uT	UMTS	TCP	uH	UMTS	HTTP

Considering the available networking technologies, the protocols supported on each network, and the protocols supported on each server device, we arrive at the eight different combinations of network and protocol shown in Table 7.4. This Table also gives short names for each combination that are used when showing the results later.

7.2.2 Timing Measurements with Java

The Java compilation model is not the standard one where Java source code is translated directly into machine-executable code. Rather, the Java compiler translates source code into *Java bytecode*, which is then executed by the *JVM* [Lindholm and Yellin, 1999]. A modern JVM will not simply interpret the bytecode as such, but will apply *just-in-time (JIT) compilation* techniques to compile the executed bytecode at run time into native machine code. This makes timing measurements more difficult, as the executed code changes while the measurement is in progress.

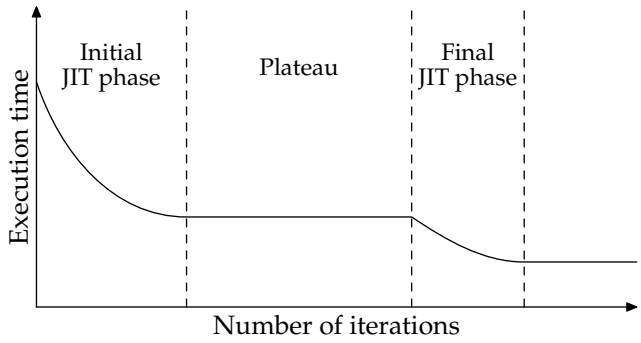


Figure 7.1: The plateau effect in JIT compilation

The standard method to remove the effect of JIT compilation in Java-based measurements is to start with a *warmup loop* that runs the measurement loop without gathering any measurements. After the loop has run sufficiently long, the actual measurement can start. Typically, the length of the warmup loop is determined experimentally, by increasing its length until the actual measurement stabilizes. A more advanced, though also more rarely used, method would be for the measurement framework itself determine when the measurement has stabilized. The risk with the latter technique is that the measurement may have a lengthy plateau where it does not change, but after an even longer time, the JVM performs even more optimization on the code, changing the measurement value again, as illustrated in Figure 7.1. We did not observe this plateau effect in our measurements.

7.2.3 Benchmarks

The characteristics of the devices and networks we used are available information, and have been covered above for the specific devices and in section 3.1 for the networks and on a more general level for the devices. However, to properly explain results of measurements, it is beneficial to understand how the devices and networks that were used actually behave. To this end, we implemented some simple benchmarks that we ran in the exact same conditions as the actual experiments. The full code of all the benchmarks can be found in Appendix A.

In addition to the devices shown in Table 7.2, we also ran the

Table 7.5: Additional devices used in the experiments

Name	CPU	Description
7610	123 MHz ARM9	A Nokia 7610 smartphone
6630	220 MHz ARM9	A Nokia 6630 smartphone

benchmarks using the two devices shown in Table 7.5. These devices are older or less capable than the others (both are from the year 2004), so running the benchmarks on them as well provides useful data for extrapolating the performance of future devices.

Our first benchmark, Knuth, was intended to measure the CPU speed and give some indication of the JIT compilation abilities of the JVM. For this, we used the random number generation algorithm recommended by Knuth [1997b]. In theory, such an algorithm might also be affected by memory speed, but the amount of data used here should be small enough to fit in the CPU cache.

The measurement in the Knuth benchmark was filling a 1000-element array with random numbers. Each timing loop performed this operation 300 times, which we determined to be appropriate to get large enough numbers that the clock granularity does not pose an issue but small enough that the first run is not significantly run with JIT-compiled code. We did not use a warmup loop, but rather measured all replications to note the JIT effect. The stable performance is computed by starting from the first measurement that is not larger than all of its subsequent measurements.

The results of the Knuth benchmark are shown in Figure 7.2, showing the time for the initial run for each device and the average time for the stable part. Interestingly, the initial run is slightly faster on the 7610 than on the 6630 and E61, even though the latter two have much faster CPUs. Apparently also the 9500 has started JIT compilation already on the initial run, as it is clearly faster on that but slower than the 6630 or E61 on the stable run. Overall, the stable results appear to be in line with the CPU speeds as shown in Table 7.2 and Table 7.5.

The second benchmark we performed, Virtual, was intended to determine the method call overhead of the JVM with two parameters, the length of the method name and invocation through an interface. The length of the method name would not seem to affect

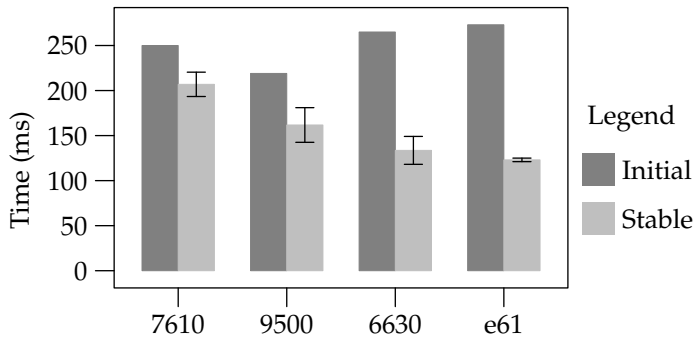


Figure 7.2: Results for the Knuth benchmark

real applications on mobile phones, as the obfuscator will shorten the names to one or two characters. However, the standard Java libraries still contain long method names, so obfuscation does not eliminate this problem completely.

In this benchmark, the method name takes on values having lengths of 1 (Short), 14 (Long), and 24 (Longest) characters. These methods belong to an interface with two implementations. There are three different styles of invocation: Same always invokes on the same object, Flip invokes on the two objects in a round-robin fashion, and Rand invokes on a randomly-chosen one of the pair. All methods use the random number generator above to generate 100 random numbers. The measurement loop is 1000 rounds and the loop is replicated 50 times. These values were chosen experimentally to provide a sufficiently long time to mitigate the effect of potentially-coarse timing granularity.

The complete results of the Virtual benchmark are shown in Table 7.6, given for each device separately. Each cell in the Table contains two numbers; the top one is the initial run and the bottom one is the stable run.

There does not appear to be much difference in the JVMs on the two phones 6630 and E61 that have a similar-speed CPU, as the times are mostly very close to each other. Again, it is probable that the 9500 begins its JIT compilation already on the initial run, since it improves clearly less for the stable run than the 7610.

The effect of long method names is very pronounced, so it appears clear that obfuscation truly is a big win, not just in terms

Table 7.6: Results for the Virtual benchmark

7610	Same	Flip	Rand
	438	453	610
Short	328	385	487
	453	609	625
Long	391	466	522
	547	609	750
Longest	484	552	640

9500	Same	Flip	Rand
	344	469	515
Short	297	359	451
	453	500	516
Long	406	431	480
	562	500	657
Longest	444	479	600

6630	Same	Flip	Rand
	235	266	297
Short	188	219	273
	266	328	359
Long	230	274	321
	312	343	406
Longest	273	305	369

e61	Same	Flip	Rand
	219	249	295
Short	182	215	264
	236	294	338
Long	217	262	307
	282	342	396
Longest	264	309	368

of JAR file size but also in method call time. Also, it would appear that the JIT compilation on all phones is capable of recognizing when an interface has only one implementation. However, it must be kept in mind that the Flip style has an additional variable access, AND operation, and branch, and the Rand style has a variable access, a memory access, and branch, though the memory access is likely a cache hit. But each time has only 1000 of these additional operations, so it would appear that the pattern of selecting the implementing class does have an effect, indicating also that if there is only one implementation of an interface, invoking through the interface may be sufficiently efficient.

The final benchmark, Net, was intended to measure the latencies and data rates of the networks that we used. The first part of this benchmark repeatedly sent one byte to the server and read the server's one byte response, measuring the time this took. In the second part, the client sent one byte to the server, which in

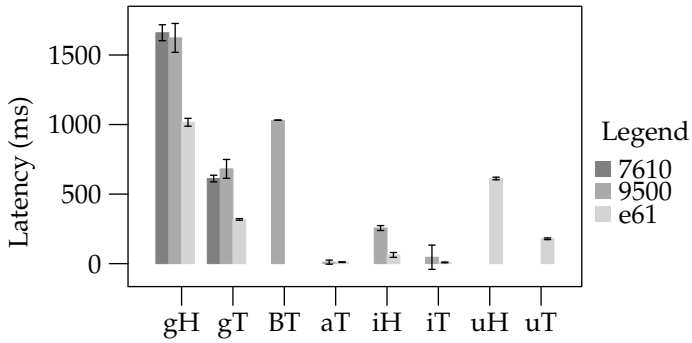


Figure 7.3: Latency results for the networks and devices

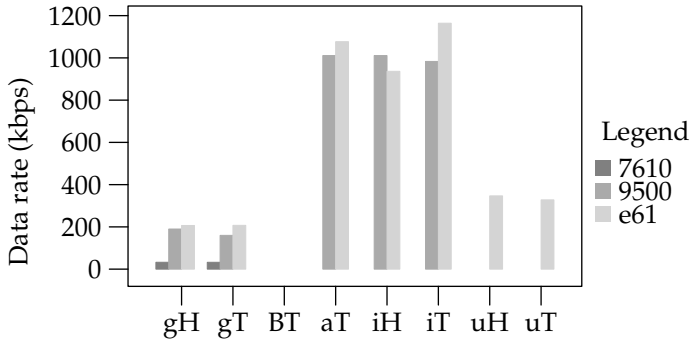


Figure 7.4: Data rate results for the networks and devices

response sent 2 MiB of random data to the client, and the client measured the total time taken by this process. In this case, we did not experiment with the 6630, since the two other benchmarks indicate its JVM performance does not differ markedly from the E61, and it supports a subset of the networks supported by the E61.

Figure 7.3 shows the latency values measured for each network on each device, and Figure 7.4 shows the data rates. Some of the measurements are missing due to problems with the Bluetooth implementation on the phones, which we will explain in section 7.8. In particular, the large Bluetooth latency is due to a workaround that must be implemented for long-term communication.

Looking at the latency figures, we see roughly what we would expect based on the measurements of others. One surprise is the GPRS latency of the E61, which is nearly one half of that measured

for the other two phones. UMTS latency is slightly lower than GPRS latency, but still in the 100 ms order of magnitude. HTTP latency is much worse than TCP, not only due to the additional network round trip but probably also due to the additional data that gets sent in the HTTP headers. The high variance of the 9500 with iT is caused by the coarse granularity of the 9500's internal clock: the clock ticks only once every 15 or 16 ms, so small and varying latency values will not be measured accurately.

For the data rate we clearly do not reach the theoretical rates listed in subsection 3.1.2. WLAN gave only 1 Mbps, and monitoring the data transfer during the experiment, we note that this is due to packet loss on the wireless link, causing the phone's TCP implementation to begin slow start again. Both the 9500 and E61 support EDGE whereas the 7610 does not, which is why its GPRS data rate is in the neighborhood of 30 kbps, while the other two phones approach 200 kbps.

7.3 Experiment Design

We ran four different experiments, described in Table 7.7. The Messaging experiment was our main experimentation tool, and in it we varied all possible parameters to determine precisely the factors that affect the system. The Security experiment exercised the XML security implementation described in section 5.4. We separated this into its own experiment because the messaging system does not yet support security in its API but also, and more importantly, security processing is mostly orthogonal to the other factors and its processing requirements dwarf every other factor. Finally, we measured the key properties of Xebu and AMME on their own.

The Messaging experiment exercised each of the three Transfer layer mappings that we implemented, each over every one of its supported networks, i.e., all eight network–protocol combinations shown in Table 7.4. This variety in one experiment, we believe, gives sufficient information to compare between different protocols, so in the security experiment we use the normal request-response interaction of HTTP over the UMTS network.

The Messaging experiment used both regular XML and Xebu as its message formats. In addition, we used both the plain formats

Table 7.7: List of experiments

Experiment	Description
Messaging	A simple request-response messaging application
Security	A request-response interaction with secured messages
Xebu	An experiment measuring Xebu document size and processing efficiency
AMME	Measurement of AMME header size and RTT calculation accuracy

Table 7.8: Formats used in the Messaging experiment

Name	Description
XML	Regular XML
XMLZ	Regular XML compressed with DEFLATE
Xebu	Basic Xebu
Xebuz	Basic Xebu compressed with DEFLATE

as well as generic compression in the protocol layer. This comes up to a total of four different formats, named in Table 7.8.

Of the formats shown in Table 7.8, XML and XMLZ are formats that are widely supported in current systems. We contended in subsection 5.4.3 that these formats are not sufficient in the presence of encrypted content, but an additional way to indicate compressed content in encrypted XML data is required. Therefore in the Security experiment we also use two additional formats, shown in Table 7.9, that compress data also prior to encryption. This is in contrast to the XMLZ and Xebuz formats, which only compress the whole message after encryption is performed.

Finally, we have some additional parameters that we also vary. Each of the parameters shown in Table 7.10 is used in the Messaging experiment. We expect the invocation style to make a large difference in the results. In the Xebu experiment we only vary message size of these parameters, as we do not believe the networking to make a difference there due to the messaging system having to store the messages for potential retransmission. The possible

Table 7.9: Additional formats used in the Security experiment

Name	Description
XMLZZ	Regular XML compressed with DEFLATE, including encrypted parts
Xebuzz	Basic Xebu compressed with DEFLATE, including encrypted parts

Table 7.10: Other variables in the messaging experiment

Variable	Description
Message size	1 or 10 `card` elements
Number of messages	2 or 20
Invocation style	Synchronous or asynchronous

values for the two numeric parameters were selected to provide a reasonable spread but still be realistic.

The messages in the experiments contain `card` elements, signifying credit card information that is being transmitted, i.e., a name, number, and expiration date. A message contains a single `data` element that contains the number of `card` elements denoted by the size parameter. The full schema of the messages in the Messaging and Xebu experiments is given in Appendix A.4. The Security experiment uses a slightly different format, described in section 7.5.

7.4 Messaging Results

The Messaging experiment used a full factorial design [Jain, 1991] of the components enumerated above. The factors are summarized in Table 7.11. Here the two UMTS protocols uT and uH are not available on the 9500, but otherwise all parameter combinations were used. Format and Gzip were made separate factors in the analysis. And as the set of available protocols is dependent on the network, the network and protocol are joined into a single factor, called Net, using the short names from Table 7.4.

Table 7.11: All experimental factors and their possible values

Device	Net	Format	Gzip	Size	Number	Style
E61	BT	XML	No	1	2	Synch
9500	aT	Xebu	Yes	10	20	Asynch
	iT					
	iH					
	gT					
	gH					
	uT					
	uH					

Table 7.12: The main factors in the Messaging experiment

Factor	Percentage
Number	36.6 %
Style	14.2 %
Style+Number	11.5 %
Net	9.5 %
Net+Number	5.8 %

7.4.1 Analysis of Variation

As can be expected, an analysis of the complete Messaging experiment results, while interesting, is not particularly illuminating due to the large amount of data and its variability. However, it serves as a useful starting point to determine the factors that should be used to divide the analysis. Therefore we show the factors whose contribution to the total variation is more than 5 % (as recommended by Jain [1991]) in Table 7.12. In this analysis, errors contribute an effect of approximately 1 %.

The results are not very surprising. It is quite natural that the number of messages has a large effect on the total messaging time, and the large effects of messaging style and network used would be expected, especially when considering the variety in the networks used. Of the factors not shown, neither the message format nor compression usage contributes anything above the level of errors. The effect of the device is larger, though not quite 5 %.

Table 7.13: The main factors for fixed number of messages

2 messages		20 messages	
Factor	Percentage	Factor	Percentage
Device	36.6 %	Style	40.9 %
Net	28.2 %	Net	24.0 %
Device+Net	8.6 %	Net+Style	13.5 %
Style	7.1 %	Device+Net	5.4 %

The experimental design itself was intended to cover a large amount of ground. Accordingly, many of the factors have values that might not all be relevant to a single use case. This is why it is educational to fix the value for some such factor and to analyze the remaining data while keeping this factor fixed. Then the further results are applicable to a certain subset of messaging applications, but potentially there is little overlap between applications with different values for the fixed factor.

As the Number factor is the one with the largest effect, we split all later analysis on that into two branches, one with 2 messages and the other with 20 messages. Table 7.13 shows, again, the factors having an effect of over 5 % in each of these cases individually.

As could be expected, invocation style does not become the major effect until the number of messages increases. However, it is still present in the Table even in the 2-message case, indicating that even for infrequent messaging there may be a benefit from asynchronicity. The device plays quite a major role in the 2-message case, but especially looking at the latencies in Figure 7.3 we note that there is a large benefit over many networks in favor of the E61. And as we did not yet eliminate the network factor, it begins to feature more prominently on the lists. Errors are starting to have a larger effect, contributing 3 % to the 2-message case.

Next we eliminate the largest factor shown in Table 7.13 for both of the cases, i.e., we fix the device for the 2-message case and the style for the 20-message case. The first of these, the effects in the 2-message case for the 9500 and the E61 individually, is shown in Table 7.14, and the second, the 20-message case for synchronous and asynchronous style, in Table 7.15.

In the 2-message case, the network has the largest effect for both

Table 7.14: The main factors for fixed device with 2 messages

E61	
Factor	Percentage
Net	42.5 %
Style	22.5 %
Size	8.6 %
Error	5.1 %

9500	
Factor	Percentage
Net	63.1 %
Style	8.6 %
Net+Style	7.7 %
Gzip	7.3 %
Error	5.2 %

Table 7.15: The main factors for fixed style with 20 messages

Synchronous	
Factor	Percentage
Net	66.3 %
Device+Net	14.7 %
Size	5.0 %

Asynchronous	
Factor	Percentage
Net	20.9 %
Size	18.7 %
Device	14.4 %
Error	7.3 %
Net+Gzip	7.1 %
Device+Net	7.0 %
Format	6.6 %

devices. Due to the worse network performance on some of the networks on the 9500, it has a larger effect there, whereas on the E61 with better network performance the invocation style features prominently. Interestingly, with two major factors held constant, message size is beginning to have a larger effect. On the E61 it is the abstract size of the message, represented by the number of elements, whereas on the 9500 the concrete size matters more, signified by added compression having a large effect.

As with the 2-message case, the 20-message case also shows the network's prominence. As with synchronous invocations the latency of the network is crucial, and since the latencies in the experiment networks have large differences, the network has a major effect in that case. In the asynchronous case where latency matters less and the data rate is not a problem with these message sizes, the network does not have quite that large an effect.

In the asynchronous case, message size is starting to matter

Table 7.16: The main factors for each net with 2 messages

Net	Factors
BT	Size (55.8 %), Device (15.9 %), Gzip (10.7 %), Format (7.1 %)
aT	Device (29.1 %), Gzip (20.5 %), Size (16.1 %), Device+Gzip (15.8 %)
iT	Device (52.9 %), Device+Gzip (21.0 %), Gzip (7.2 %), Size (6.2 %)
iH	Device (84.1 %), Error (8.7 %)
gT	Device (55.1 %), Style (19.3 %), Size (8.5 %), Error (5.9 %)
gH	Device (67.7 %), Style (19.9 %)
uT	Style (42.7 %), Gzip (19.2 %), Error (14.5 %), Size (10.4 %), Gzip+Style (5.0 %)
uH	Style (65.1 %), Size (9.7 %), Error (6.0 %), Gzip+Style (5.1 %)

quite a bit, with Size, Gzip, and even Format having an effect of more than 5 %, and there is no clear major individual effect. The effect of errors is starting to mount, but not in the synchronous case. We also note that the device has an effect in both cases, but it has a much larger effect in the asynchronous case, potentially caused by the prominence of the message size.

As the network had the largest effect in all four cases where two factors were held constant, we next investigate the effects when the network is fixed. We keep the division into the 2- and 20-message cases, but that is the only other thing we keep fixed. As the number of networks is so large, we change the presentation format slightly in Table 7.16 and Table 7.17 that report the effects when the number of messages and the network are kept fixed.

There are some things to keep in mind when looking at these Tables, namely that not all possible measurements were made. As UMTS is not supported on the 9500 but only on the E61, the Device factor cannot have an effect in either of the UMTS networks. Also, due to problems with the Bluetooth implementation, it was not possible to use the asynchronous style over Bluetooth, so invocation style cannot appear as a factor in the Bluetooth case.

Table 7.17: The main factors for each net with 20 messages

Net	Factors
BT	Size (56.1 %), Gzip (14.5 %), Format (12.8 %), Format+Size (8.1 %), Gzip+Size (6.3 %)
aT	Style (36.3 %), Size (23.2 %), Device+Style (8.8 %), Gzip (7.7 %)
iT	Style (38.1 %), Device (14.8 %), Device+Style (10.8 %), Size (10.2 %), Error (8.7 %), Device+Gzip (8.1 %)
iH	Style (51.0 %), Device (27.6 %), Error (7.3 %)
gT	Style (81.2 %), Size (7.3 %)
gH	Style (77.4 %), Device (14.3 %)
uT	Style (79.2 %), Gzip (6.4 %)
uH	Style (88.1 %)

We can see that with the smaller number of messages in Table 7.16, the device has the largest effect, except over Bluetooth. However, on the better-performing networks, WLAN and to some extent even UMTS, message size is becoming prominent through the Size and Gzip factors. Invocation style is important for high-latency networks but has little effect on low-latency WLAN.

For the larger number of messages in Table 7.17, invocation style is the overriding concern on the mobile phone networks, explaining nearly all variation there. It also features prominently on WLAN, as the number of messages becomes so large that the synchronous style suffers from many additional round trips over the network. Size-related factors make a clear appearance on the lists, though not with HTTP.

One consideration that did not make a very visible appearance was the message format. It would appear that, with all the other factors present, it eventually matters very little whether the messages are serialized as XML or Xebu. However, this applies only to the time spent for messaging and not to, say, energy consumption. While the precise amount of data sent does not matter to the execution time, it has a large effect on the battery life, so these results are not indicative that a binary format is a useless idea.

7.4.2 Graphical Presentation

Due to the large number of different factors, it is not feasible to draw graphs explaining everything that happens. Because of this, we will here limit our consideration to only the E61 device and the 10-element message size. As the E61 is a newer model, its performance is more indicative of the current and near-future state of the art. As will be shown in section 7.6, the size differences of the messages do not yet appear in the 1-element case, so that is why we selected the 10-element case for that. We also note that the 2-message case is not very interesting, so we limit consideration to the 20-message case.

With these limitations it suffices to display one graph in Figure 7.5 that shows both invocation styles on all networks and considers all message formats. Note again that asynchronous results for Bluetooth are not available, and the Xebu bar for synchronous Bluetooth is also missing due to similar reasons.

As would be expected based on the analysis of variation, the times in the synchronous case are all clearly larger than those in the asynchronous case. Furthermore, the improvement in performance when moving to asynchronous style is clearly larger for the higher-latency mobile phone networks. There is still a minor advantage for WLAN even in the asynchronous case, though.

An interesting point to note, which is not evident in the analysis of variation due to the presence of so many other effects, is that the combination of format and gzip does have a clear effect on most of the individual networks. However, this effect is not unambiguous across all measurements. In the synchronous case, message size reduction helps, and gzip performs better than either of the formats, most probably due to the need for fewer TCP segments. On the other hand, in the asynchronous case, the network is constantly busy, so the added CPU time of compression and decompression acts to increase the total processing time, especially over WLAN.

We also note that the ad-hoc technologies, WLAN and Bluetooth, perform markedly worse than infrastructure-based WLAN. When considering the capabilities of the server, this appears self-evident. After all, in the Ad-hoc and Bluetooth cases the server is a 9500 that has to perform essentially the same actions as are performed by the client, doubling the total CPU time.

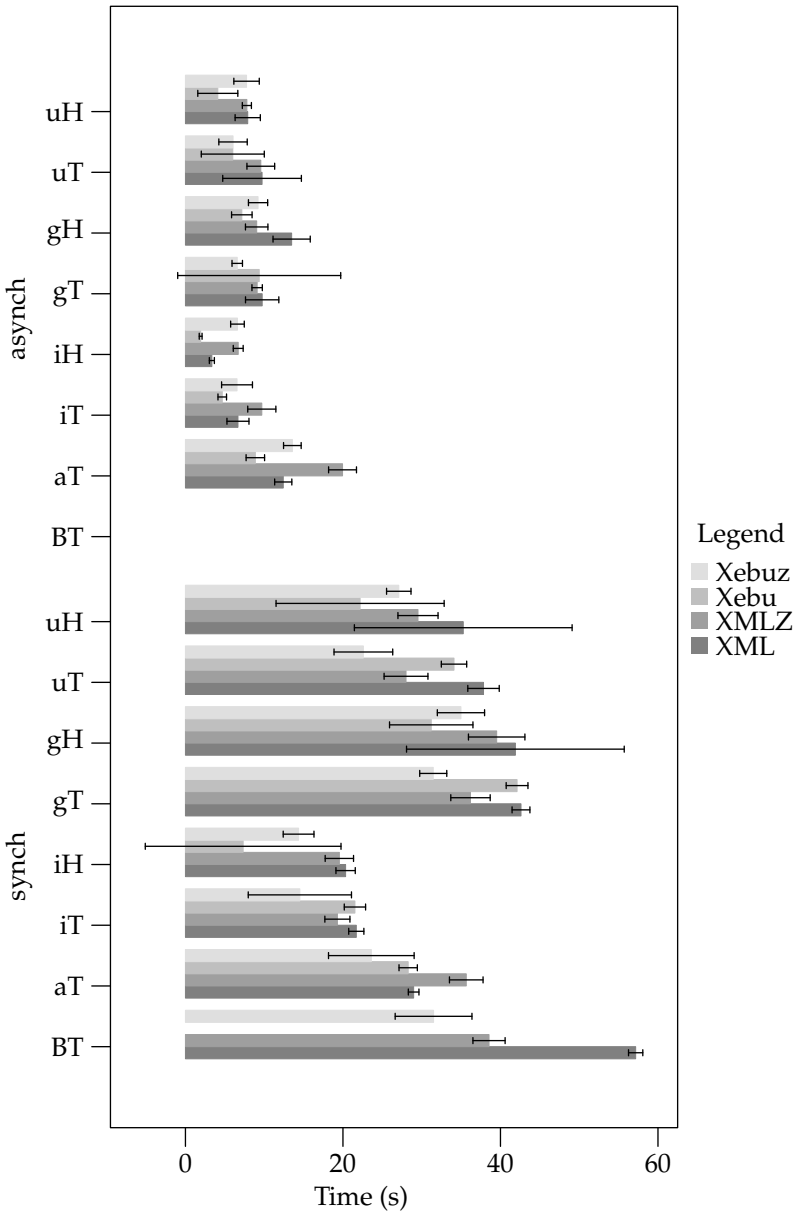


Figure 7.5: Messaging results by style, network, and format

Comparing the relative performance of TCP and HTTP is also illuminating. Namely, there does not appear to be a significant performance difference in favor of either, despite HTTP being a heavier protocol. We consider these results to validate the AMME approach, in particular the HTTP Transfer mapping and the message compaction module.

7.5 Security Results

While we have implemented both XML Signature and XML Encryption in our system, the messaging API is mostly not XML-based. Because of this, there are currently no facilities in the message system API for securing messages, since as of yet there has been no user of such functionality. According to our design principle of not implementing functionality before it is required by some application, we have therefore not yet designed a security API.

Because of the lack of security support in the message system proper, the Security experiment client uses another SOAP format, where the body consists of the `data` element of the Messaging experiment schema of Appendix A.4 and the header contains a WS-Security header. The client signs this message and then encrypts it, the order recommended by experts [Ferguson and Schneier, 2003], filling in the WS-Security header. The processed message is then sent as an HTTP request to the server, which echoes it back. The client decrypts the response and verifies the signature in it.

The security processing includes a number of phases, shown in Figure 7.6. We measure the time taken by each phase individually, which is intended to point to the areas most in need of improvement. We also measure the total time taken in serializing and parsing the message, including all security processing, and the time it takes for communication. As mentioned above, we used only the E61 with regular HTTP over the UMTS network.

As the Security experiment had fewer factors, we extended the number of different message sizes. We varied the number of elements from 5 to 50 at 5-element increments. The runs were made on each of the six formats from Table 7.8 and Table 7.9. We used 100 replications, split into smaller chunks that were run at different times to eliminate effects from temporary fluctuation in connec-

Serialization	Parsing
Symmetric key generation	Key decryption
Key encryption	Symmetric decryption
Serializing for encryption	Parsing decrypted data
Symmetric encryption	Digest verification
Digest computation	Signature verification
Signature value computation	

Figure 7.6: Phases in the Security experiment

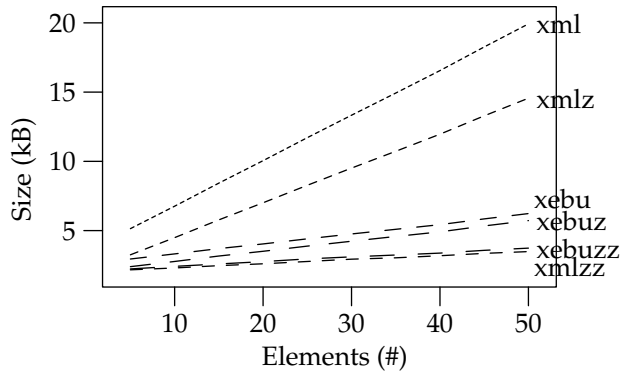


Figure 7.7: Security experiment message sizes

tion quality (originally we ran all replications consecutively and did see clear fluctuation effects).

The sizes of the messages are shown in Figure 7.7. Recall that the formats with one z in the suffix use only HTTP-level compression that can do little with encrypted data whereas the formats with two z's compress the contents of the `EncryptedData` element prior to encryption, and also apply compression at the HTTP level.

From the sizes of XMLZZ and Xebuzz we see that Xebu does not, at least in this case, compress any better than XML. However, XML size improves markedly even with HTTP-level compression whereas Xebu is hardly affected at all. The reason for this is that the encrypted content is a sequence of essentially arbitrary bytes that needs to be base64-encoded for inclusion into an XML document. The compression then eliminates the redundancy of this encoding, decreasing document size by approximately 1/4 for large messages that are mostly payload.

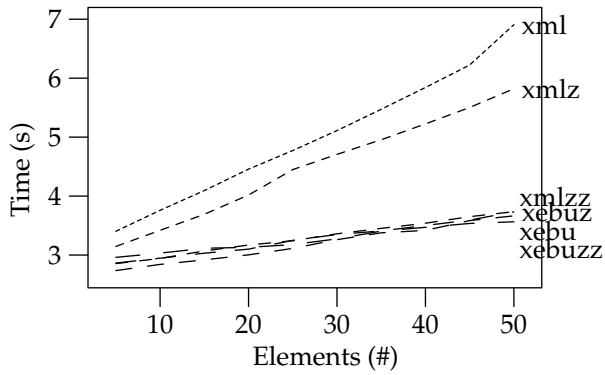


Figure 7.8: Security experiment total times

The measured total times, from when the application begins serializing the request to when it has finished parsing the response, are shown in Figure 7.8. There is clear correlation with the message sizes, but it is not perfect, as the Xebu formats do comparatively better than when looking at the size alone. The jump in XML performance for the last case is an interesting one, which we will explain later when it is even more visible.

The times taken for communication are shown in Figure 7.9. In contrast to the other measurements, we took the minimum values in this case. We saw very large variance in the timing results, making computation of means somewhat fruitless, as the results did not follow any sort of clear trend, and even had huge jumps in places. This measurement therefore represents optimal conditions in the network. The results shown in Figure 7.8 also use these minimum communication times.

Communication time, unlike total execution time, is practically perfectly correlated with message size. As the server does not perform any processing, this is the expected result. We note that the messages here are small enough that if we were to compute the data rate of the connection based on the message sizes and these communication times, we would not get results anywhere close to the data rate shown in Figure 7.4. This is, again, an artifact of TCP's slow start algorithm.

The other two components of the total time are serialization, shown in Figure 7.10, and parsing, shown in Figure 7.11. Unlike

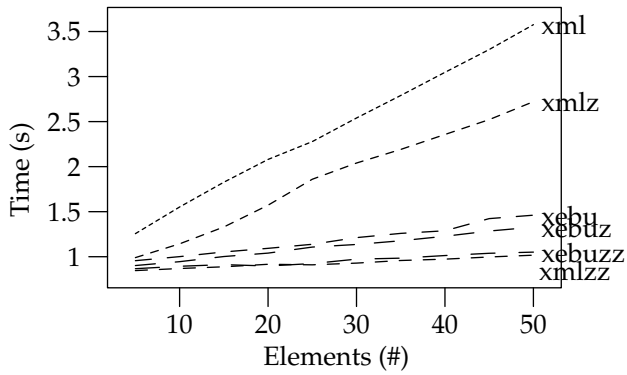


Figure 7.9: Security experiment communication times

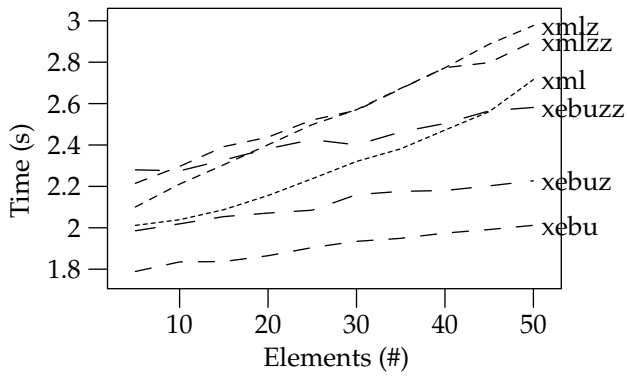


Figure 7.10: Security experiment serialization times

the communication times, there was little variation in these results, so the graphs here show means, which are also the components of the total times in Figure 7.8. Note that the X axis shows the number of elements, and the different formats produce differently-sized messages for these, as shown in Figure 7.7.

In both cases we note that adding compression markedly increases the processing time for Xebu, but adding the compression before encryption actually does not affect the XML case at all, or even improves it in the case of parsing. One reason is that because of compression there is much less XML data to encrypt, so the encryption and decryption processing are much faster. In addition, the compression ratio with XML is so much better than Xebu that

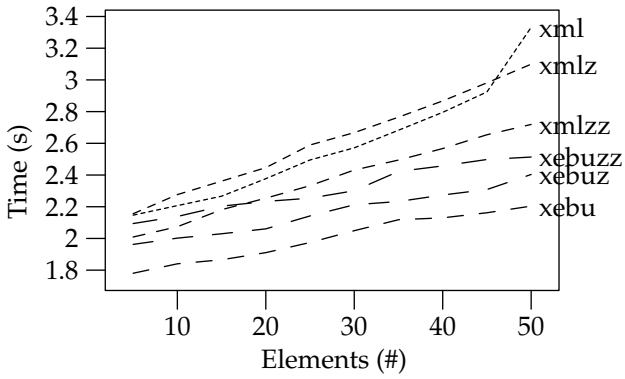


Figure 7.11: Security experiment parsing times

there is less data to move around during processing, which also affects total processing time.

In both serialization and parsing we see odd effects for XML, as the curve increases much more sharply than for the other formats. In serialization this can be explained by noting that the size of the XML document grows much more rapidly, which leads to the odd effect that its required serialization time seems also to grow more rapidly than that of the other formats.

On the parsing side there is the sharp jump at the 50-element point, which was already visible in Figure 7.8. Our investigation revealed that this is the point where the message size goes over 16384 bytes, indicating that the parser has a dynamic buffer of this size somewhere that needs to be enlarged during processing. We confirmed this by experimenting with message sizes up to 100 elements and noted a similar sharp jump when the message size increased beyond 32768 bytes.

Finally, we show the breakdown of the times for serialization and parsing in Figure 7.12 and Figure 7.13, respectively. These Figures show how much time each phase shown in Figure 7.6 took as a portion of the total time, with each number of elements represented as a row in the rectangle, and the number of elements growing downward. A large portion of each processing is also other processing, which consists of the non-security related serialization and parsing.

As in our earlier measurements [Kangasharju et al., 2006], we

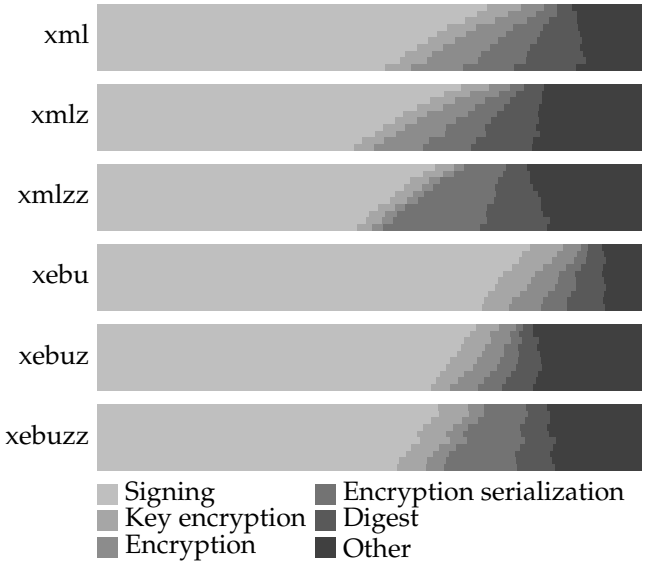


Figure 7.12: Security experiment serialization time breakdown

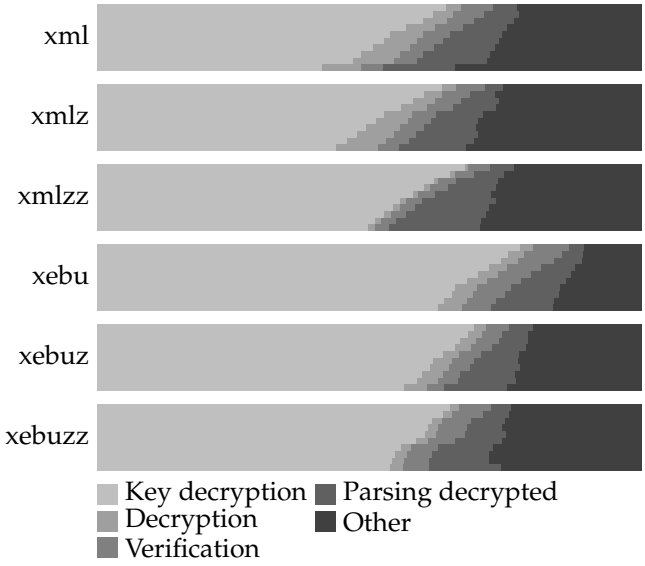


Figure 7.13: Security experiment parsing time breakdown

note that most of the time in all cases is spent in RSA secret-key operations, signature generation and key decryption. This is not likely to be caused by a poor implementation, as a simple benchmark showed that the relative performance of the different security operations here is approximately the same as with the high-performance OpenSSL toolkit[5].

What we do note from these Figures is that, with Xebu, a much smaller proportion of the time is spent on the matters related to actual serialization and parsing and more on the security processing. Furthermore, when contrasting to our proof-of-concept system in [Kangasharju et al., 2006], we note that the relative time spent on the serialization and parsing has decreased, undoubtedly due to the efficient implementation based on the XAS API.

7.6 Xebu Results

We made the Xebu experiment a completely separate one even though we also measured serialization and parsing times in the Messaging experiment. This was because the only way to measure elapsed time in JavaME is by wall-clock time. As the messaging system is multithreaded, wall-clock time measurement is disturbed by other threads getting CPU time, making the measurements imprecise. Also, even though we could simply take the measured minimum values, this would not help in measuring memory consumption, which we also wish to do.

Of the factors listed in Table 7.11, the Xebu experiment only used Device, Format, and Size. Since the messaging system is designed to store sent messages for potential retransmission, the Net, Number, and Style parameters should not have an effect on the message serialization and parsing times.

We show the sizes of the documents in Table 7.18. This also includes Xebu-S, the Xebu format with full schema optimizations included, i.e., pre-tokenization, data type encoding, and COA. We note immediately that Xebu-S performs extremely well on these documents, which is to be expected as the schema for the messages

[5] http://www.openssl.org/

Table 7.18: Document sizes in the Xebu experiment

	XML		Xebu		Xebu-S	
Size	plain	gzip	plain	gzip	plain	gzip
1	777	351	501	348	72	75
10	3136	643	1171	725	472	396

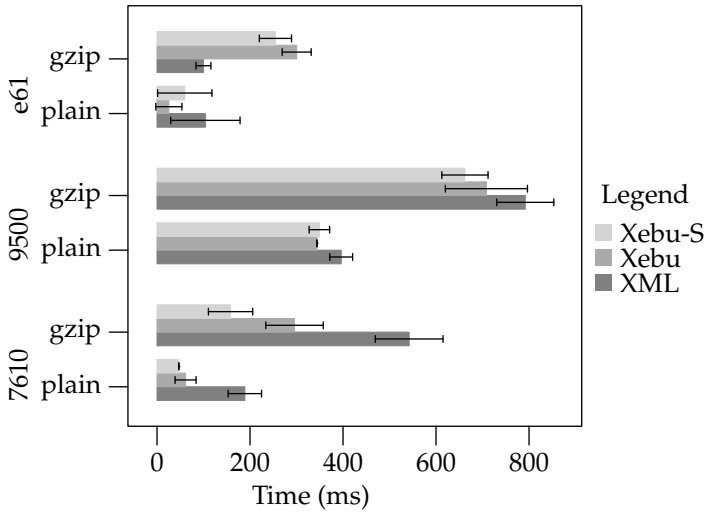


Figure 7.14: Xebu experiment serialization times

is basically linear, i.e., an element follows another with little choice as to the next element.

The sizes also show that Xebu does not compress as well with gzip as XML. While in the 1-element message compressed Xebu is still slightly smaller than compressed XML, this advantage is completely lost in the 10-element case. Xebu-S, on the other hand, already reduces the size so much that gzip has little that it can do.

From now on we will focus solely on the 10-element case, as the performance in the 1-element case is too dependent on initialization efficiency. The serialization times for all the devices, formats, and compression options for this message are shown in Figure 7.14. We have added the 7610 into these measurements to get some idea of what our weakest device is capable of with Xebu.

We note immediately that the performance of the 9500 is ex-

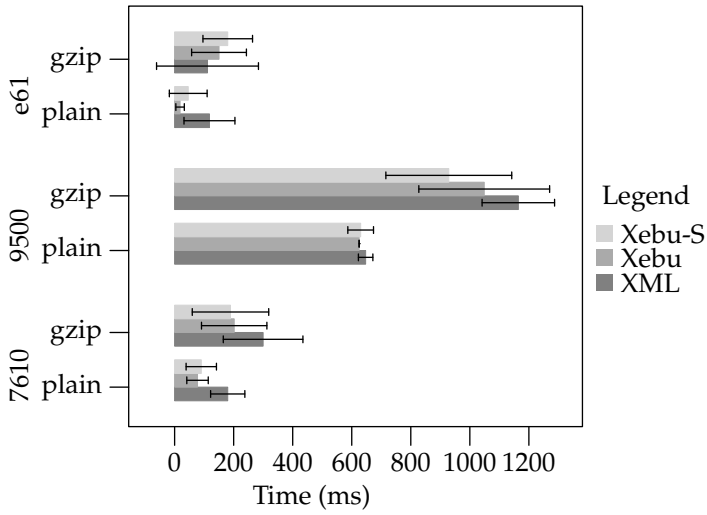


Figure 7.15: Xebu experiment parsing times

tremely poor in all cases. While we do not have detailed profiling data available, we suspect this is caused by a poor implementation of character encoding. In some of our earlier profiling, we have noticed that well over 50 % of the time spent in Xebu processing can be caused by character encoding and decoding.

The results on the E61 are somewhat of an oddity. Without gzip Xebu appears to perform better than XML, but when gzip is added, Xebu's performance worsens quite a bit. However, with XML the addition of gzip does nothing to the serialization time. We do know that the Java implementation on the E61 appears to be better than on the other phones, but it is hard to see what could cause this behavior.

The parsing times are shown in Figure 7.15. Here we see behavior familiar from prior measurements, with Xebu being clearly faster than XML when gzip is not applied, apart from the 9500 where there is no difference. Looking at the results for both 7610 and E61, we see that Xebu performs equally well on both. However, the results on the 9500 show that mobile applications still need testing and profiling on all target devices.

This was the only experiment in which we also measured dynamic memory consumption. Because of garbage collection, it is very difficult to measure this correctly in Java, especially on mobile

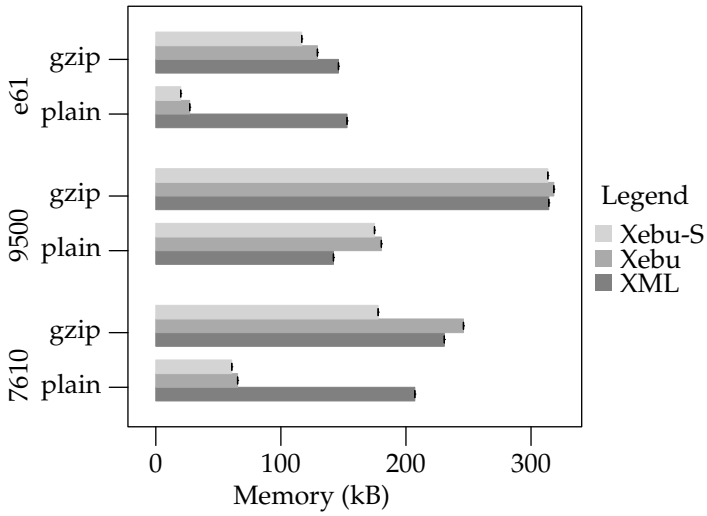


Figure 7.16: Xebu experiment serialization memory

phones where there is little functionality to affect the collection and no way to examine the state of the heap beyond the amount of free memory. We ran these experiments by making sure all garbage was collected (using standard techniques [Roubtsov, 2002]) immediately prior to the experiment, and report only the maximum value to ensure having caught a run that was not interrupted by garbage collection.

The memory consumption measurements are displayed in Figure 7.16 for serialization and in Figure 7.17 for parsing. The memory consumption on the 9500 is again not consistent with the others, and especially the parsing measurements would appear to correlate strongly with the timing measurements. Since the amount of memory allocated there is quite large, this could explain some of the poor timings.

The kXML serializer consumes a large amount of memory on the 7610 and E61 compared with Xebu. This again correlates with the timing measurements, indicating a potential problem in kXML. It would also appear that the code that is used by kXML and Xebu has seen much improvement in allocation behavior in the E61, as the memory consumption is much lower than on the other devices. Xebu still performs slightly better than kXML, and we can say that the memory consumption on the E61 is definitely acceptable.

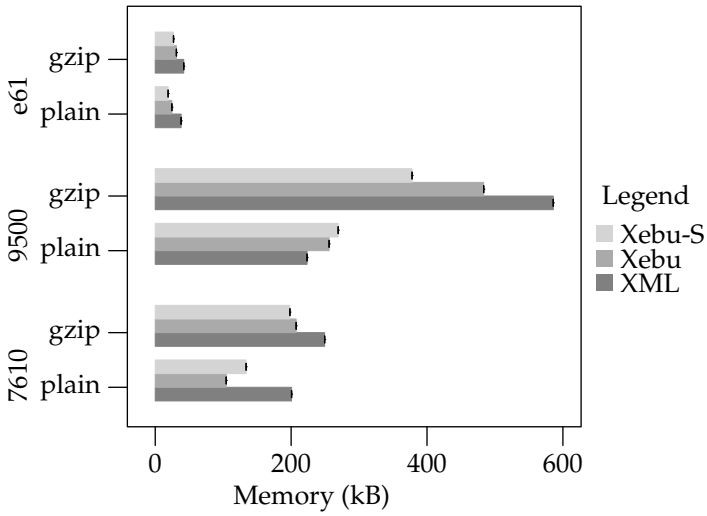


Figure 7.17: Xebu experiment parsing memory

7.7 AMME Results

Some of the features of the AMME protocol are partially visible in the results presented in section 7.4, but we also measured some characteristics of it individually. The important features that we considered to be so measurable are the overhead that the AMME Mobility headers cause for the messages and the accuracy of the RTTs measured by the RTT module described in subsection 4.5.4.

The AMME header is independent on the actual messages, but if several application messages are carried inside a single AMME message, the MESSAGE-BUNDLE header will add something to the header size. Therefore we measured three different header sizes: one for the synchronous case with no MESSAGE-BUNDLE header, and two for the asynchronous case, for the 2- and 20-message cases separately. Finally, two protocols are sufficient, since the Bluetooth and TCP mappings both use the stream headers.

The sizes for each of the six cases enumerated above are shown in Table 7.19. Sizes for individual application messages are as shown in Table 7.18. The Small asynchronous message contains 2 application messages, the Large message 16.

Comparison with message sizes, especially the smallest ones, seems to point to the AMME header causing quite a bit of over-

Table 7.19: AMME header sizes

Protocol	Synch (B)	Asynch (B) Small	Asynch (B) Large
Stream	58	64	91
HTTP	147	162	267

head. However, since the messaging measurement was intended to simulate one-off interaction, the sizes in Table 7.19 include the ACCEPT-TYPE header, the value of which contains the strings `application/x-ebu` and `text/xml`. This is nearly half of the smallest Stream header size, so if the interaction is long-running, the effect of the AMME header will be much smaller.

We performed the RTT measurement over a GPRS link, using HTTP as the protocol, since it has a natural application-level round trip. We simply included a measurement of the HTTP request-response interaction time for the data messages. The average time we measured with HTTP was 1.86 seconds, and the AMME RTT module stabilized at 1.84 seconds. Since the AMME measurement, as illustrated in Figure 4.8, discounts local processing, it would be expected that the AMME measurement would be slightly lower.

This measurement was performed in the synchronous case with no other activity going on while an HTTP request is outstanding. We also experimented in the asynchronous case, and got much less stable results. The RTT measurement fluctuated widely, and could be as high as 6 or 7 seconds. Since the AMME measurement is performed at the application level, other network activity and local processing will affect the time it takes for data to transfer between the RTT module and the network, thus extending the time and making it more variable.

In a sense, the RTT module is working correctly in this case. Namely, its values are intended for use by applications to determine the current application-level RTT, and since the module is on the path of message processing, it is measuring what it is supposed to. However, high network latency is a different problem from high application load, so the RTT module alone is not sufficient to provide accurate information on the network conditions.

7.8 Practical Considerations

Actually running the experiments was not completely straightforward. While the implementation can be correct, written according to the interface specifications, the underlying platform can have bugs. Our experience has been that such bugs are frequently encountered with a system as complicated as this one. We list here the symptoms of the problems we encountered, our conclusions as to their proximate causes, and the ways we solved them either in the system implementation or just for the experiments.

The latency results in Figure 7.3 and the Messaging results in Figure 7.5 showed exceptionally poor performance for Bluetooth. It turned out that the stream API of Bluetooth nearly always returned an end-of-file indication when trying to read a second block of data from the stream, even if more data was available. This was possible to work around by waiting a while after a failed read and trying again, which succeeded. However, the wait time had to be set close to 1 s, making the latency be dominated by the wait and not by the protocol.

Another reason for not being able to provide Bluetooth measurements for all cases was poor implementation quality of the protocol on the 9500. Whenever too much data was coming in or going out, as in the asynchronous cases or even the synchronous ones with Xebu, the JVM running the program on the 9500 crashed, and we could not find any way to work around this. Finally, there were occasional problems with getting the E61 to connect using Bluetooth. While the discovery phase seemed to find the other device, it never started communicating. Oddly, this happened with only certain configurations and was repeatable in some cases but not in others.

In general, we found the E61 to be a reliable device with good performance, perhaps indicating that the phones are truly making technological progress, and may be capable of hosting complex pervasive applications very soon in the future. We had one interesting problem that did not affect the measurements but would be an annoyance in real use. Namely, whenever we used the ad-hoc WLAN for measurements, the phone rebooted itself after exiting the program and shutting down the WLAN interface. We do not believe such behavior would be acceptable to users.

Communicating over the mobile phone networks also requires paying attention to what the operator is doing. All mobile phone networks function through an operator's Network Address Translation (NAT) system: the phone itself gets an IP address from some private non-routable space and the operator's system converts between this and a routable address belonging to the operator. This means that it is not possible to host Internet server applications on the phone without doing something like what our protocol's HTTP Transfer mapping does with the token messages.

However, the token messages are not the complete solution. Leaving an HTTP request hanging at the server will cause connection termination by the operator at some point, which is why the HTTP server side in our system will time out when waiting for a response to a token message. When this timeout fires, the server will simply send an empty message as a response, causing the client to recreate the token and open a new connection. We have observed timeouts as low as a few minutes, but have recently heard that many current operators have much longer timeouts.

7.9 Summary

The first thing that can be summarized from the measurements is that there are no easy solutions. Different approaches work best in different situations, and measurements are dependent on a variety of things, some of which may not be evident at first glance, and others not even after careful study. Thus, the system needs to support a variety of networks and communication patterns to achieve the best results.

The benchmarking results provide a view, not only of current performance, but also of potential future performance. Of particular interest is the fact that the E61 gets much better performance over EDGE than the 9500, and also that there is not a very large difference in performance between EDGE and UMTS on the E61.

On a network with low latency, such as WLAN, it would appear that decreasing message size provides useful benefits, and this is also wise advice in general, as larger messages take more energy to transmit. However, Figure 7.5 shows that applying gzip in the asynchronous case actually worsens total performance. As

the CPU sees much more use in this case than in the synchronous one, additional compression quite naturally increases processing time. So decreasing size at the cost of additional CPU time may not be wise for applications that also do other processing during communication, which speaks in favor of a binary format.

In security processing, we note that Xebu, with or without compression, achieves approximately the same results as XML with full compression. Figure 7.12 and Figure 7.13 show that most of the time in this case is consumed by RSA secret-key operations. Other asymmetric algorithms would show similar performance, but one benefit of RSA, as commonly used, is that its public-key operations are very efficient.

Since RSA has a performance asymmetry between its secret-key and public-key operations, it suffices to eliminate secret-key operations, even at the cost of adding public-key operations. Our basic recommendation is to have the mobile client pre-generate keys securely when there is sufficient power available. It then sends these keys when appropriate, encrypted for the server. The server indicates that it used the sent key, letting the client avoid a secret-key operation by adding a public key operation and some communication. We provide more detailed discussion of this proposal in [Kangasharju et al., 2007b].

Xebu appears to be an acceptable format, especially when a precise schema is available. We do note that adding generic compression to Xebu seems to result in larger documents than XML compressed generically. This is most likely caused by the explicit presence of the token values, which breaks repetitive sequences. This could be mitigated by not requiring the token values when there are still unmapped ones available, or even eliminated by precisely specifying the token assignment algorithm.

CHAPTER 8

EXI Format Comparison and Analysis

> It honestly doesn't occur to them.

The Efficient XML Interchange (EXI) WG at the W3C performed a large amount of measurements on a number of proposed binary XML formats. The author of this dissertation is a current participant in this group representing the University of Helsinki, and was responsible for analyzing and summarizing the measurement results. This analysis, in conjunction with the actual results, formed the basis for the WG's decision to select Efficient XML as the baseline for the EXI format.

8.1 Preliminary Considerations

In the initial phase, the EXI WG needed to determine what to measure. The initial work on this was performed by the XBC WG in the properties document [W3C, 2005e] that lists a large collection of characteristics a data format might have, and in the characterization document [W3C, 2005c] that determined which properties are the most important.

The measurement process of the EXI WG used real XML data currently in use in deployed systems. However, in scenarios where

a binary format is desired, not all features of XML are needed, and a format may achieve additional size reduction if it is aware of the precise needs of a use case. This consideration was used in classifying both formats and test data.

8.1.1 Measurement of Properties

According to the XBC properties document [W3C, 2005e], a *property* is

> a unique characteristic of an XML format which affects the format's utility for some collection of use cases.

A companion document [W3C, 2005d] defines for each property how to determine whether a format possesses that property and to what extent. The properties are divided into three classes: *format* properties are based only on the format specification, *algorithmic* properties are characteristics of an implementation, and *additional considerations* are desirable properties that are not feasible to actually measure.

The two main properties that have driven the development of alternate formats are *Compactness* and *Processing Efficiency*, since these are required by a large number of use cases [W3C, 2005c] and not usually considered to be properties of XML. The former is simply document size and the latter is the speed at which it is possible to parse or serialize format instances. For mobile devices, other important properties are *Small Footprint* and *Space Efficiency* that measure the size of a processor implementation and the amount of dynamic memory required during processing, respectively.

The Compactness measurement defined by the XBC WG [W3C, 2005d] considers tokenization to be the basic method of binary formats, applicable to any XML data and any use case. In addition, two advanced techniques are of more limited applicability. *Document analysis* refers to more global analysis of the document, using well-known compression techniques, and may not be suitable when Processing Efficiency is crucial. *Schema-based techniques*, on the other hand, are only usable when a suitable schema for the document is available.

These two additional options divide the Compactness measurement into four different *application classes* [W3C, 2007b]:

Neither Only tokenization is used.

Document Document analysis is applicable, schema-based techniques are not.

Schema Schema-based techniques are applicable, document analysis is not.

Both Both document analysis and schema-based techniques are applicable.

While the application classes were conceived for the Compactness property, they will clearly affect the measurement of other properties as well. For instance, the compression methods used in document analysis are typically very time-consuming in comparison to the other processing, so Processing Efficiency also has to be considered per application class.

The XBC WG said very little about actually measuring Processing Efficiency. The measurement itself lists some scenarios, from basic XML parsing and serialization to constructing an application data model or executing queries. However, the intent behind measuring is to pit implementations against each other and measure the time spent.

8.1.2 Preservation of Information

The XML Information Set includes all the information in an XML document. However, not all use cases require preserving all of this information. For instance, many applications that use XML only as a format for structured data might not care about anything except elements and attributes. Another example is provided by SOAP, which actually forbids DTDs and processing instructions.

Because of these considerations, some use cases might get more use out of a binary format that is able to improve its Compactness by knowing which information will not appear in the processed documents. To take this possibility into account, the WG defined a *fidelity scale* for formats and test documents, indicating which information a format can preserve or a test document requires.

The fidelity scale goes from -1 to 4, with -1 indicating that only a subset of the XPath data model [W3C, 2007h] is preserved,

and 4 indicating that the exact byte sequence of the original XML document is preserved. The most relevant levels on this scale for formats are 1, preservation of the full XML Information Set, and 2, which also includes preservation of all declarations in a DTD.

The fidelity scale is complemented by *fidelity options*, which indicate more precisely the information to preserve. The WG defined six fidelity options,

- preservation of white space,
- preservation of comments,
- preservation of processing instructions,
- preservation of namespace prefixes,
- preservation of lexical values, and
- preservation of the document type declaration and internal subset,

which mostly indicate specific information in an XML document that needs to be preserved.

Both white space and lexical value preservation are only applicable when a schema is used. A schema may indicate that some white space is not significant, and the preservation option defines that even this insignificant white space needs to be preserved. Similarly, a schema will assign types to some values in the document. Lexical value preservation means that the specific representation of these values must be preserved, instead of merely preserving the actual typed value.

The fidelity scale and options are closely tied to the property *Round-trip Support*, which requires the format to preserve all relevant information in an XML document. Since the relevant information varies by use case, each document is assigned values for the fidelity options to indicate precisely the information that must be preserved. A candidate may therefore take advantage of the fidelity options by dropping information from the XML document that is not used by the application at all.

8.2 Comparison Framework

Of the properties identified as requirements by the XBC WG, some are measurable by simply looking at the specification, and some can even be included in any format without altering the specification materially. However, some properties must be measured from a real implementation. Such properties include the most crucial ones, Compactness, Processing Efficiency, and Space Efficiency.

Compactness measurement is simple when an implementation exists. It is enough to simply run the set of test documents through the implementation's serializer and measure the sizes of the resulting candidate documents. However, it was determined that some form of verification was needed for the results, to make sure that all candidates preserved all of the relevant information in the test documents. This verification took the form of measuring the Round-trip Support property.

8.2.1 XML Differencing with Faxma

To measure Round-trip Support, the Fuego Core project provided their XML differencing tool, Faxma [Lindholm et al., 2006], to the WG. This tool can take either XML document sources or parsed event streams as arguments, and it can be made to print a report on the differences, similarly to the Unix `diff` program. We integrated the differencing process as an optional component in the Compactness measurement.

Getting Faxma to work in the Neither application class was essentially a simple task. The candidates are required to provide SAX parsers for their formats, so comparison is a simple matter of translating these SAX event streams into XAS fragments and calling the comparison method. One thing that this required was the implementation of a SAX filter that understands the fidelity options and removes the content that does not need to be preserved.

The Schema case was somewhat more complex, as the fidelity option for preserving lexical values requires type information to be available. This requires a validating parse to construct the PSVI. For ease of implementation, this was implemented by serializing the provided SAX event streams into XML and then using a DOM-based parser on the resulting documents. The constructed DOM

tree then has type annotations for the nodes containing typed data, which can be used to construct XAS typed items.

While the eventual code that got used in the EXI measurement framework is in places specific to the test data (for instance, it has special-case code for one list type instead of recognizing any list type as defined by XML Schema), the full Faxma tool is mostly generic and applicable to any XML documents and any schemas. We intend to eliminate the special-casing, replacing it with either generic code or user-specified data, and release Faxma as a general-purpose schema-aware XML differencing tool. Its feature list, which we understand is unequalled among such tools, includes

- capability to process almost any XML (DTDs are currently poorly supported),
- ability to disregard information not deemed interesting,
- ability to compare typed data in either their lexical space or value space, and
- ability to process invalid documents while still comparing valid typed data in the value space.

8.2.2 Measuring Processing Efficiency

Measurement of Processing Efficiency and Space Efficiency is more difficult. While in Compactness and Round-trip Support measurements it can be expected that the implementation is deterministic, so that each run produces the same result, measuring time and memory usage will not produce the exact same results every time. Further issues are caused by the running time depending on what has been running previously, due to caching and the Java execution model, the latter of which we explained in subsection 7.2.2.

In addition to the warmup loop required for stable timings, the measurement itself must be run several times to detect incidental variation that always exists in timing measurements, and sometimes also in memory usage measurements. However, this looping means that the data on which the measurements are run is used constantly, making it a prime candidate for caching. If the data comes from a disk, the disk cache will store it. If the data is in

memory, the processor cache may occasionally be large enough to store it. Having the test data cached will not give realistic performance numbers, since a significant part of any data processing is actually moving the data into and out of the processor.

However, it is not feasible to simply disable the caches to avoid this phenomenon. An efficient implementation of a format processor will take advantage of the caching that is present in the platform, and this can make a large difference. Therefore, what is needed is a way to ensure that the actual data used during the measurement run has not been cached without disabling the caches during the run.

For in-memory processing there are some options. One is to create several copies of the test data in various locations in the memory, and to use a different one for each measurement run [Kostoulas et al., 2006]. The number of copies is set high enough so that when it is necessary to reuse a copy, it is certain that the data cannot be in the processor cache. Another option, available with Java, would be to create a new buffered stream class for each run, as this will ensure that there is a new location in memory for the actual test data on each run.

It can be argued, though, that an in-memory measurement is not a realistic reflection of the capabilities of a format. In real applications, data usually comes from an external source such as a disk or a network, and is similarly output somewhere external. Therefore to get a proper measurement of Processing Efficiency, it is necessary to use a real data source and target, potentially several different ones.

The solution that was implemented is based on networking. Namely, all data is read from or written to a TCP connection. For serialization experiments, the server at the other end simply receives the data and discards it. For parsing experiments, the server is initially provided with the document to use, and it will send a continuous stream consisting of copies of this document. The basic measurement is performed using the local loopback interface as the network, and further measurements can be performed by simply switching to a real network.

8.3 Measurement Analysis

The measurements provided a vast assortment of data to analyze. As XML is a widely applicable format, there are few commonalities in the test data across the full suite. Accordingly, it is not realistic to expect that it would be possible to determine a single figure of merit for comparison of the candidates. Rather, the analysis process must first identify common ground, provide summary figures for each identified group, and then compare candidates in each group, potentially drawing more general conclusions if such are warranted.

We do not present the complete analysis of the measurements here, but have chosen to focus on a subset of the test documents relevant to mobile devices, fitting with our main topic. Also, we only analyze a subset of the formats, as including more formats would not provide much additional insight. The full analysis can be found in [W3C, 2007b].

8.3.1 Test Data Classification

The data that was used in the analyzed measurements consists of 88 documents, representing 21 different XML vocabularies, called *test groups*. Coverage of the XBC use cases [W3C, 2005f] is nearly complete, with only 4 use cases out of 18 lacking representative documents, and 3 of them being application-specific (X3D Graphics Model Compression, Serialization and Transmission; XMPP Instant Messaging Compression; SyncML for Data Synchronization).

The use cases themselves provide one natural way of classifying the test documents. Each test group is accompanied by a list of use cases for which it is suitable [W3C, 2007b], so tabulating these use cases and grouping related test groups together gives a set of *use groups* shown in Table 8.1, a higher-level division than the test groups, with still sufficient commonality between included documents that it should be possible to draw general conclusions across complete use groups.

Another division comes from considering various complexity metrics of XML documents. Qureshi and Samadzadeh [2005] consider metrics such as the size of the serialized document in bytes, the total number of elements in the document, and the height of

Table 8.1: Use groups identified in the EXI test suite

Name	Representative use case
Broadcast	Metadata in Broadcast Systems
Document	Electronic Documents
Finance	Intra/Inter Business Communication
Military	Military Information Interoperability
Scientific	Supercomputing and Grid Processing
Sensor	Sensor Processing and Communication
Storage	XML Documents in Persistent Store
Web services	Web Services for Small Devices

Table 8.2: Content density clusters in the EXI test suite

Name	CD	Size
High	over 33 %	any
Low-Large	under 33 %	over 100 kB
Low-Small	under 33 %	over 1 kB and under 100 kB
Low-Tiny	under 33 %	under 1 kB

the document when seen as a tree. They also consider an algorithm that assigns a complexity value to a DTD and based on that, and document-specific metrics, develop a complexity measure for XML documents. Using a DTD extraction tool [Garofalakis et al., 2000] it is possible to extend this measure to arbitrary XML.

For the EXI analysis, two complexity metrics were chosen. The more obvious one is the size in bytes of each document. The other metric is called *content density (CD)* and is measured by dividing the amount of content by the total document size. Here content is defined as the values of attributes and text data in the document. This gives rise to *CD clusters*, shown in Table 8.2. The thresholds were chosen as somewhat natural ones that would split the test data into clusters of approximately equal size.

The CD clusters have an orthogonal purpose to use groups. Whereas use groups are intended to measure the performance of an implementation in a specific use case, the purpose of the CD clusters is to measure the performance of format's structure and data encoding capabilities. The documents in the High cluster

Table 8.3: Analyzed test groups

Group	Size	Description
ASMTF	4	Military information messages
CBMS	4	Broadcast metadata
Google	3	SOAP Web service messages
JTLM	2	Military SOAP messages
Location	2	Individual coordinates
SVGTiny	2	SVG documents
WSDL	2	WSDL documents

have relatively little actual XML structure, so differences there are most likely due to different performance in encoding text data. On the other hand, the documents in the Low clusters exercise mostly the structure encoding implementations. Division by size makes it possible to also determine the amount of overhead required of the implementation: in the Low-Tiny cluster most of the processing time is likely taken by initialization and not by document processing as such.

For the purposes of the analysis presented here, we took the test groups relevant to mobile devices and selected some of the documents from those test groups. The main use groups we examined were Military, Sensor, and Web services, but the Convergence of Broadcast and Mobile Services (CBMS)[1] and Scalable Vector Graphics (SVG) [W3C, 2003a] documents were also chosen, since multimedia is expected to be more prominent on mobile devices in the future. The documents range in size from 100 bytes to 100 kilobytes and have a large range of different CD values.

8.3.2 Analysis Methodology

The total amount of measurement data gathered by the framework is huge, as each individual document is measured for all candidates and application classes in both Compactness and Processing Efficiency. Even more, Processing Efficiency was measured with a

[1] http://www.dvb.org/groups_modules/technical_module/tmcbms/index.xml

number of underlying data sources and sinks. This all combines to an immense amount of data that needs to be summarized.

To further complicate the summarization, the test documents come from a variety of sources, with the intention that the test suite capture as wide an area of XML usage as feasible. Therefore it is certain that aggregating all the measurements into a single figure of merit cannot produce useful results, but rather a number of different aggregate numbers are needed. In particular, this will be a concern in real use of EXI where an application designer typically has only a certain form of XML document in mind and wishes to know how EXI performs in that case.

Our analysis proceeded along two different paths. The splitting into use groups and CD clusters provided usefully small and homogeneous collections of documents over which aggregate conclusions can potentially be drawn, so they were taken as the basic division for analysis. In addition, the whole test suite was collected into one, but keeping in mind that analysis on that can only provide imagery, not useful conclusions. The mobile group described above was locally added by the author, as the scripts written for gathering and aggregating the data were designed to support exactly this kind of use.

The first analysis was performed through graphical inspection. The results of each use group and CD cluster were plotted for each different application class. Such graphs give a useful overview of the results for each particular groups of documents. In particular, anomalous results are clearly visible so that they can be investigated and eliminated. In addition, the graph can be used to form impressions of the relative performances of the candidates, both in relation to each other and to the baseline.

The graphical inspection was further supplemented with the detailed measurements. Impressions acquired from the graphs were confirmed by evaluating the results from individual test documents. The individual results were also used to determine ranges of performance for each of the candidates. Finally, wide differences were inspected and explained by considering the content of the test documents.

The second analysis used statistical methods to summarize the performance into a single ratio over the baseline. As noted above, extracting a single figure of merit over the complete set of test data

is not feasible, so this analysis was also split by application class and use group or CD cluster.

The basis of this summarizing was the assumption that a use group or a CD cluster is sufficiently homogeneous that the performance ratio of a candidate compared to XML is approximately constant across the group. Thus, by plotting the results into a (baseline, candidate) coordinate system the plot should resemble a straight line, from which the ratio can be acquired through linear regression analysis.

As in all statistics, the values computed from this cannot be said to be exact. The regression analysis produces some deviation, which is taken into account by reporting 95 % confidence intervals for the ratios. This is computed in the standard manner, i.e., the interval is $[\mu - t(0.975) \times \sigma, \mu - t(0.975) \times \sigma]$ where μ is the slope of the regression line, σ is its standard deviation, and t is the Student distribution with the appropriate degrees of freedom.

The baseline for comparison in these analyses also merited careful consideration. In the graphs it was decided to always show the basic XML candidate. This is supplemented by also having a compressed form of XML or a faster processor as each test run permits. For the regression analysis it was decided to select the best XML performer as the baseline. In the case of Compactness, this is also required, since document analysis usually improves its compression ratio with increasing document size, thus making the relationship between compressed and uncompressed sizes non-linear.

The analysis, graphs, and tables presented here form only a subset of the complete analysis in [W3C, 2007b]. We limit the investigated candidates to Xebu, due to it being included in this dissertation, Fast Infoset (denoted FI), due to it being the best-known modern format, and Efficient XML (denoted EFX), due to it having been selected as the basis for the future EXI format. Furthermore, graphs will be shown only for the Neither application class, though data from other application classes will be discussed.

We noted above that the properties Space Efficiency and Small Footprint are important for mobile devices. Neither of these properties was measured by the EXI WG, so we cannot include any measurements of Space Efficiency here. However, most of the format implementations were made available, so it is possible to measure their footprints, which are shown in Table 8.4. The difference

Table 8.4: The footprints of the candidate EXI implementations

Format	Size (kB)
XML	1629
Xals	291
Xebu	184
FI	1222
EFX	–

between Xebu's size here and in Table 7.1 is because this measurement includes Xebu, kXML, and large parts of XAS, and is not obfuscated in any manner.

These footprints were measured by adding together the sizes of the library files needed to run the candidate. The XML figure is from the standalone Xerces distribution. Xals is a fast XML parser used for parsing efficiency measurements, so its size includes only a parser. As the Efficient XML implementation was only made available to the W3C personnel, its size could not be included. One final point to note is that the Xebu implementation is written for the Java MIDP libraries. Since the Java library size is not included in the measurements and the other formats are written for the full Java Standard Edition (JavaSE) libraries, this may penalize Xebu somewhat, though it is not feasible to quantify by how much.

8.3.3 Compactness Analysis

We begin by showing a graph of Compactness in the Neither class of the three candidates in Figure 8.1. We refrain from showing graphically the results of the other application classes as a complete set of graphs would take up too much space without providing much additional information. This graph plots the compression ratio of the candidates against XML, i.e., the measurement is XML document size divided by format document size, plotted against XML document size in bytes. Due to the variety of XML document sizes (the smallest is just slightly over 100 bytes and the largest a bit under 100 kilobytes) the X axis in the graph uses a logarithmic scale. This should not cause confusion as it is the individual documents that are interesting, not just their sizes.

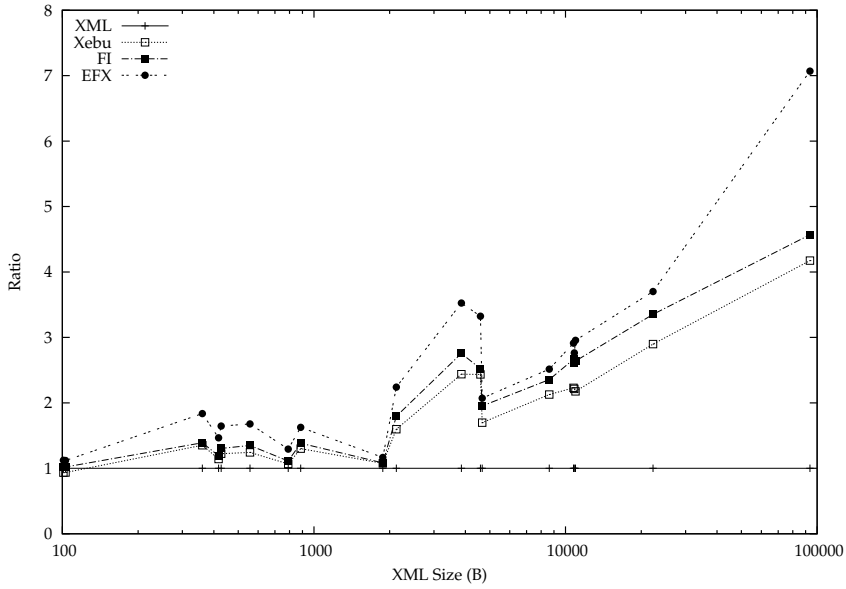


Figure 8.1: Compactness results in the Neither class

From the graph we can distinguish a clear performance difference in favor of Efficient XML compared to Fast Infoset and Xebu, with Fast Infoset being slightly better than Xebu and both clearly improving over XML. Furthermore, examining the individual results, the documents where Efficient XML has the largest advantage belong to the ASMTF and JTLM groups, i.e., military messages for which Efficient XML has been explicitly designed.

The performance ratios acquired through regression analysis are shown in Table 8.5. As the technique summarizes the whole run into a single interval, we show the results for all application classes instead of just Neither as in Figure 8.1. Here the baseline for the Document and Both classes is XML compressed with gzip. Since neither Xebu nor Fast Infoset includes a compression algorithm, the Document and Both results for these are the results of compressing the Neither and Schema documents with gzip.

First of all, it must be noted that the results for Xebu and Fast Infoset change very little when a schema is available. This is because in the Schema class both use only the technique we call pretokenization. Fast Infoset does not support any other schema-

Table 8.5: Compactness ratios

Format	Neither	Document	Schema	Both
Xebu	[3.71, 4.44]	[0.86, 0.97]	[3.80, 4.58]	[0.88, 1.02]
FI	[4.18, 4.86]	[1.01, 1.16]	[4.50, 5.06]	[1.29, 1.61]
EFX	[5.38, 7.47]	[1.10, 1.48]	[8.40, 12.67]	[1.43, 2.44]

based technique, and Xebu's type encoding and COA could not be used, since the test suite included only XML Schema definitions, which were not supported by the Xebu implementation of the time. Automatic conversion of XML Schema to RELAX NG was attempted, but the available tools were not sufficiently robust to produce usable results.

From these results we can conclude that Efficient XML is clearly superior to Xebu and Fast Infoset. The latter two are approximately equal, with Fast Infoset appearing to have a slight advantage. It would also seem that Fast Infoset compresses better with gzip than Xebu. Both of these are likely to result from the explicit token assignments in Xebu documents, as they both increase document size and break larger repetitive sequences, hindering gzip.

8.3.4 Processing Efficiency Analysis

Processing Efficiency includes two components, serialization and parsing. In fact, the XBC property [W3C, 2005e] includes even more processing like data binding and query evaluation as well. However, the EXI measurement framework only includes serialization and parsing through the SAX API. There are also facilities for running each candidate with its own native API, but this functionality was not considered to be stable enough to include the results in the analysis.

Unlike Compactness where the measurement is largely independent of any actual implementation (this usually depends on possible optional features, which the XBC property *Single Conformance Class* was written to discourage), Processing Efficiency is not feasible to analyze properly without an implementation. While complexity analysis based on format specification may give rough directions, format processors are typically linear in input size, and

any constant factors in software as complex as an XML processor are not discernible simply from a specification.

The dependence of this property on existing implementations also means that the XML processors used for comparison need to be efficient. As an example, Head et al. [2006] note that XML parsing performance can easily vary by a factor of 2 or 3 depending on the implementation. Because of such concerns, the EXI WG issued a public call for efficient XML processors. The result of this was that for parsing, the WG could compare against Fujitsu's efficient Xals parser, but for serialization only Java's default implementation based on Apache Xerces was used.

The network-based measurement system described above in subsection 8.2.2 was mostly used over the loopback interface. Our experience [Kangasharju and Tarkoma, 2007] is that the loopback interface on Linux is sufficiently fast not to be the bottleneck of processing. However, recently measurements were also made over real networks, and in addition to the loopback interface we also present measurement over a 11 Mbps WLAN link as that is interesting from the point of view of mobile messaging.

Serialization Efficiency

When considering alternate XML serialization formats, serialization performance is usually accorded much less importance than parsing performance. Typically, XML serialization is seen as an essentially trivial matter, consisting of a tree traversal with `print` statements, so it is not expected that there would be large gains in serialization. This view is also a likely reason why there is little standardization in XML serialization APIs.

As with the Compactness measurements, we show graphs only for the Neither class, and show the performance ratios for all application classes. As mentioned above, we examine both the loopback measurements, to determine the CPU efficiency of the implementations, and the WLAN measurements, to determine the performance in an environment closer to that of mobile devices.

The serialization efficiency graph for the Neither class over the loopback interface, shown in Figure 8.2, mostly supports the view of serialization being hard to improve. This is again plotted as a ratio compared to XML performance, so the X axis shows the num-

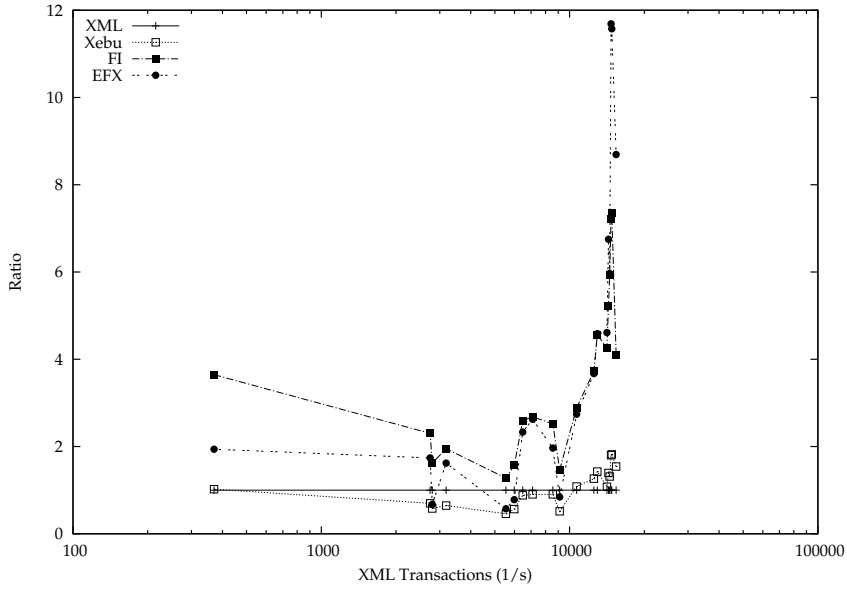


Figure 8.2: Serialization efficiency results in the Neither class over loopback

ber of serializations that the baseline was able to perform in a fixed time, which was the measurement made in the EXI framework. This means that the documents on the X axis are roughly in the opposite order compared to the graph in Figure 8.1 (only roughly because the number of transactions is not perfectly correlated with document size).

While Fast Infoset clearly performs better than XML, the performance of Efficient XML is not consistently better, and Xebu is better than XML only on smaller documents where it is more likely that the initialization costs dominate the processing time. Since the baseline is Xerces, which is known to be a heavyweight parser, it is to be expected that its performance for small documents would not be very good.

The serialization efficiency ratios over the loopback interface are shown in Table 8.6 for Xebu, Fast Infoset, and Efficient XML. These ratios confirm the view from Figure 8.2 that while it is possible to improve serialization efficiency, the potential improvement is small. Comparing these ratios to those in Table 8.5 we note that

Table 8.6: Serialization efficiency ratios over loopback

Format	Neither	Document	Schema	Both
Xebu	[0.91, 1.07]	[0.48, 0.55]	[0.80, 0.91]	[0.48, 0.55]
FI	[3.03, 3.88]	[2.46, 2.93]	[1.82, 1.98]	[2.36, 2.70]
EFX	[1.45, 2.03]	[1.65, 1.90]	[2.36, 2.81]	[2.57, 2.95]

Table 8.7: Serialization efficiency ratios over 802.11b

Format	Neither	Document	Schema	Both
Xebu	[3.58, 4.51]	[0.95, 1.03]	[3.62, 4.49]	[0.92, 1.11]
FI	[4.16, 4.93]	[1.01, 1.13]	[4.05, 4.60]	[1.19, 1.57]
EFX	[4.90, 6.80]	[1.10, 1.47]	[7.00, 10.03]	[1.62, 2.61]

they are quite different, indicating that it truly is the CPU that is the bottleneck in this measurement.

For the WLAN network, we get the serialization efficiency ratios shown in Table 8.7. Comparing these numbers to those in Table 8.5 for Compactness, we note that there is no significant difference between the results, which is also why we do not show these results as a graph, since the graph is essentially the mirror image of Figure 8.1. This is an expected result, as the implementations are fast enough to process data faster than the 11 Mbps network is able to transmit it. In a way, this result demonstrates that message size is a much more important concern than the CPU efficiency of a format, though it must be kept in mind that in a real application there would also be other processing consuming CPU time.

Parsing Efficiency

We show the results for parsing in the same manner as for serialization. Namely, graphs are shown for the Neither class while ratios are computed for all application classes. We investigate again the same networks, the loopback interface and WLAN.

The graph for parsing efficiency in the Neither class over the loopback interface is shown in Figure 8.3. As with the serialization graph, this shows ratios of processor performance measured in transactions per second compared to the baseline that is the de-

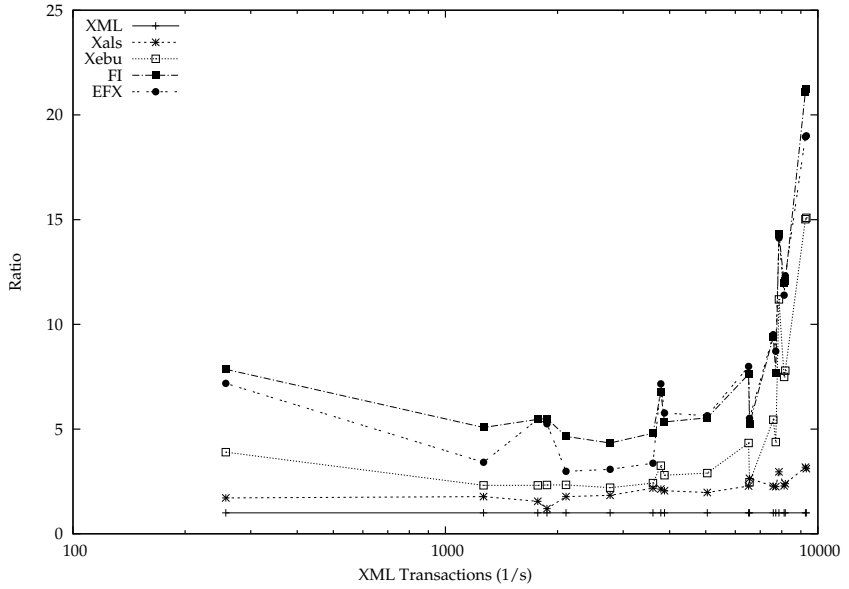


Figure 8.3: Parsing efficiency results in the Neither class over loopback

fault Java implementation. The faster Xals parser is also included in the graph and should be considered the main comparison point.

The graph shows a consistent performance for all candidates and Xals for all the larger documents that achieve a lower number of transactions. Again, there is a clear upward improvement in performance when looking at the smallest documents. Since also Xals improves its performance, we can consider this further evidence that the poor performance of the baseline here indicates a large startup cost. Nevertheless, the improvement of the candidates for these smaller documents is much larger than for Xals.

The efficiency ratios for this measurement, shown in Table 8.8, are computed against Xals, as that is the baseline for comparison in parsing. These ratios are clearly better than the ones for serialization in Table 8.6, showing that there is indeed more to improve on the parsing side by moving to an alternate format.

There are few surprises in parsing efficiency over the WLAN interface in light of the comparable results for serialization above. The parsing efficiency ratios in Table 8.9 do not differ significantly from the ratios in Table 8.5 or Table 8.7, confirming that the net-

Table 8.8: Parsing efficiency ratios over loopback

Format	Neither	Document	Schema	Both
Xebu	[1.96, 2.39]	[1.89, 2.35]	[1.73, 2.12]	[1.99, 2.51]
FI	[4.10, 4.85]	[3.12, 3.97]	[5.00, 5.71]	[3.66, 4.51]
EFX	[3.20, 4.40]	[2.78, 3.47]	[3.80, 4.32]	[3.03, 3.30]

Table 8.9: Parsing efficiency ratios over 802.11b

Format	Neither	Document	Schema	Both
Xebu	[3.55, 4.46]	[0.94, 1.01]	[3.68, 4.57]	[0.88, 1.04]
FI	[4.17, 4.96]	[1.00, 1.10]	[4.21, 4.76]	[1.18, 1.54]
EFX	[5.04, 7.20]	[1.08, 1.41]	[7.09, 10.11]	[1.54, 2.55]

work is the bottleneck. Further confirmation is available by examining the results of Xals, which we do not show here. Namely, the performance ratio of Xals compared to the baseline remains at a constant 1 for all documents, showing that the CPU efficiency of the parser is of little importance.

8.4 Summary

The results shown above demonstrate that it is possible to improve both Compactness and Processing Efficiency from XML without sacrificing XML compatibility. While this does not seem a profound realization, there is an often-expressed claim that this would be impossible. Such claims usually stem from a misunderstanding that an improved XML-compatible format would have to first produce the XML and then compress it, which would indeed lead to worse Processing Efficiency. This is why the property *Directly Readable and Writable* [W3C, 2005e], i.e., directly accessible through an XML API, is required for an EXI candidate.

Comparing to the results of section 7.6, Xebu would appear to perform worse here. At least one explanation for this is that the Xebu implementation used here was the same one that was used on mobile devices. As seen in section 7.6, even changing the phone could change the results noticeably, so it is understandable that the

results change when running on a regular computer. We do not believe these results to indicate that an efficient implementation of Xebu for regular computers is impossible.

One important measurement that is unfortunately missing here is measurement through a candidate's native API. For instance, in a scientific application with much floating-point content, a type-aware binary format with a corresponding API can easily be 20–30 times faster than XML on documents of any size [Kangasharju and Tarkoma, 2007]. Since the forthcoming EXI format includes native support for typed data, it is likely that an EXI API will emerge to support such use cases. We believe our results in XML API design could prove helpful to this EXI API design as well.

Part V

Conclusions

CHAPTER 9

Conclusions

The butler did it.

We now have several years of experience in using this messaging system, or its predecessor, as a component in the Fuego middleware. There is much that we have learned about writing software for mobile computing and about the general state of XML messaging. Based on this experience, we remain optimistic that in the future, mobile devices need not be relegated into a ghetto, but can participate on the Internet as full-fledged members.

9.1 Contributions

To summarize the work presented in this dissertation, we can distinguish the following significant contributions:

XML Processing
: Existing XML APIs usually require applications to also know and care about XML as such. In our view, the API should provide a unified approach to processing XML either as XML or as just a data interchange format, and our work on the XAS API validates this view. Our expectation is that XML processing in the future will require extensibility and versatility from the processing system, as developers invent novel

ways to treat XML, and we designed the XAS API to explicitly support such development.

The multi-layered view of XML illustrated in Figure 5.11 that XAS supports has already proved itself. This is evidenced by our being able to implement efficient XML processing applications in different areas [Kangasharju, 2007; Lindholm and Kangasharju, 2008] using only the generic functionality of XAS, whereas previous works in similar areas [Fernandes and Raghavachari, 2005; Imamura et al., 2002; Lu et al., 2005] have been special-purpose implementations with poorer functionality.

XML Binary Serialization

While there exist a number of binary serialization formats for XML data, Xebu has a few unique features. First of all, the design explicitly considers small mobile devices by bounding the dynamic memory required during processing, and as the measurements in subsection 8.3.3 show, this does not penalize Compactness excessively. Second, Xebu is able to use structure information from a schema, does not require it, and is capable of processing some invalid documents. Apart from the EXI format, which is still in development, none of the well-known binary formats have all of these features.

XML Security with Binary XML

While there has been some consideration toward XML security in the binary XML world (see, e.g., [Williams, 2007]), our work goes further than anything prior. We have explicitly considered the existing specifications and identified minimal changes needed for improved support of alternate formats. Our recommendations on efficient usage patterns for XML security in the context of mobile computing would seem to provide clear benefits.

Measurements in Real Environments

We performed extensive measurements with a number of real mobile devices and real wireless networks. The analysis in section 7.4 provides useful guidance for application developers in determining which factors are important in mobile computing. We also noted that the varying conditions

in different wireless networks and different communication patterns make issues such as whether to apply generic compression very dependent on the application itself and its environment.

Complete Available System
The implemented messaging system is a complete work, and all of its components have been designed for mobile computing. It will be released under a F/OSS license, available to others for use, extension, or further development.

9.2 Lessons Learned

In the final evaluation of our previous system [Kangasharju, 2006] we noted some possible enhancements and identified future work related to XML messaging. As a result of the previous system being mostly tested on laptops, it did not perform as well on mobile phones as could have been hoped. For this work, we essentially restarted the implementation from scratch, designing it based on the ideas identified as successful while being able to ditch the baggage that contributed nothing but bloat. Such a "Plan to throw one away" approach is a well-recognized one in software engineering [Brooks, 1995].

The measurements in chapter 7 demonstrate one thing: there is no single solution for all the needs of pervasive computing. Because of this, we consider context-awareness even in the middleware to be a necessary component of the future pervasive systems, so that the system can adapt even its message syntax and used compression methods to the available networking conditions. Unlike the context that is traditionally considered, e.g., location or activity, context for middleware is something related to the device such as current network connectivity or battery energy level.

The ability of the messaging system to support a variety of networks, communication patterns, and message formats is a crucial one. As the communication environment changes, so must the system's behavior, and the more options a system has, the more likely it is to be able to retain good performance even in varying conditions. As a counterpoint to this, we must note that such versatility

increases code size. Our current system, including all of the measurement applications, comes to slightly over 300 kB after obfuscation, which may be a bit high for some applications or devices.

We believe that for research work on mobile devices, Java is an appropriate programming language to use. The ability to basically begin programming immediately without having to learn a number of conventions that should not be necessary in a modern programming language is a significant time-saver. Also, due to this better accessibility, implementing a middleware platform in Java is likely to have more impact, as the chance of adoption by others is increased. Finally, we do not consider Java performance to be a hindrance, but contend that it is the device itself that in the end limits what can be usefully done.

The key points in the design of Xebu that are rarely present in other binary formats are the presence of explicit tokens in the serialized document, bounding the size of the token mappings, and the ability of the COA to process some invalid documents. Of these, the first is closely tied to the second, as without a bounded token mapping there is no reason for an eviction policy. The presence of the explicit tokens makes this policy a matter for only the serializer and not a question of interoperability. The usefulness of the COA approach, however, is now somewhat in question, as the proposed EXI format is able to process any invalid document.

9.3 Binary XML Future

The Xebu binary format fulfills its purpose well. It achieves a reasonable size reduction, especially with a suitable schema, without compromising on processing speed like generic compression schemes do. We have identified use of the default character encoding system as a major bottleneck and are considering ways to replace it. The method used in the proposed EXI format [W3C, 2007a] in particular holds a lot of promise in our opinion, and we may replace Xebu's character encoding with that shortly.

While Xebu as a format is surpassed by Efficient XML, and therefore the eventual EXI, as shown by the measurements in section 8.3, we believe it still has something to recommend for it. Namely, the implementation is very simple, and the design of the

format explicitly considers memory-constrained devices. Bounding the processor state is especially useful in processing a stream of messages while retaining state between the messages, a scenario for which Xebu was explicitly designed and which other binary formats do not seem to consider as their state grows without bounds. This *bounded tables* feature is also under consideration for the EXI format [W3C, 2007a, appendix E].

Another reason to consider Xebu is that it is the only modern binary format for mobile devices with a F/OSS license. Because of this, several people working on mobile devices are already interested in Xebu, though we are not aware of any adoption in products. We believe this situation may continue for a while, as the first similarly licensed EXI implementations are likely to be written for regular computers, and efficient implementations on mobile devices may take a while to appear.

In the mobile space, it is extremely probable that a binary format will be adopted in the near future, and most probably this will be EXI. The work presented here is an extensive consideration of various issues in adopting a new XML format, and should prove useful for implementers of future EXI-based systems. Our participation in the EXI WG also has the goal of widening the dissemination of these results, to the benefit of future adopters.

9.4 Future Work

In our previous work [Kangasharju, 2006] we identified three future work items for XML messaging. Of these, the work presented here has addressed XML security in a satisfactory way, though we do not yet have an implementation of the more efficient authentication and encryption schemes.

Of the other future enhancements, we have performed preliminary experiments on both binary-aware XML APIs and content-based XML routing. On the binary API front our experiments in data binding show that a system aware of the token mappings of Xebu can perform much better than one required to do string matching of the XML. When the problem is not in the structure but in the content, we have already demonstrated significant gains

from the access to the underlying byte stream and typed data support of XAS [Kangasharju and Tarkoma, 2007].

However, when considering a binary-aware API, we must also consider the added complexity. Our preliminary experiments did not use the dynamic tokenization of Xebu, and it is likely that an efficient binary-aware API would have to rely only on static token mappings extracted from a schema. Because of the uncertainty of the benefits, we consider it better to wait for EXI to progress a bit more and then start designing a binary-aware API for that format. As there are some commonalities between EXI and Xebu, it might be also possible to implement the API for Xebu as well.

On the content-based XML routing front, we have so far only charted the current state of the art. This has revealed the existing work to be more database-oriented and to not consider the propagation of filters and especially deletions that are crucial for an XML-based P/S system [Tarkoma and Kangasharju, 2006]. At the moment we are not sure whether to progress towards exact matching at all routers, with a more complex routing table implementation, or towards simplification of the propagated XPath filters, with the requirement to do exact filtering at the edge nodes. Both approaches appear to be viable avenues for future research.

In addition, as our experience in using Xebu has grown, we have identified some deficiencies that could be reasonably rectified [Kangasharju and Koskimies, 2008]. First, the elimination of explicit token values when not all values have been yet assigned would help with small documents. Second, the COA needs to be somewhat reconsidered. Its current form as pure finite automata does not work well with the hierarchical structure of XML, and it would therefore be useful to consider some form of a stack extension. Finally, while the typed data in Xebu is usable currently, it has proved difficult to integrate with existing type-aware XML processing pipelines, so some reconsideration of the typed data implementation would seem advisable.

9.5 Concluding Remarks

As a final conclusion to the work we have performed, we do not believe that XML messaging is fundamentally incompatible with

the world of mobile devices. Of the components that we identified as needing improvement, the most likely one to be replaced is the XML format. We do not see new messaging protocols to be as easily adopted, and consider the disappearance of BEEP evidence that even sound designs have difficulties to establish themselves. The area of XML APIs we consider to have been somewhat neglected, and believe that there is room for innovations in API design. It is possible that with the maturing of EXI interest in APIs will be renewed, as people will begin designing EXI APIs and at the same time, we hope, improving the state of XML APIs in general.

It would also appear that mobile phones will be usable in the future distributed systems as full-fledged members. The performance that we measured, especially its development in the past few years, indicates that rich applications can be written for the current and upcoming devices. We were also delighted to note that the stability of the operating system and language implementation have improved, so we expect programming to also become easier, as fewer workarounds are needed. There still remains work to do before writing pervasive applications is as easy as writing fixed-network distributed applications, though.

Part VI

Appendix and References

CHAPTER A

Additional Code

The complete code of the messaging system will eventually be made available through the Helsinki Open Source Laboratory[1]. However, the precise code used can be useful in fully understanding some parts, especially the benchmarks of subsection 7.2.3, so we reproduce relevant code excerpts in this appendix. We include only the code that is relevant, and not, e.g., `main` or `commandAction` methods, or measurement drivers when they do not contribute to the results significantly.

A.1 The Knuth Benchmark

The code for the Knuth benchmark is a direct translation of the C code given in [Knuth, 1997b, section 3.6]. It also includes a `ranStart` method to fill the `ranX` array initially, but as we do not measure the performance of that, it has been left out of this appendix.

```
private static final int KK = 100;
private static final int LL = 37;
private static final int MM = 1 << 30;
private static final int INNER = 300;

private static int ranX[] = new int[KK];
```

[1] http://hoslab.cs.helsinki.fi/

```
private static int modDiff (int x, int y) {
  return (x - y) & (MM - 1);
}

public static void ranArray (int[] aa) {
  int i, j;
  for (j = 0; j < KK; j++) {
    aa[j] = ranX[j];
  }
  for (; j < aa.length; j++) {
    aa[j] = modDiff(aa[j - KK], aa[j - LL]);
  }
  for (i = 0; i < LL; i++, j++) {
    ranX[i] = modDiff(aa[j - KK], aa[j - LL]);
  }
  for (; i < KK; i++, j++) {
    ranX[i] = modDiff(aa[j - KK], ranX[i - LL]);
  }
}

// Driver code, this is timed
for (int j = 0; j < INNER; j++) {
  ranArray(aa);
}
```

A.2 The Virtual Benchmark

```
interface Iface {
  int s ();
  int longMethodName ();
  int veryLongUsefulMethodName ();
}

class A implements Iface {

  private int[] aa = new int[100];
```

```
  public int longMethodName () {
    KnuthBench.ranArray(aa);
    return aa[0];
  }

  public int s () {
    KnuthBench.ranArray(aa);
    return aa[0];
  }

  public int veryLongUsefulMethodName () {
    KnuthBench.ranArray(aa);
    return aa[0];
  }
}

class B implements Iface {
  // Exact copy of A
}

private static final int INNER = 1000;

// Initialization for measurement
A a = new A();
B b = new B();
Iface ii = null;
int sum = 0;
boolean[] xs = new boolean[INNER];
for (int i = 0; i < INNER; i++) {
  xs[i] = (random.nextInt() >>> 31) == 0;
}

// Driver loop
for (int j = 0; j < INNER; j++) {
  switch (style.getSelectedIndex()) {
    case 0: // Same
      ii = a;
      break;
```

```
    case 1: // Flip
      ii = (j & 1) == 0 ? (Iface) a : b;
      break;
    case 2: // Random
      ii = xs[j] ? (Iface) a : b;
      break;
  }
  switch (method.getSelectedIndex()) {
    case 0: // Short
      sum += ii.s();
      break;
    case 1: // Long
      sum += ii.longMethodName();
      break;
    case 2: // Longest
      sum += ii.veryLongUsefulMethodName();
      break;
  }
}
```

A.3 The Net Benchmark

```
private static final int LOOP = 30;
private static final int FILE_SIZE = 2 * 1024 * 1024;
// data is initialized to random bytes
private static byte[] data = new byte[8192];

private static void streamServer (InputStream in, OutputStream out)
    throws IOException {
  while (true) {
    int b = in.read();
    if (b == 'l') {
      out.write(b);
      out.flush();
    } else if (b == 'b') {
      int n = FILE_SIZE;
      while (n > 0) {
```

```
        out.write(data);
        n -= data.length;
      }
      out.flush();
      return;
    }
  }
}

private static void streamClient (InputStream in, OutputStream out)
    throws IOException {
  for (int i = 0; i < LOOP; i++) {
    // Each iteration is timed
    out.write('l');
    out.flush();
    int b = in.read();
    }
  }
  out.write('b');
  out.flush();
  int n;
  // This loop is timed
  while ((n = in.read(data)) >= 0) {
    // Do nothing
  }
}

private static void httpClient (String url) throws IOException {
  for (int i = 0; i < LOOP; i++) {
    HttpConnection conn = (HttpConnection) Connector.open(url);
    conn.setRequestMethod(HttpConnection.POST);
    conn.setRequestProperty("Connection", "close");
    OutputStream out = conn.openOutputStream();
    out.write('l');
    int code = conn.getResponseCode();
    if (code == HttpConnection.HTTP_OK) {
      InputStream in = conn.openInputStream();
      int b = in.read();
    }
```

```
    conn.close();
  }
  HttpConnection conn = (HttpConnection) Connector.open(url);
  conn.setRequestMethod(HttpConnection.POST);
  conn.setRequestProperty("Connection", "close");
  OutputStream out = conn.openOutputStream();
  out.write('b');
  int code = conn.getResponseCode();
  if (code == HttpConnection.HTTP_OK) {
    InputStream in = conn.openInputStream();
    int n;
    while ((n = in.read(data)) >= 0) {
      // Do nothing
    }
  }
  conn.close();
}
```

A.4 Message Experiment Schema

The messages used in the experiments of chapter 7 are SOAP messages, each containing some headers defined by the messaging system for addressing and correlating messages, and a body containing a sequence of elements representing credit cards. The precise RELAX NG compact syntax schema for the messages is given below. The `mts:sender` header block is present in messages for which a response is expected and the `mts:response` header block is present in response messages.

```
namespace soap = "http://www.w3.org/2003/05/soap-envelope"
namespace mts = "http://www.hiit.fi/fuego/fc/mts"
namespace exp = "http://www.hiit.fi/fuego/fc/exper"

start = element soap:Envelope {
  header, body
}

header = element soap:Header {
```

```
  element mts:id { xsd:long },
  element mts:target { xsd:string },
  element mts:sender { xsd:string }?,
  element mts:response { xsd:long }?
}

body = element soap:Body {
  element mts:data {
    element data {
      element exp:card {
        element exp:name { xsd:string },
        element exp:number { xsd:string },
        element exp:expYear { xsd:int },
        element exp:expMonth { xsd:int }
      }*
    }
  }
}
```

References

ABU-GHAZALEH, N. AND LEWIS, M. J. (Nov. 2005). Differential deserialization for optimized SOAP performance. In *ACM/IEEE Conference on Supercomputing*.
(Cited on page 41.)

ABU-GHAZALEH, N., LEWIS, M. J., AND GOVINDARAJU, M. (Jun. 2004). Differential serialization for optimized SOAP performance. In *13th IEEE Symposium on High Performance Distributed Computing*. pp. 55–64.
(Cited on page 40.)

AIKEN, B., STRASSNER, J., CARPENTER, B. E., FOSTER, I., LYNCH, C., MAMBRETTI, J., MOORE, R., AND TEITELBAUM, B. (Feb. 2000). *RFC 2768: Network Policy and Services: A Report of a Workshop on Middleware*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc2768.txt
(Cited on pages 4 and 56.)

AITENBICHLER, E., KANGASHARJU, J., AND MÜHLHÄUSER, M. (Mar. 2005). Experiences with MundoCore. In *PerCom Workshop on Middleware Support for Pervasive Computing*. pp. 168–172.
(Cited on page 62.)

ALLMAN, M. AND PAXSON, V. (Sep. 1999). On estimating end-to-end network path properties. In *ACM SIGCOMM 1999*. pp. 263–274.
(Cited on page 53.)

APPEL, A. W. (1998). *Modern Compiler Implementation in ML* (Cam-

bridge University Press, Cambridge, United Kingdom).
(Cited on page 127.)

AUGUSTEIJN, L. (Sep. 1998). Sorting morphisms. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira (Eds.), *Advanced Functional Programming* (Springer-Verlag, Heidelberg, Germany), *Lecture Notes in Computer Science*, vol. 1608, pp. 1–27.
(Cited on page 131.)

AVARO, O. AND SALEMBIER, P. (Jun. 2001). MPEG-7 systems: Overview. *IEEE Transactions on Circuits and Systems for Video Technology*, *11*(6), pp. 760–764.
(Cited on page 117.)

BAGRODIA, R., CHU, W. W., KLEINROCK, L., AND POPEK, G. (Dec. 1995). Vision, issues, and architecture for nomadic computing. *IEEE Personal Communications*, *2*(6), pp. 14–27.
(Cited on page 55.)

BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., AND KATZ, R. H. (Dec. 1997). A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, *5*(6), pp. 756–769.
(Cited on page 54.)

BALASUBRAMANIAM, S. AND PIERCE, B. C. (Oct. 1998). What is a file synchronizer? In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*. pp. 98–108.
(Cited on page 61.)

BARR, K. C. AND ASANOVIC, K. (Aug. 2006). Energy-aware lossless data compression. *ACM Transactions on Computer Systems*, *24*(3), pp. 250–291.
(Cited on page 47.)

BAYER, M. E. (Sep. 2005). *Analysis of Binary XML Suitability for NATO Tactical Messaging*. Master's thesis, Naval Postgraduate School, Monterey, California, USA.
(Cited on page 116.)

BEA (Oct. 2003). *JSR 173: Streaming API for XML*. BEA Systems Inc., San Jose, California, USA.
URL http://jcp.org/aboutJava/communityprocess/final/jsr173/index.html
(Cited on page 22.)

BENETAZZO, L., BERTOCCO, M., NARDUZZI, C., AND TITTOTO, R. (Sep. 2003). Analysis and measurement of TCP/IP performance over GPRS networks. In [Conti et al., 2003], pp. 261–275.
(Cited on page 53.)

BERNERS-LEE, T., FIELDING, R. T., AND MASINTER, L. (Jan. 2005). *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc3986.txt
(Cited on pages 16 and 69.)

BIRRELL, A. D. AND NELSON, B. J. (Feb. 1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, *2*(1), pp. 39–59.
(Cited on page 56.)

BISHOP, M. (Nov. 2003). *Computer Security: Art and Science* (Addison-Wesley, Boston, Massachusetts, USA).
(Cited on page 28.)

BlueSIG (Nov. 2004). *Specification of the Bluetooth System, Core Package version 2.0*. Bluetooth SIG.
(Cited on page 49.)

BORDER, J., KOJO, M., GRINER, J., MONTENEGRO, G., AND SHELBY, Z. (Jun. 2001). *RFC 3135: Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc3135.txt
(Cited on page 54.)

BROOKS, JR., F. P. (1995). *The Mythical Man-Month* (Addison-Wesley, Boston, Massachusetts, USA).
(Cited on page 203.)

BROWNELL, D. (Jan. 2002). *SAX2* (O'Reilly, Sebastopol, California, USA).
(Cited on page 22.)

BRÜGGEMANN-KLEIN, A., MURATA, M., AND WOOD, D. (Apr. 2001). Regular tree and regular hedge languages over unranked alphabets. Tech. Rep. HKUST-TCSC-2001-05, Hong Kong University of Science and Technology.
(Cited on page 18.)

BURROWS, M. AND WHEELER, D. J. (May 1994). A block-sorting lossless data compression algorithm. Research Report 124, Systems Research Center, Digital Equipment Corporation.
(Cited on page 111.)

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. (Aug. 1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* (John Wiley & Sons, Hoboken, New Jersey, USA).
(Cited on page 94.)

CAI, J. AND GOODMAN, D. J. (Oct. 1997). General packet radio service in GSM. *IEEE Communications Magazine*, *35*(10), pp. 122–131.
(Cited on page 48.)

CAMP, T., BOLENG, J., AND DAVIES, V. (Aug. 2002). A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing*, *2*(5), pp. 483–502.
(Cited on page 56.)

CAMPADELLO, S., KOSKIMIES, O., RAATIKAINEN, K., AND HELIN, H. (Sep. 2000). Wireless Java RMI. In *Fourth International Enterprise Distributed Object Computing Conference*. pp. 114–123.
(Cited on page 58.)

CAPONE, A., FRATTA, L., AND MARTIGNON, F. (Apr. 2004). Bandwidth estimation schemes for TCP over wireless networks. *IEEE Transactions on Mobile Computing*, *3*(2), pp. 129–143.
(Cited on page 53.)

CAPORUSCIO, M., CARZANIGA, A., AND WOLF, A. L. (Dec. 2003). Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, *29*(12), pp. 1059–1071.
(Cited on page 59.)

CARZANIGA, A., ROSENBLUM, D. S., AND WOLF, A. L. (Aug. 2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, *19*(3), pp. 332–383.
(Cited on page 58.)

CASNER, S. L. AND JACOBSON, V. (Feb. 1999). *RFC 2508: Compressing IP/UDP/RTP Headers for Low-Speed Serial Links*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc2508.txt
(Cited on page 112.)

CHEN, W., NAUGHTON, J. F., AND BERNSTEIN, P. A. (Eds.) (May 2000). *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*.
(Cited on pages 227 and 235.)

CHENEY, J. (Mar. 2001). Compressing XML with multiplexed hierarchical PPM models. In *Data Compression Conference*. pp. 163–172.
(Cited on page 111.)

CHIU, K., DEVADITHYA, T., LU, W., AND SLOMINSKI, A. (Dec. 2005). A binary XML for scientific applications. In *First International Conference on e-Science and Grid Computing*.
(Cited on page 115.)

CHIU, K., GOVINDARAJU, M., AND BRAMLEY, R. (Jul. 2002). Investigating the limits of SOAP performance for scientific computing. In *11th IEEE Symposium on High Performance Distributed Computing*. pp. 246–254.
(Cited on page 39.)

CHIU, K. AND LU, W. (May 2004). A compiler-based approach to schema-specific XML parsing. In *First International Workshop on*

High Performance XML Processing.
(Cited on pages 41 and 42.)

CLEARY, J. G. AND WITTEN, I. H. (Apr. 1984). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, *32*(4), pp. 396–402.
(Cited on pages 47 and 111.)

COKUS, M. AND WINKOWSKI, D. (Dec. 2002). XML sizing and compression study for military wireless data. In *XML Conference and Exposition*.
(Cited on pages 117 and 118.)

CONTI, M., GIORDANO, S., GREGORI, E., AND OLARIU, S. (Eds.) (Sep. 2003). *Personal Wireless Communications*, *Lecture Notes in Computer Science*, vol. 2775 (Springer-Verlag, Heidelberg, Germany).
(Cited on pages 221 and 233.)

COOK, W. R. AND BARFIELD, J. (Sep. 2006). Web services versus distributed objects: A case study of performance and interface design. In [IEEE, 2006], pp. 419–426.
(Cited on page 60.)

DEERING, S. E. AND HINDEN, R. M. (Dec. 1998). *RFC 2460: Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc2460.txt
(Cited on page 51.)

DETTMER, R. (Jul. 1991). GSM: European cellular goes digital. *IEE Review*, *37*(7), pp. 253–257.
(Cited on page 48.)

DEUTSCH, L. P. (May 1996a). *RFC 1951: DEFLATE Compressed Data Format Specification version 1.3*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc1951.txt
(Cited on page 47.)

DEUTSCH, L. P. (May 1996b). *RFC 1952: GZIP File Format Specification Version 4.3*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc1952.txt
(Cited on page 104.)

DEY, A. K. (Feb. 2001). Understanding and using context. *Personal and Ubiquitous Computing*, *5*(1), pp. 4–7.
(Cited on page 61.)

DIERKS, T. AND ALLEN, C. (Jan. 1999). *RFC 2246: The TLS Protocol*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc2246.txt
(Cited on page 31.)

DIFFIE, W. AND HELLMAN, M. (Nov. 1976). New directions in cryptography. *IEEE Transactions on Information Theory*, *22*(6), pp. 644–654.
(Cited on pages 29 and 30.)

EASTLAKE, D. E. AND JONES, P. E. (Sep. 2001). *RFC 3174: US Secure Hash Algorithm 1 (SHA1)*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc3174.txt
(Cited on page 30.)

ELFWING, R., PAULSSON, U., AND LUNDBERG, L. (Dec. 2002). Performance of SOAP in Web service environment compared to CORBA. In *Ninth Asia-Pacific Software Engineering Conference*. pp. 84–93.
(Cited on pages 59 and 60.)

ELGAMAL, T. (Jul. 1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, *31*(4), pp. 473–481.
(Cited on page 29.)

EUGSTER, P. T., FELBER, P. A., GUERRAOUI, R., AND KERMARREC, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, *35*(2), pp. 114–131.
(Cited on page 58.)

FERGUSON, N. AND SCHNEIER, B. (2003). *Practical Cryptography* (Wiley Publishing, Indianapolis, Indiana, USA).
(Cited on page 161.)

FERNANDES, R. AND RAGHAVACHARI, M. (Nov. 2005). Inflatable XML processing. In G. Alonso (Ed.), *Proceedings of the 6th International ACM/IFIP/USENIX Middleware Conference* (Springer-Verlag, Heidelberg, Germany), *Lecture Notes in Computer Science*,

vol. 3790, pp. 144–163.
(Cited on pages 107 and 202.)

FIEGE, L., GARTNER, F. C., KASTEN, O., AND ZEIDLER, A. (Jun. 2003). Supporting mobility in content-based publish/subscribe middleware. In M. Endler and D. C. Schmidt (Eds.), *Proceedings of the 4th International Conference on Middleware* (Springer-Verlag, Heidelberg, Germany), *Lecture Notes in Computer Science*, vol. 2672, pp. 103–122.
(Cited on page 59.)

FIELDING, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine.
(Cited on page 23.)

FIELDING, R., GETTYS, J., MOGUL, J., NIELSEN, H. F., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. (Jun. 1999). *RFC 2616: Hypertext Transfer Protocol — HTTP/1.1*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc2616.txt
(Cited on pages 69 and 105.)

FIX (Aug. 2001). *The Financial Information Exchange Protocol (FIX) version 4.3*. FIX Protocol Ltd.
(Cited on page 39.)

FOKKER, J. (May 1995). Functional parsers. In J. Jeuring and E. Meijer (Eds.), *Advanced Functional Programming* (Springer-Verlag, Heidelberg, Germany), *Lecture Notes in Computer Science*, vol. 925, pp. 1–23.
(Cited on page 127.)

FORNO, F. AND SAINT-ANDRE, P. (Oct. 2005). *JEP-0072: SOAP Over XMPP*. Jabber Software Foundation.
URL http://www.jabber.org/jeps/jep-0072.html
(Cited on page 25.)

FOSTER, I. AND KESSELMAN, C. (Eds.) (Aug. 2004). *The Grid 2: Blueprint for a New Computing Infrastructure* (Morgan Kaufmann Publishers).
(Cited on page 39.)

FREED, N. AND BORENSTEIN, N. (Nov. 1996a). *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc2045.txt
(Cited on page 26.)

FREED, N. AND BORENSTEIN, N. (Nov. 1996b). *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc2046.txt
(Cited on pages 26 and 96.)

FREIER, A. O., KARLTON, P., AND KOCHER, P. C. (Nov. 1996). *The SSL Protocol Version 3.0*. Netscape Communications.
URL http://wp.netscape.com/eng/ssl3/draft302.txt
(Cited on page 31.)

FURUSKÄR, A., MAZUR, S., MÜLLER, F., AND OLOFSSON, H. (Jun. 1999). EDGE: Enhanced data rates for GSM and TD-MA/136 evolution. *IEEE Personal Communications*, *6*(3), pp. 56–66.
(Cited on page 49.)

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, Boston, Massachusetts, USA).
(Cited on page 93.)

GARFINKEL, S. (Dec. 1994). *PGP: Pretty Good Privacy* (O'Reilly, Sebastopol, California, USA).
(Cited on page 29.)

GARFINKEL, S. L., MARGRAVE, D., SCHILLER, J. I., NORDLANDER, E., AND MILLER, R. C. (2005). How to make secure email easier to use. In G. C. van der Veer and C. Gale (Eds.), *CHI '05: CHI '05 extended abstracts on Human factors in computing systems* (ACM Press), pp. 701–710.
(Cited on page 30.)

GAROFALAKIS, M., GIONIS, A., RASTOGI, R., SESHADRI, S., AND SHIM, K. (May 2000). XTRACT: A system for extracting document type descriptors from XML documents. In [Chen et al.,

2000], pp. 165–176.
(Cited on page 185.)

GEER, D. (Jul. 2006). UWB standardization effort ends in controversy. *IEEE Computer*, *39*(7), pp. 13–16.
(Cited on page 51.)

GELERNTER, D. (Jan. 1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, *7*(1), pp. 80–112.
(Cited on page 61.)

GIRARDOT, M. AND SUNDARESAN, N. (May 2000). Millau: an encoding format for efficient representation and exchange of XML over the Web. In *Ninth International World Wide Web Conference*.
URL http://www9.org/w9cdrom/154/154.html
(Cited on page 114.)

GRIMM, R., DAVIS, J., LEMAR, E., MACBETH, A., SWANSON, S., ANDERSON, T., BERSHAD, B., BORRIELLO, G., GRIBBLE, S., AND WETHERALL, D. (Nov. 2004). System support for pervasive applications. *ACM Transactions on Computer Systems*, *22*(4), pp. 421–486.
URL http://www.cs.nyu.edu/rgrimm/one.world/papers/tocs04.pdf
(Cited on page 62.)

HAAHR, M., CUNNINGHAM, R., AND CAHILL, V. (Aug. 1999). Supporting CORBA applications in a mobile environment. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*. pp. 36–47.
(Cited on page 58.)

HALSTEAD, JR., R. H. (Oct. 1985). MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, *7*(4), pp. 501–538.
(Cited on page 57.)

HAROLD, E. R. (2006). XOM design principles.
URL http://www.xom.nu/designprinciples.xhtml
(Cited on page 90.)

HARTIKAINEN, V.-M., LIIMATAINEN, P. P., AND MIKKONEN, T. (Feb. 2006). On mobile Java memory consumption. In *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*.
(Cited on page 87.)

HEAD, M. R., GOVINDARAJU, M., VAN ENGELEN, R., AND ZHANG, W. (Nov. 2006). Benchmarking XML processors for applications in Grid Web services. In *ACM/IEEE Conference on Supercomputing*.
(Cited on pages 40 and 192.)

HERICKO, M., JURIC, M. B., ROZMAN, I., BELOGLAVEC, S., AND ZIVKOVIC, A. (Aug. 2003). Object serialization analysis and comparison in Java and .NET. *ACM SIGPLAN Notices*, *38*(8), pp. 44–54.
(Cited on page 60.)

HONG, J. I. AND LANDAY, J. A. (2001). An infrastructure approach to context-aware computing. *Human-Computer Interaction*, *16*(2/4), pp. 287–303.
(Cited on page 62.)

IEEE (Mar. 1999). *IEEE Std 802.11 — Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, USA.
(Cited on page 49.)

IEEE (Jul. 2004a). *IEEE International Conference on Web Services* (Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, USA).
(Cited on pages 246 and 254.)

IEEE (Oct. 2004b). *IEEE Std 802.16 — Air Interface for Fixed Broadband Wireless Access Systems*. Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, USA.
(Cited on page 50.)

IEEE (Sep. 2006). *IEEE International Conference on Web Services* (Institute of Electrical and Electronic Engineers, Piscataway, New

Jersey, USA).
(Cited on pages 224, 232, and 254.)

IMAMURA, T., CLARK, A., AND MARUYAMA, H. (Nov. 2002). A stream-based implementation of XML encryption. In *ACM Workshop on XML Security*. pp. 11–17.
(Cited on pages 43, 101, and 202.)

ISO (1986). *ISO 8879:1986. Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland.
(Cited on page 13.)

ITU (2002a). *Abstract Syntax Notation One (ASN.1) Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. ITU-T Rec. X.690.
(Cited on pages 114, 115, and 120.)

ITU (2002b). *Abstract Syntax Notation One (ASN.1) Specification of Basic Notation*. International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. ITU-T Rec. X.680.
(Cited on page 114.)

ITU (2002c). *Abstract Syntax Notation One (ASN.1) Specification of Packed Encoding Rules (PER)*. International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. ITU-T Rec. X.691.
(Cited on pages 114, 117, and 119.)

ITU (2004). *Mapping W3C XML Schema Definitions into ASN.1*. International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. ITU-T Rec. X.694.
(Cited on pages 117, 119, and 120.)

ITU (May 2005). *Generic Applications of ASN.1: Fast infoset*. International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. ITU-T Rec. X.891.
(Cited on page 119.)

JACOBSON, V. (Feb. 1990). *RFC 1144: Compressing TCP/IP Headers for Low-Speed Serial Links*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc1144.txt
(Cited on page 112.)

JAIN, R. (1991). *The Art of Computer Systems Performance Analysis* (John Wiley & Sons, Hoboken, New Jersey, USA).
(Cited on pages 153 and 154.)

JELLIFFE, R. (Oct. 2002). *The Schematron Assertion Language 1.5*. Academia Sinica Computing Centre.
URL http://xml.ascc.net/resource/schematron/Schematron2000.html
(Cited on page 18.)

JOHNSON, D. B., PERKINS, C. E., AND ARKKO, J. (Jun. 2004). *RFC 3775: Mobility Support in IPv6*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc3775.txt
(Cited on page 52.)

JURIC, M. B., KEZMAH, B., HERICKO, M., ROZMAN, I., AND VEZOCNIK, I. (May 2004). Java RMI, RMI tunneling and Web services comparison and performance analysis. *ACM SIGPLAN Notices*, *39*(5), pp. 58–65.
(Cited on page 60.)

KANGASHARJU, J. (Jun. 2004). Abstract mobile message exchange. In *Fourth Berkeley-Helsinki Ph.D. Student Workshop on Telecommunication Software Architectures*.
URL http://www.cs.helsinki.fi/u/jkangash/amme.pdf
(Cited on page 87.)

KANGASHARJU, J. (Apr. 2006). *An XML Messaging Service for Mobile Devices*. Licentiate thesis, University of Helsinki, Department of Computer Science, Helsinki, Finland.
URL http://ethesis.helsinki.fi/julkaisut/mat/tieto/lt/kangasharju/
(Cited on pages 7, 203, and 205.)

KANGASHARJU, J. (Jul. 2007). Efficient implementation of XML security for mobile devices. In *IEEE International Conference on Web*

Services (Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, USA), pp. 134–141.
URL http://dx.doi.org/10.1109/ICWS.2007.81
(Cited on pages 8 and 202.)

KANGASHARJU, J. AND KOSKIMIES, O. (Jun. 2008). Using bit-efficient XML to optimize data transfer of XForms-based mobile services. In *10th International Conference on Enterprise Information Systems*. In submission.
URL http://www.hiit.fi/files/fi/fc/papers/kangasharju-iceis08-submit.pdf
(Cited on pages 133 and 206.)

KANGASHARJU, J. AND LINDHOLM, T. (Feb. 2005). A sequence-based type-aware interface for XML processing. In M. H. Hamza (Ed.), *Proceedings of the Ninth IASTED International Conference on Internet and Multimedia Systems and Applications* (ACTA Press), pp. 83–88.
URL http://www.cs.helsinki.fi/u/jkangash/xml-interface.pdf
(Cited on pages 7, 90, and 95.)

KANGASHARJU, J., LINDHOLM, T., AND TARKOMA, S. (May 2005a). Requirements and design for XML messaging in the mobile environment. In N. Anerousis and G. Kormentzas (Eds.), *Proceedings of the Second International Workshop on Next Generation Networking Middleware*. pp. 29–36.
URL http://www.cs.helsinki.fi/u/jkangash/xml-messaging-mobile.pdf
(Cited on page 7.)

KANGASHARJU, J., LINDHOLM, T., AND TARKOMA, S. (Sep. 2006). On encrypting and signing binary XML messages in the wireless environment. In [IEEE, 2006], pp. 637–644.
URL http://www.hiit.fi/files/fi/fc/papers/icws06-binary-security.pdf
(Cited on pages 8, 38, 47, 90, 101, 165, and 167.)

KANGASHARJU, J., LINDHOLM, T., AND TARKOMA, S. (Nov. 2007a). XML messaging on mobile devices: From requirements to implementation. *Computer Networks*, *51*(16), pp. 4634–4654.

URL http://dx.doi.org/10.1016/j.comnet.2007.06.008
(Cited on page 8.)

KANGASHARJU, J., LINDHOLM, T., AND TARKOMA, S. (2007b). XML security with binary XML for mobile Web services. *International Journal of Web Services Research*. To appear.
URL http://www.cs.helsinki.fi/u/jkangash/kangasharju-jwsr07.pdf
(Cited on page 175.)

KANGASHARJU, J. AND RAATIKAINEN, K. (Sep. 2003). Byte-efficient representation of XML messages. In [W3C, 2003b].
URL http://www.w3.org/2003/08/binary-interchange-workshop/08-xebu.pdf
(Cited on page 8.)

KANGASHARJU, J. AND TARKOMA, S. (Jun. 2007). Benefits of alternate XML serialization formats in scientific computing. In *Workshop on Service-Oriented Computing Performance*. pp. 23–30.
URL http://doi.acm.org/10.1145/1272457.1272461
(Cited on pages 40, 192, 197, and 206.)

KANGASHARJU, J., TARKOMA, S., AND LINDHOLM, T. (Nov. 2005b). Xebu: A binary format with schema-based optimizations for XML data. In A. H. H. Ngu, M. Kitsuregawa, E. Neuhold, J.-Y. Chung, and Q. Z. Sheng (Eds.), *The 6th International Conference on Web Information Systems Engineering* (Springer-Verlag, Heidelberg, Germany), *Lecture Notes in Computer Science*, vol. 3806, pp. 528–535.
URL http://www.hiit.fi/files/fi/fc/papers/wise05-xebu.pdf
(Cited on pages 7, 120, and 121.)

KANGASHARJU, J., TARKOMA, S., AND RAATIKAINEN, K. (Sep. 2003). Comparing SOAP performance for various encodings, protocols, and connections. In [Conti et al., 2003], pp. 397–406.
URL http://www.cs.helsinki.fi/u/jkangash/soap-performance.pdf
(Cited on pages 6 and 75.)

KARRI, R. AND MISHRA, P. (Apr. 2003). Optimizing the energy consumed by secure wireless sessions — Wireless Transport

Layer Security case study. *Mobile Networks and Applications*, *8*(2), pp. 177–185.
(Cited on page 47.)

KESHAV, S. (2005). Why cell phones will dominate the future Internet. *Computer Communication Review*, *35*(2), pp. 83–86.
(Cited on pages 4 and 46.)

KHARE, R. (Jul. 1999). W* effect considered harmful. *IEEE Internet Computing*, *3*(4), pp. 89–92.
(Cited on page 55.)

KNUDSEN, J. AND LI, S. (Apr. 2005). *Beginning J2ME: From Novice to Professional* (Apress), 3rd ed.
(Cited on page 142.)

KNUTH, D. E. (Sep. 1997a). *Fundamental Algorithms*, *The Art of Computer Programming*, vol. 1 (Addison-Wesley, Boston, Massachusetts, USA), 3rd ed.
(Cited on page 94.)

KNUTH, D. E. (Sep. 1997b). *Seminumerical Algorithms*, *The Art of Computer Programming*, vol. 2 (Addison-Wesley, Boston, Massachusetts, USA), 3rd ed.
(Cited on pages 147 and 211.)

KOHLHOFF, C. AND STEELE, R. (Jul. 2004). Evaluating SOAP for high performance applications in capital markets. *Journal of Computer Systems, Science, and Engineering*, *19*(4), pp. 241–251.
(Cited on page 39.)

KOMU, M., TARKOMA, S., KANGASHARJU, J., AND GURTOV, A. (Sep. 2005). Applying a cryptographic namespace to applications. In *First International ACM Workshop on Dynamic Interconnection of Networks*.
URL http://doi.acm.org/10.1145/1080776.1080786
(Cited on page 6.)

KOSTOULAS, M. G., MATSA, M., MENDELSOHN, N., PERKINS, E., HEIFETS, A., AND MERCALDI, M. (May 2006). XML screamer: An integrated approach to high performance XML parsing, validation and deserialization. In L. Carr, D. De Roure, A. Iyengar,

C. A. Goble, and M. Dahlin (Eds.), *Proceedings of the 15th International World Wide Web Conference*. pp. 93–102.
(Cited on pages 41 and 183.)

LAUKKANEN, M. AND HELIN, H. (Jun. 2003). Web services in wireless networks — what happened to the performance? In L.-J. Zhang (Ed.), *Proceedings of the International Conference on Web Services*. pp. 278–284.
(Cited on page 60.)

LEVINSON, E. (Aug. 1998). *RFC 2387: The MIME Multipart/Related Content-type*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc2387.txt
(Cited on page 26.)

LEWIS, H. R. AND PAPADIMITRIOU, C. H. (1998). *Elements of the Theory of Computation* (Prentice Hall, Upper Saddle River, New Jersey, USA), 2nd ed.
(Cited on page 120.)

LIEFKE, H. AND SUCIU, D. (May 2000). XMill: an efficient compressor for XML data. In [Chen et al., 2000], pp. 153–164.
(Cited on pages 110 and 111.)

LINDHOLM, T. (Sep. 2003). XML three-way merge as a reconciliation engine for mobile data. In *Third ACM International Workshop on Data Engineering for Wireless and Mobile Access*. pp. 93–97.
URL http://www.hiit.fi/files/fi/fc/papers/mobide03-pc.pdf
(Cited on page 61.)

LINDHOLM, T. AND KANGASHARJU, J. (Apr. 2008). How to edit gigabyte XML files on a mobile phone with XAS, RefTrees, and RAXS. In *The Seventeenth World Wide Web Conference*. In submission.
URL http://www.hiit.fi/files/fi/fc/papers/lindholm-www08-submit.pdf
(Cited on pages 107 and 202.)

LINDHOLM, T., KANGASHARJU, J., AND TARKOMA, S. (Jun. 2005). A hybrid approach to optimistic file system directory tree syn-

chronization. In V. Kumar, A. B. Zaslavsky, U. Çetintemel, and A. Labrinidis (Eds.), *Fourth International ACM Workshop on Data Engineering for Wireless and Mobile Access*.
URL http://doi.acm.org/10.1145/1065870.1065879
(Cited on pages 6 and 90.)

LINDHOLM, T., KANGASHARJU, J., AND TARKOMA, S. (Oct. 2006). Fast and simple XML tree differencing by sequence alignment. In D. F. Brailsford (Ed.), *ACM Symposium on Document Engineering*. pp. 75–84.
URL http://www.hiit.fi/files/fi/fc/papers/doceng06-pc.pdf
(Cited on page 181.)

LINDHOLM, T. AND YELLIN, F. (Apr. 1999). *The Java Virtual Machine Specification* (Addison-Wesley, Boston, Massachusetts, USA), 2nd ed.
(Cited on page 145.)

LISKOV, B. AND SHRIRA, L. (Jun. 1988). Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In R. L. Wexelblat (Ed.), *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 260–267.
(Cited on page 56.)

LU, W., CHIU, K., AND PAN, Y. (Sep. 2006). A parallel approach to XML parsing. In *7th IEEE/ACM International Conference on Grid Computing*. pp. 223–230.
(Cited on page 42.)

LU, W., CHIU, K., SLOMINSKI, A., AND GANNON, D. (Jul. 2005). A streaming validation model for SOAP digital signature. In *14th IEEE Symposium on High Performance Distributed Computing*. pp. 243–252.
(Cited on pages 43, 101, and 202.)

MAKINO, S., TATSUBORI, M., TAMURA, K., AND NAKAMURA, Y. (Jul. 2005). Improving WS-Security performance with a template-based approach. In *IEEE International Conference on*

Web Services (Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, USA), pp. 588–595.
(Cited on page 43.)

MASCOLO, C., CAPRA, L., ZACHARIADIS, S., AND EMMERICH, W. (Apr. 2002). XMIDDLE: A data-sharing middleware for mobile computing. *Personal and Wireless Communications*, *21*(1), pp. 77–103.
(Cited on page 61.)

MEGIDDO, N. AND MOCHA, D. S. (Mar. 2003). ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*.
(Cited on page 124.)

MEYER, M. (Sep. 1999). TCP performance over GPRS. In *IEEE Wireless Communications and Networking Conference*. pp. 1248–1252.
(Cited on page 53.)

MIKKONEN, T. (Feb. 2007). *Programming Mobile Devices: An Introduction for Practitioners* (John Wiley & Sons, Hoboken, New Jersey, USA).
(Cited on page 48.)

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. (1997). *The Definition of Standard ML (Revised)* (MIT Press, Cambridge, Massachusetts, USA).
(Cited on page 127.)

MOSKOWITZ, R. AND NIKANDER, P. (May 2006). *RFC 4423: Host Identity Protocol (HIP) Architecture*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc4423.txt
(Cited on pages 30 and 52.)

MURATA, M., LEE, D., MANI, M., AND KAWAGUCHI, K. (Nov. 2005). Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, *5*(4), pp. 660–704.
(Cited on pages 17 and 18.)

NIEDERMEIER, U., HEUER, J., HUTTER, A., STECHELE, W., AND KAUP, A. (Aug. 2002). An MPEG-7 tool for compression and streaming of XML data. In *IEEE International Conference on Multimedia and Expo*. pp. 521–524.
(Cited on pages 116 and 117.)

NIKANDER, P., WALL, J., AND YLITALO, J. (Feb. 2003). Integrating security, mobility, and multi-homing in a HIP way. In *Proceedings of Network and Distributed Systems Security Symposium* (Internet Society), pp. 87–99.
URL http://www.tcm.hut.fi/~pnr/publications/NDSS03-Nikander-et-al.pdf
(Cited on page 52.)

OASIS (Dec. 2001). *RELAX NG Specification*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA.
URL http://www.relaxng.org/spec-20011203.html
(Cited on pages 18 and 131.)

OASIS (Apr. 2002a). *Message Service Specification, Version 2.0*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA.
URL http://www.oasis-open.org/committees/ebxml-msg/documents/ebMS_v2_0.pdf
(Cited on page 27.)

OASIS (Nov. 2002b). *RELAX NG Compact Syntax*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA.
URL http://www.relaxng.org/compact-20021121.html
(Cited on pages 18 and 127.)

OASIS (Mar. 2003). *SOAP Message Security: Minimalist Profile (MProf)*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA. Working Draft.
URL http://xml.coverpages.org/WSS-MinimalistProfile-20030307.pdf
(Cited on page 43.)

OASIS (Aug. 2004a). *Web Services Reliable Messaging: WS-Reliability 1.1*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA.
URL http://docs.oasis-open.org/wsrm/2004/06/WS-Reliability-CD1.086.pdf
(Cited on page 28.)

OASIS (Mar. 2004b). *Web Services Security: SOAP Message Security 1.0*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA.
URL http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0
(Cited on page 28.)

OASIS (Feb. 2006a). *Web Services Security: SOAP Message Security 1.1*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA.
URL http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf
(Cited on page 37.)

OASIS (Feb. 2006b). *Web Services Security UsernameToken Profile 1.1*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA.
URL http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-UsernameTokenProfile.pdf
(Cited on page 37.)

OASIS (Feb. 2006c). *Web Services Security X.509 Certificate Token Profile 1.1*. Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA.
URL http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-x509TokenProfile.pdf
(Cited on page 37.)

OKASAKI, C. (Jul. 1999). *Purely Functional Data Structures* (Cambridge University Press).
(Cited on page 92.)

O'MAHONY, D. (Jan. 1998). UMTS: the fusion of fixed and mobile networking. *IEEE Internet Computing*, *2*(1), pp. 49–56.
(Cited on page 48.)

OMG (Mar. 2004). *Common Object Request Broker Architecture (CORBA/IIOP), version 3.0.3*. Object Management Group, Needham, Massachusetts, USA.
(Cited on pages 22, 39, 57, and 72.)

OMG (May 2005). *Wireless Access and Terminal Mobility in CORBA, Version 1.2*. Object Management Group, Needham, Massachusetts, USA.
(Cited on page 57.)

O'TUATHAIL, E. AND ROSE, M. T. (Jun. 2002). *RFC 3288: Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP)*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc3288.txt
(Cited on page 78.)

PARADISO, J. A. AND STARNER, T. (Jan. 2005). Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, *4*(1), pp. 18–27.
(Cited on page 46.)

PERICAS-GEERTSEN, S. (Dec. 2003). Binary interchange of XML Infosets. In *XML Conference and Exposition*.
(Cited on page 113.)

PERKINS, C. E. (Ed.) (2001). *Ad Hoc Networking* (Addison-Wesley, Boston, Massachusetts, USA).
(Cited on pages 51 and 55.)

PERKINS, C. E. (Aug. 2002). *RFC 3344: IP Mobility Support for IPv4*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc3344.txt
(Cited on page 52.)

PERKINS, C. E. AND ROYER, E. M. (Feb. 1999). Ad-hoc on-demand distance vector routing. In *Second IEEE Workshop on Mobile Computing Systems and Applications*. pp. 90–100.
(Cited on page 56.)

PERRIN, T. (Aug. 2003). Public key distribution through "cryptoIDs". In *New Security Paradigms Workshop*. pp. 87–102.
(Cited on page 30.)

PORCINO, D. AND HIRT, W. (Jul. 2003). Ultra-wideband radio technology: Potential and challenges ahead. *IEEE Communications Magazine*, *41*(7), pp. 66–74.
(Cited on page 50.)

POSTEL, J. (Aug. 1980). *RFC 768: User Datagram Protocol*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc768.txt
(Cited on page 52.)

POSTEL, J. (Sep. 1981a). *RFC 791: Internet Protocol*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc791.txt
(Cited on page 51.)

POSTEL, J. (Sep. 1981b). *RFC 793: Transmission Control Protocol*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc793.txt
(Cited on page 52.)

POTLAPALLY, N. R., RAVI, S., RAGHUNATHAN, A., AND JHA, N. K. (Feb. 2006). A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *IEEE Transactions on Mobile Computing*, *5*(2), pp. 128–143.
(Cited on page 47.)

QURESHI, M. H. AND SAMADZADEH, M. H. (Apr. 2005). Determining the complexity of XML documents. In *International Conference on Information Technology: Coding and Computing*. pp. 416–421.
(Cited on page 184.)

RAATIKAINEN, K., CHRISTENSEN, H. B., AND NAKAJIMA, T. (Oct. 2002). Application requirements for middleware for mobile and pervasive systems. *Mobile Computing and Communications Review*, *6*(4), pp. 16–24.
(Cited on pages 4 and 61.)

RAMYA, T. B. (Aug. 2005). *A Gateway for SIP and Generic Publish/-Subscribe Events Interworking*. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engi-

neering, Espoo, Finland.
(Cited on page 6.)

RIVA, O. (Nov. 2006). Contory: A middleware for the provisioning of context information on smart phones. In M. van Steen and M. Henning (Eds.), *ACM/IFIP/USENIX 7th International Middleware Conference* (Springer-Verlag, Heidelberg, Germany), *Lecture Notes in Computer Science*, vol. 4290, pp. 219–239.
(Cited on pages 9 and 50.)

RIVA, O. AND KANGASHARJU, J. (2007). Challenges and lessons in developing middleware on smart phones. *IEEE Computer*. In submission.
URL http://www.cs.helsinki.fi/u/riva/publications/riva_ieeecomputer07.pdf
(Cited on page 63.)

RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. (Feb. 1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*(2), pp. 120–126.
(Cited on page 29.)

ROBERTS, L. G. (Nov. 1978). The evolution of packet switching. *Proceedings of the IEEE*, *66*(11), pp. 1307–1313.
(Cited on page 48.)

ROSE, M. T. (Mar. 2001). *RFC 3080: The Blocks Extensible Exchange Protocol Core*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc3080.txt
(Cited on page 77.)

ROUBTSOV, V. (Aug. 2002). Java tip 130: Do you know your data size?
URL http://www.javaworld.com/javaworld/javatips/jw-javatip130.html
(Cited on page 170.)

SALTZER, J. H. (Aug. 1993). *RFC 1498: On the Naming and Binding of Network Destinations*. Internet Engineering Task Force.
URL http://www.ietf.org/rfc/rfc1498.txt
(Cited on page 52.)

SALTZER, J. H., REED, D. P., AND CLARK, D. D. (Nov. 1984). End-to-end arguments in system design. *ACM Transactions on Computer Systems*, *2*(4), pp. 277–288.
(Cited on pages 48 and 53.)

SANDOZ, P., PERICAS-GEERTSEN, S., KAWAGUCHI, K., HADLEY, M., AND PELEGRI-LLOPART, E. (Aug. 2003). Fast Web services. On Sun Developer Network.
URL http://developer.java.sun.com/developer/technicalArticles/WebServices/fastWS/index.html
(Cited on pages 116 and 117.)

SANDOZ, P., TRIGLIA, A., AND PERICAS-GEERTSEN, S. (Jun. 2004). Fast Infoset. On Sun Developer Network.
URL http://java.sun.com/developer/technicalArticles/xml/fastinfoset/
(Cited on page 114.)

SATYANARAYANAN, M. (Aug. 2001). Pervasive computing: Vision and challenges. *IEEE Personal Communications*, *8*(4), pp. 10–17.
(Cited on page 55.)

SCHNEIDER, J. (Sep. 2003). Theory, benefits and requirements for efficient encoding of XML documents. In [W3C, 2003b].
URL http://www.agiledelta.com/EfficientXMLEncoding.htm
(Cited on page 120.)

SCHNEIER, B. (1990). *Applied Cryptography* (John Wiley & Sons, Hoboken, New Jersey, USA), 2nd ed.
(Cited on page 105.)

SERIN, E. (Mar. 2003). *Design and Test of the Cross-Format Schema Protocol (XFSP) for Networked Virtual Environments*. Master's thesis, Naval Postgraduate School, Monterey, California, USA.
(Cited on pages 116 and 120.)

SHIRASUNA, S., SLOMINSKI, A., FANG, L., AND GANNON, D. (Nov. 2004). Performance comparison of security mechanisms for Grid services. In R. Buyya (Ed.), *5th IEEE/ACM International Workshop on Grid Computing*. pp. 360–364.
(Cited on pages 43 and 102.)

SLOMINSKI, A. (Mar. 2004). On using XML pull parsing Java APIs. On XmlPull Web site.
URL http://www.xmlpull.org/history/index.html
(Cited on pages 22 and 121.)

SM (Nov. 2002). *JSR 118: Mobile Information Device Profile Version 2.0*. Sun Microsystems Inc. and Motorola Inc.
URL http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html
(Cited on page 47.)

SOSNOSKI, D. M. (Sep. 2003a). XBIS XML Infoset encoding. In [W3C, 2003b].
URL http://www.w3.org/2003/08/binary-interchange-workshop/09-Sosnoski-position-paper.pdf
(Cited on page 114.)

SOSNOSKI, D. M. (Jan. 2003b). XML and Java technologies: Data binding, part 1: Code generation. On IBM DeveloperWorks.
URL http://www-128.ibm.com/developerworks/xml/library/x-databdopt/
(Cited on page 23.)

SPERBERG-MCQUEEN, C. M. (Oct. 2005). XML and semi-structured data. *ACM Queue*, *3*(8), pp. 34–41.
(Cited on page 21.)

STEVENS, W. R. (1997). *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI* (Prentice Hall, Upper Saddle River, New Jersey, USA), 2nd ed.
(Cited on page 56.)

Sun (Mar. 2001). *JSR 36: Connected Device Configuration (CDC)*. Sun Microsystems Inc., Santa Clara, California, USA.
URL http://jcp.org/aboutJava/communityprocess/final/jsr036
(Cited on page 47.)

Sun (Aug. 2002). *JSR 46: Foundation Profile*. Sun Microsystems Inc., Santa Clara, California, USA.
URL http://jcp.org/aboutJava/communityprocess/final/

jsr046
(Cited on page 47.)

Sun (Mar. 2003a). *JSR 139: Connected Limited Device Configuration 1.1*. Sun Microsystems Inc., Santa Clara, California, USA.
URL http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html
(Cited on page 47.)

Sun (Jan. 2003b). *JSR 31: Java Architecture for XML Binding (JAXB)*. Sun Microsystems Inc., Santa Clara, California, USA.
URL http://jcp.org/aboutJava/communityprocess/final/jsr031/index.html
(Cited on page 60.)

Sun (2004). *Java Remote Method Invocation Specification*. Sun Microsystems Inc., Santa Clara, California, USA.
URL ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf
(Cited on page 58.)

SUNDARESAN, N. AND MOUSSA, R. (May 2001). Algorithms and programming models for efficient representation of XML for Internet applications. In *Tenth International World Wide Web Conference*. pp. 366–375.
URL http://www10.org/cdrom/papers/542/index.html
(Cited on page 116.)

TAKASE, T., MIYASHITA, H., SUZUMURA, T., AND TATSUBORI, M. (May 2005). An adaptive, fast, and safe XML parser based on byte sequences memorization. In A. Ellis and T. Hagino (Eds.), *Proceedings of the 14th International World Wide Web Conference*. pp. 692–701.
(Cited on pages 41, 42, and 43.)

TARKOMA, S. (Apr. 2006). *Efficient Content-based Routing, Mobility-aware Topologies, and Temporal Subspace Matching*. Ph.D. thesis, University of Helsinki, Department of Computer Science, Helsinki, Finland.
URL http://ethesis.helsinki.fi/julkaisut/mat/tieto/vk/tarkoma/
(Cited on page 6.)

TARKOMA, S. AND KANGASHARJU, J. (Sep. 2006). Optimizing content-based routers: Posets and forests. *Distributed Computing*, *19*(1), pp. 62–77.
URL http://dx.doi.org/10.1007/s00446-006-0002-0
(Cited on page 206.)

TARKOMA, S. AND KANGASHARJU, J. (Apr. 2007). On the cost and safety of handoffs in content-based routing systems. *Computer Networks*, *51*(6), pp. 1459–1482.
URL http://dx.doi.org/10.1016/j.comnet.2006.07.016
(Cited on page 59.)

TARKOMA, S., KANGASHARJU, J., LINDHOLM, T., AND RAATIKAINEN, K. (Sep. 2006). Fuego: Experiences with mobile data communication and synchronization. In *17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*.
URL http://dx.doi.org/10.1109/PIMRC.2006.254072
(Cited on page 6.)

TIAN, M., VOIGT, T., NAUMOWICZ, T., RITTER, H., AND SCHILLER, J. (Jul. 2004). Performance considerations for mobile web services. *Computer Communications*, *27*(11), pp. 1097–1105.
(Cited on page 60.)

UNICODE CONSORTIUM (Aug. 2003). *The Unicode Standard, Version 4.0* (Addison-Wesley, Boston, Massachusetts, USA).
(Cited on page 14.)

VACIRCA, F., RICCIATO, F., AND PILZ, R. (Jul. 2005). Large-scale RTT measurements from an operational UMTS/GPRS network. In *First International Conference on Wireless Internet*. pp. 190–197.
(Cited on page 53.)

VAN DER VLIST, E. (Dec. 2003). *RELAX NG* (O'Reilly, Sebastopol, California, USA).
(Cited on page 20.)

VAN ENGELEN, R. A. (Jul. 2004). Constructing finite state automata for high performance Web services. In [IEEE, 2004a].
(Cited on page 41.)

VOGELS, W. (Nov. 2003). Web services are not distributed objects. *IEEE Internet Computing*, *7*(6), pp. 59–66.
(Cited on page 60.)

W3C (Feb. 1998). *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/1998/REC-xml-19980210
(Cited on page 14.)

W3C (Jun. 1999). *WAP Binary XML Content Format*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/1999/06/NOTE-wbxml-19990624
(Cited on pages 54, 112, and 113.)

W3C (May 2000a). *Simple Object Access Protocol (SOAP) 1.1*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
(Cited on page 24.)

W3C (Dec. 2000b). *SOAP Messages with Attachments*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211
(Cited on page 26.)

W3C (Mar. 2001). *Canonical XML Version 1.0*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2001/REC-xml-c14n-20010315
(Cited on pages 21, 33, 35, and 92.)

W3C (Dec. 2002a). *Decryption Transform for XML Signature*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2002/REC-xmlenc-decrypt-20021210
(Cited on page 33.)

W3C (Jul. 2002b). *Exclusive XML Canonicalization Version 1.0*. World Wide Web Consortium, Cambridge, Massachusetts,

USA. W3C Recommendation.
URL http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/
(Cited on pages 33 and 35.)

W3C (Jun. 2002c). *SOAP Version 1.2 Email Binding*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/TR/2002/NOTE-soap12-email-20020626
(Cited on page 25.)

W3C (Aug. 2002d). *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2002/REC-xhtml1-20020801
(Cited on page 54.)

W3C (Dec. 2002e). *XML Encryption Syntax and Processing*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/
(Cited on page 33.)

W3C (Feb. 2002f). *XML Signature Syntax and Processing*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/
(Cited on page 33.)

W3C (Jan. 2003a). *Scalable Vector Graphics (SVG) 1.1 Specification*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2003/REC-SVG11-20030114/
(Cited on page 186.)

W3C (Sep. 2003b). *W3C Workshop on Binary Interchange of XML Information Item Sets* (World Wide Web Consortium).
URL http://www.w3.org/2003/08/binary-interchange-workshop/Report.html
(Cited on pages 8, 118, 233, 243, 244, and 254.)

W3C (Dec. 2004a). *Architecture of the World Wide Web, Volume One*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2004/REC-webarch-20041215/
(Cited on pages 16 and 27.)

W3C (Apr. 2004b). *Document Object Model (DOM) Level 3 Core Specification*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/
(Cited on page 22.)

W3C (Jun. 2004c). *SOAP 1.2 Attachment Feature*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/TR/2004/NOTE-soap12-af-20040608/
(Cited on page 26.)

W3C (Aug. 2004d). *Web Services Addressing (WS-Addressing)*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Member Submission.
URL http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/
(Cited on page 27.)

W3C (Feb. 2004e). *XML Information Set*. World Wide Web Consortium, Cambridge, Massachusetts, USA, 2nd ed. W3C Recommendation.
URL http://www.w3.org/TR/2004/REC-xml-infoset-20040204/
(Cited on page 20.)

W3C (Oct. 2004f). *XML Schema Part 1: Structures*. World Wide Web Consortium, Cambridge, Massachusetts, USA, 2nd ed. W3C Recommendation.
URL http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/
(Cited on pages 17 and 124.)

W3C (Oct. 2004g). *XML Schema Part 2: Datatypes*. World Wide Web Consortium, Cambridge, Massachusetts, USA, 2nd ed.

W3C Recommendation.
URL http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/
(Cited on pages 17 and 99.)

W3C (May 2005a). *Describing Media Content of Binary Data in XML*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/TR/2005/NOTE-xml-media-types-20050504
(Cited on page 26.)

W3C (Jan. 2005b). *SOAP Message Transmission Optimization Mechanism*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2004/REC-soap12-mtom-20050125/
(Cited on page 26.)

W3C (Mar. 2005c). *XML Binary Characterization*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/TR/2005/NOTE-xbc-characterization-20050331/
(Cited on pages 118, 177, and 178.)

W3C (Mar. 2005d). *XML Binary Characterization Measurement Methodologies*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/TR/2005/NOTE-xbc-measurement-20050331/
(Cited on pages 118 and 178.)

W3C (Mar. 2005e). *XML Binary Characterization Properties*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Note.
URL http://www.w3.org/TR/2005/NOTE-xbc-properties-20050331/
(Cited on pages 20, 118, 177, 178, 191, and 196.)

W3C (Mar. 2005f). *XML Binary Characterization Use Cases*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C

Note.
URL http://www.w3.org/TR/2005/NOTE-xbc-use-cases-20050331/
(Cited on pages 118 and 184.)

W3C (Jan. 2005g). *XML-binary Optimized Packaging*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2004/REC-xop10-20050125/
(Cited on page 26.)

W3C (Aug. 2006a). *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, Cambridge, Massachusetts, USA, 4th ed. W3C Recommendation.
URL http://www.w3.org/TR/2004/REC-xml-20060816/
(Cited on pages 3, 13, 14, 17, and 91.)

W3C (Aug. 2006b). *Extensible Markup Language (XML) 1.1*. World Wide Web Consortium, Cambridge, Massachusetts, USA, 2nd ed. W3C Recommendation.
URL http://www.w3.org/TR/2006/REC-xml11-20060816/
(Cited on page 14.)

W3C (Aug. 2006c). *Namespaces in XML*. World Wide Web Consortium, Cambridge, Massachusetts, USA, 2nd ed. W3C Recommendation.
URL http://www.w3.org/TR/2006/REC-xml-names-20060816/
(Cited on page 15.)

W3C (May 2006d). *Web Services Addressing 1.0 — Core*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/
(Cited on pages 27 and 75.)

W3C (May 2006e). *Web Services Addressing 1.0 — SOAP Binding*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509/
(Cited on page 27.)

W3C (Jul. 2007a). *Efficient XML Interchange (EXI) Format 1.0*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Working Draft.
URL http://www.w3.org/TR/2007/WD-exi-20070716/
(Cited on pages 120, 204, and 205.)

W3C (Jul. 2007b). *Efficient XML Interchange Measurements Note*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Working Draft.
URL http://www.w3.org/TR/2007/WD-exi-measurements-20070725/
(Cited on pages 8, 178, 184, and 188.)

W3C (Apr. 2007c). *SOAP Version 1.2 Part 1: Messaging Framework*. World Wide Web Consortium, Cambridge, Massachusetts, USA, 2nd ed. W3C Recommendation.
URL http://www.w3.org/TR/2007/REC-soap12-part1-20070427/
(Cited on page 24.)

W3C (Apr. 2007d). *SOAP Version 1.2 Part 2: Adjuncts*. World Wide Web Consortium, Cambridge, Massachusetts, USA, 2nd ed. W3C Recommendation.
URL http://www.w3.org/TR/2007/REC-soap12-part2-20070427/
(Cited on page 24.)

W3C (Jun. 2007e). *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2007/REC-wsdl20-20070626/
(Cited on page 44.)

W3C (Jan. 2007f). *XML Path Language (XPath) 2.0*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2007/REC-xpath20-20070123/
(Cited on page 20.)

W3C (Jan. 2007g). *XQuery 1.0: An XML Query Language*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C

Recommendation.
URL http://www.w3.org/TR/2007/REC-xquery-20070123/
(Cited on page 20.)

W3C (Jan. 2007h). *XQuery 1.0 and XPath 2.0 Data Model (XDM)*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/
(Cited on pages 20 and 179.)

W3C (Jan. 2007i). *XSL Transformations (XSLT) Version 2.0*. World Wide Web Consortium, Cambridge, Massachusetts, USA. W3C Recommendation.
URL http://www.w3.org/TR/2007/REC-xslt20-20070123/
(Cited on page 20.)

WALDO, J., WYANT, G., WOLLRATH, A., AND KENDALL, S. (Nov. 1994). A note on distributed computing. Tech. Rep. TR-94-29, Sun Microsystems Inc., Santa Clara, California, USA.
URL http://research.sun.com/techrep/1994/abstract-29.html
(Cited on page 60.)

WALL, T. AND CAHILL, V. (Sep. 2001). Mobile RMI: Supporting remote access to Java server objects on mobile hosts. In *International Symposium on Distributed Objects and Applications*. pp. 41–51.
(Cited on page 58.)

WAP (2001a). *Wireless Application Protocol: Architecture Specification*. WAP Forum.
(Cited on page 54.)

WAP (Sep. 2001b). *Wireless Markup Language Version 2.0*. WAP Forum.
(Cited on page 54.)

WAP (2001c). *Wireless Transport Layer Security Specification*. WAP Forum.
(Cited on page 54.)

WATSON, R. W. AND MAMRAK, S. A. (May 1987). Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, *5*(2), pp. 97–120.
(Cited on page 42.)

WEISER, M. (Jul. 1993). Some computer science issues in ubiquitous computing. *Communications of the ACM*, *36*(7), pp. 75–84.
(Cited on page 55.)

WERNER, C., BUSCHMANN, C., BRANDT, Y., AND FISCHER, S. (Sep. 2006). Compressing SOAP messages by using pushdown automata. In [IEEE, 2006], pp. 19–26.
(Cited on pages 116 and 118.)

WERNER, C., BUSCHMANN, C., AND FISCHER, S. (Jul. 2004). Compressing SOAP messages by using differential encoding. In [IEEE, 2004a], pp. 540–547.
(Cited on page 112.)

WHEELER, D. A. (Jul. 2002). More than a gigabuck: Estimating GNU/Linux's size.
URL http://www.dwheeler.com/sloc/
(Cited on page 82.)

WILLIAMS, S. D. (Sep. 2003). Efficiency structured XML (esXML/esDOM): A standard binary infoset and API. In [W3C, 2003b].
URL http://www.esxml.org/w3cout/W3C_esXMLPres_big.html
(Cited on page 120.)

WILLIAMS, S. D. (Sep. 2007). Efficient XML Interchange (EXI) with XML Signature and Encryption: Reuse and opportunity. In *W3C Workshop on Next Steps for XML Signature and Encryption*.
URL http://www.w3.org/2007/xmlsec/ws/papers/13-williams
(Cited on page 202.)

WINER, D. (Jun. 2003). *XML-RPC Specification*.
URL http://www.xmlrpc.com/spec
(Cited on page 24.)

YAU, S. S. AND KARIM, F. (Jan. 2004). An adaptive middleware for context-sensitive communications for real-time applications in ubiquitous computing environments. *Real-Time Systems*, *26*(1), pp. 29–61.
(Cited on page 61.)

YOUSSEF, M., YOUSSEF, A., RIEGER, C., SHANKAR, U., AND AGRAWALA, A. (Jun. 2006). PinPoint: An asynchronous time-based location determination system. In P. Gunningberg, L.-Å. Larzon, M. Satyanarayanan, and N. Davies (Eds.), *Proceedings of MobiSys '06: 4th international conference on Mobile systems, applications, and services*. pp. 165–176.
(Cited on page 85.)

ZIV, J. AND LEMPEL, A. (May 1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, *23*(3), pp. 337–343.
(Cited on page 110.)

TIETOJENKÄSITTELYTIETEEN LAITOS
PL 68 (Gustaf Hällströmin katu 2 b)
00014 Helsingin yliopisto

DEPARTMENT OF COMPUTER SCIENCE
P.O. Box 68 (Gustaf Hällströmin katu 2 b)
FIN-00014 University of Helsinki, FINLAND

JULKAISUSARJA **A**

SERIES OF PUBLICATIONS **A**

Reports may be ordered from: Kumpula Science Library, P.O. Box 64, FIN-00014 University of Helsinki, FINLAND.

A-2004-2 A. Gurtov: Efficient data transport in wireless overlay networks. 141 pp. (Ph.D. thesis)

A-2004-3 K. Vasko: Computational methods and models for paleoecology. 176 pp. (Ph.D. thesis)

A-2004-4 P. Sevon: Algorithms for Association-Based Gene Mapping. 101 pp. (Ph.D. thesis)

A-2004-5 J. Viljamaa: Applying Formal Concept Analysis to Extract Framework Reuse Interface Specifications from Source Code. 206 pp. (Ph.D. thesis)

A-2004-6 J. Ravantti: Computational Methods for Reconstructing Macromolecular Complexes from Cryo-Electron Microscopy Images. 100 pp. (Ph.D. thesis)

A-2004-7 M. Kääriäinen: Learning Small Trees and Graphs that Generalize. 45+49 pp. (Ph.D. thesis)

A-2004-8 T. Kivioja: Computational Tools for a Novel Transcriptional Profiling Method. 98 pp. (Ph.D. thesis)

A-2004-9 H. Tamm: On Minimality and Size Reduction of One-Tape and Multitape Finite Automata. 80 pp. (Ph.D. thesis)

A-2005-1 T. Mielikäinen: Summarization Techniques for Pattern Collections in Data Mining. 201 pp. (Ph.D. thesis)

A-2005-2 A. Doucet: Advanced Document Description, a Sequential Approach. 161 pp. (Ph.D. thesis)

A-2006-1 A. Viljamaa: Specifying Reuse Interfaces for Task-Oriented Framework Specialization. 285 pp. (Ph.D. thesis)

A-2006-2 S. Tarkoma: Efficient Content-based Routing, Mobility-aware Topologies, and Temporal Subspace Matching. 198 pp. (Ph.D. thesis)

A-2006-3 M. Lehtonen: Indexing Heterogeneous XML for Full-Text Search. 185+3 pp. (Ph.D. thesis).

A-2006-4 A. Rantanen: Algorithms for ^{13}C Metabolic Flux Analysis. 92+73 pp.(Ph.D. thesis).

A-2006-5 E. Terzi: Problems and Algorithms for Sequence Segmentations. 141 pp. (Ph.D. Thesis).

A-2007-1 P. Sarolahti: TCP Performance in Heterogeneous Wireless Networks.(Ph.D. Thesis).

A-2007-2 M. Raento: TCP Exploring privacy for ubiquitous computing: Tools, methods and experiments. (Ph.D. thesis).

A-2007-3 L. Aunimo: Methods for Answer Extraction in Textual Question Answering 127+18 pp. (Ph.D. Thesis).

A-2007-4 T. Roos: Statistical and Information-Theoretic Methods for Data Analysis. 82+75 pp. (Ph.D. Thesis).

A-2007-5 S. Leggio: A Decentralized Session Management Framework for Heterogeneous Ad-Hoc and Fixed Networks. 230 pp. (Ph.D. Thesis).

A-2007-6 O. Riva: Middleware for Mobile Sensing Applications in Urban Environments. 195 pp. (Ph.D. thesis).

A-2007-7 K. Palin: Computational Methods for Locating and Analyzing Conserved Gene Regulatory DNA Elements. 130 pp. (Ph.D. Thesis)

A-2008-1 I. Autio: Modeling Efficient Classification as a Process of Confidence Assessment and Delegation. 212 pp. (Ph.D. Thesis)