An XML Messaging Service for Mobile Devices

Jaakko Kangasharju

Helsinki, February 4, 2006
Licentiate Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Acknowledgments

First of all, I would like to thank the advisor of my postgraduate studies, Professor Kimmo Raatikainen, for the opportunity to work on this topic. He has permitted me the freedom to pursue my own interests, but has always been available to advise and has provided many pointers on interesting avenues to consider.

The Fuego Core project, where the work for this thesis was performed, is an excellent environment for research. The atmosphere in the project is very relaxed, and all of its past and present members very competent. Discussions within the group have been very stimulating for my own work, and I hope I have contributed similarly to others' work.

As I have noticed during this work, a middleware platform cannot exist in a vacuum. Design of the system and its interfaces needs to be driven by the needs of messaging applications, and these needs cannot all be understood in advance. In that spirit, I would like to thank Sasu Tarkoma and Marko Saaresto for early use of the messaging system and for discovering several issues, Tancred Lindholm for using the XAS API and prompting generalization of many initially-specific parts, and Oriana Riva, whose needs in data transmission were the reason for designing the Object Representation Language described in section 4.5.

Finally, I would like to thank Dr. Jussi Kangasharju and Sasu Tarkoma for reading a draft version of this thesis. Their comments were very helpful in preparing the final version. Any omissions, unclarities, or mistakes that remain are, naturally, my responsibility.

Contents

List of Figures

List of Tables

List of Abbreviations

AMME	Abstract Mobile Message Exchange
API	Application Programming Interface
ARC	Adaptive Replacement Cache
ASN.1	Abstract Syntax Notation One
BEEP	Blocks Extensible Exchange Protocol
COA	Codec Omission Automaton
CORBA	Common Object Request Broker Architecture
DOA	Decoding Omission Automaton
DOM	Document Object Model
DTD	Document Type Definition
EOA	Encoding Omission Automaton
EXI	Efficient XML Interchange
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
GUI	Graphical User Interface
HIP	Host Identity Protocol
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
JIT	just-in-time
JVM	Java Virtual Machine
LAN	Local Area Network

LRU	Least Recently Used
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
MTP	Message Transfer Protocol
MTS	Message Transfer Service
NAT	Network Address Translation
OASIS	Organization for the Advancement of Structured Information Standards
ORL	Object Representation Language
PDA	Personal Digital Assistant
PER	Packed Encoding Rules
PPM	Prediction by Partial Matching
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
StAX	Streaming API for XML
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery, and Integration
UMTS	Universal Mobile Telecommunications System

URI	Universal Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAN	Wide Area Network
WAP	Wireless Application Protocol
WBXML	WAP Binary XML
WG	Working Group
WLAN	Wireless LAN
WS-I	Web Services Interoperability Organization
WSDL	Web Services Description Language
WWW	World Wide Web
XBC	XML Binary Characterization
XFSP	Cross-Format Schema Protocol
XML	Extensible Markup Language
XOP	XML-binary Optimized Packaging
XSBC	XML Schema-based Binary Compression

Chapter 1

Introduction

Current trends indicate that computing in the future will be radically different from what it is today. The significant revolution is driven by miniaturization of computing devices, which makes it possible to include computing capabilities in more and more devices as well as for people to carry considerable computing capabilities with them at all times.

This new environment, with computing capabilities available everywhere, often vanishing into the background, is an active research topic, popularly called *ubiquitous* [99] or *pervasive* [80] computing. Our research is concentrated on the support layers for new applications, built on personal mobile devices, and therefore we use the term *mobile environment* for this future computing scenario.

Whatever the applications of the future will be, they will be highly interconnected, with a need to communicate both with other devices and with available network infrastructure services. A system providing an integrated interface to such communication capabilities and auxiliary services is typically called *middleware* [1], and a general-purpose middleware platform is a powerful tool for distributed application development.

Existing deployed middleware platforms are typically based on one of two paradigms. *Distributed objects*, exemplified by Common Object Request Broker Architecture (CORBA) [64], provide object-oriented interfaces to clients, with communication happening by invoking methods over the network. *Message-oriented middleware*, like IBM's MQSeries [36], provides a more loosely-coupled interface. Here the middleware does not impose any syntax on messages, but only provides addressing and delivery. However, existing middleware is typically designed for fixed networks, even Local Area Networks (LANs), and is not suitable for the mobile environment.

For the new environment a new approach to building systems will be needed. To provide an easy way to build applications, it is fruitful to start this work with a middleware platform. Since communication is a fundamental component of middleware, we will focus on providing a mes-

sage transfer service to be used by other components of the middleware and by applications. Furthermore, we will concentrate solely on point-to-point communication and leave concerns such as application-level routing to users of the service.

The internals of the message transfer service are dependent on two main issues: the protocol to be used for communication and the syntax of messages that are sent. As the message transfer service needs to provide a general messaging capability, it need not provide any semantics for messages. Externally, a design point will also be the interface that is provided to messaging applications.

A common design for application-layer protocols on the Internet [52, 24] has been to use plain text as the communication syntax. This is seen as beneficial for simplicity of implementation and for ease of debugging. However, Internet protocols typically do not have a unified syntax for their messages, each adapting some common principles to its own purposes.

In recent years, a common text-based syntax for interoperable data has emerged in XML [119]. XML provides a standard way to represent structured data in a tree format, and it has been intentionally designed to be simple to implement. An abundance of implementations and technologies related to processing XML in various manners is a testament to the success of this design. As a standard way to represent structured data, XML would appear to be ideal to select as the message syntax.

However, XML is not obviously suitable for the future computing environment of small weak devices and expensive wireless communication. XML is a very verbose format and its text-based nature makes it more expensive to process than previous binary formats. Furthermore, the protocols typically used for XML messaging are designed for fixed network use, so wireless networks may bring out latent inefficiencies. A prominent example where the design of an application-layer protocol interacts badly with TCP is provided by Java RMI [13].

In spite of XML's apparent unsuitability, the trend in fixed networks is clearly towards XML messaging. We believe it to be important for mobile devices to participate equally in the full networked infrastructure, so in our view it is important to select the same technologies for both fixed and mobile networking. Therefore we have investigated the issues that XML has, and have attempted to come up with solutions.

Our solution, presented in this work, is a Message Transfer Service based on XML messaging. This service has been designed as a component of a larger middleware platform, and its requirements are driven by our analysis of the problem areas of XML messaging. We have implemented solutions for each of the identified problematic areas and consider our message transfer service to demonstrate that XML is a feasible selection as the message syntax.

We begin with an introduction to XML messaging and the mobile en-

vironment in chapter 2 where we also include a review of existing measurements of XML performance. Then, chapter 3 describes the architecture and interfaces of our message transfer service and gives an overview of the three key components. These components are the detailed topics of the next three chapters: chapter 4 shows our Application Programming Interface (API) for processing XML data, chapter 5 defines our XML serialization format, and chapter 6 presents our work in the protocol area. We present experiments comparing our solutions to more standard ones in chapter 7. Finally, chapter 8 concludes the thesis by listing the lessons we have learned and our planned future work.

Chapter 2

XML and the Mobile Environment

Extensible Markup Language (XML) [119] 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. Many of these are specifications of the World Wide Web Consortium (W3C), but due to the large interest in XML some of these are not even mature enough for standardization. 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 XML and the XML Stack

XML was originally born from the desire to streamline Standard Generalized Markup Language (SGML) [38] for use on the World Wide Web. For this purpose the designers set the following design goals (from [119]):

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.

```
<?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

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 [102] was completed in 1998. Currently XML version 1.0 is in its third edition [119], and there is also version 1.1 [120] to address Unicode [95] evolution and concerns about whitespace handling. However, due to XML 1.1 being 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. 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. Elements may contain other elements (like `name` here), *text* (`Wagner`), or *attributes* (`nationality`).

```
<?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

While XML did achieve its goal of simplicity, at least when compared with SGML, use on the heterogeneous World Wide Web (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.

A solution to this is provided by XML Namespaces [103]. This specification defines that Universal Resource Identifiers (URIs) function as ways to group related XML names together, thus separating unrelated names from each other. Then the complete name of an XML item will be the combination of its *namespace URI* and its *local name*. To represent these names in XML documents, URIs will need to be mapped to *prefixes*. 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 (`:`) as the *qualified name* 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 having certain elements and attributes ar-

```
<!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

ranged in a certain way. For instance, a processor for the document in Figure 2.2 will expect a `favorite-composers` 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 commonly called *well-formed* (though many will point out that this term is not needed, since there can be no non-well-formed XML). Schemas divide the class of XML documents into two sub-classes: *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 the XML specification [119], is called Document Type Definition (DTD). Rules expressible in a DTD provide a simple context-free 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 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

```
<?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:complexType>
  </xs:element>
  ...
  <xs:element name="born" type="xs:date"/>
</xs:schema>
```

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

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 two omissions are fixed with XML Schema [109, 110], an XML schema language developed by the W3C. Semantically speaking, XML Schema is a superset of DTDs [61], i.e., for any DTD there exists an XML Schema that validates exactly the same XML documents.

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 `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 [66]. This latter is based on the theory of tree languages and automata [10], and is seen by many to be a much cleaner solution than XML Schema. RELAX NG is strictly more expressive than either DTD or XML Schema [61].

The last well-known current schema language is called Schematron [45]. This language takes a different approach to the other schema languages described above in that it does not use any form of grammars to define XML

document structure. Instead, it uses *patterns*, which are matched against nodes of the XML document tree. These patterns then contain *rules*, which define how the environment around the matched pattern needs to look like.

Schematron can be seen to be a higher-level tool than the other schema languages, as the pattern language is strictly more expressive. Furthermore, Schematron is also recommended to be used as an additional tool with other schema languages, by using the other language to validate the many simple structural constraints, and then using Schematron to process the few constraints that are not expressible in other languages.

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 [96],

> 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 oldest one is XML Infoset [123], 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.

Another data model from the W3C, currently in its last stages to becoming a W3C Recommendation, is the XQuery 1.0 and XPath 2.0 data model [137]. This was produced for the needs of the XML processing languages XQuery [136] and XSLT [138], and their associated addressing language XPath [135]. It extends the Infoset with support for type information and collection representation.

It can also be said that any API for XML processing induces a data model on XML derived from the information presented by the API. For instance, the Document Object Model (DOM) [118] provides an essentially tree-like view of XML with support for both namespace use and namespace ignorance. Another API, Simple API for XML (SAX) [9], provides a sequential view of XML, splitting it into *events*, each approximately corresponding to an Infoset information item.

For the purposes of many applications, these various data models are perfectly suitable. However, as is pointed out in [132], 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). Furthermore, when signing XML documents it is imperative that the exact bytes that were signed can be produced by the verifier.

We can naturally see XML, produced by the grammar in the XML definition and complemented with a character encoding, as a byte-sequence-based 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 [107]. Canonical XML is a way to have several independent XML processors produce the same byte sequence from two "equivalent" XML documents. There is no explicit 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 [86], 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 for Messaging

The technologies described above can be considered to form a basis for XML-based messaging. Clearly the basic specification defines the syntax of messages. Use of namespaces makes it possible to specify pieces of generic functionality that can be added to any message. This is useful for, e.g., routing information, so namespace support is another necessary component.

As messaging is typically machine-to-machine communication, the syntax of messages can be more rigidly specified than with human-produced XML. The various schema languages can be used for this purpose. Since it will be quite common that a message envelope will be specified generically, ancillary information such as routing also generically but independently of the message envelope, and the actual message content by each application, namespace support is crucial, as is the ability to easily combine different schemas.

As we noted, messaging applications will typically view XML through some data model, as an interoperable representation format for their data. Serialization of such data is typically performed by traversing the atomic components of the data in some well-defined order, emitting the serialized form of each component as it is encountered. This kind of implementation

does not have an explicit data model for XML. Rather, it will implicitly use some streaming model such as SAX.

We also briefly touched on the XML processing and query languages XSLT, XQuery, and XPath while discussing data models. These technologies also have their place in a messaging application. For instance, XPath expressions can be used to select routing information from a message, either by locating a specific header or by making a decision based on a variety of content extracted from the message. Conceivably, XSLT could be used to transform messages, and possibly combine several messages into one. However, we are not aware of such use of XSLT; the typical implementations of message transformations appear to be based on non-XML technologies.

Finally, with messaging there are always the questions of security, privacy, and trust. These issues can be handled with digital signatures for authentication and encryption for confidentiality. In the XML world it is possible to selectively encrypt and sign XML documents using XML Encryption [113] and XML Signatures [114]. As alluded to in subsection 2.1.3, the XML Signature specification is complemented by Canonical XML [107] and Exclusive XML Canonicalization [111], which provide a distinguished form for serializing XML fragments. These two canonicalization specifications differ in how they treat the *context* of a fragment, e.g., the namespaces that are declared in some ancestor element of the fragment.

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 XML messaging. An alternate method of implementing Web services is *Representational State Transfer (REST)* [23], which is based only on the capabilities of Hypertext Transfer Protocol (HTTP), and in all ways attempts to build systems in the same manner as the WWW itself is built.

2.2.1 XML Protocols

The first well-known use of XML for interchange of programming language data was the XML-RPC [101] system of UserLand Software. This is a simple way to do Remote Procedure Calls (RPCs) using XML over HTTP. It supports encoding of structured data and arrays in the form expected of programming languages.

```
<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 2.5: The SOAP message structure

While XML-RPC is evidently suitable for a variety of applications, it lacks the kind of extensibility that is often required of distributed applications. To improve on this, Simple Object Access Protocol (SOAP) [105] 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 programming language data into XML, the so-called SOAP *encoding rules*, which define how to encode arbitrary programming language data as 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 Working Group (WG)[1] of the Web Services Activity[2]. This Working Group produced version 1.2 of SOAP [115], which relegates most of the areas specific to protocols and usage scenarios to its adjuncts [116].

The SOAP 1.2 specification only defines the outer structure of a SOAP message, illustrated in Figure 2.5. This figure shows the root element, `Envelope`, with its optional `Header` and mandatory `Body` children. The children of the `Header` element are called *header blocks*, and the example illustrates the common attributes that SOAP 1.2 defines for header blocks.

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 *SOAP intermediaries*, and finally

[1] http://www.w3.org/2000/xp/Group/

[2] http://www.w3.org/2002/ws/

being processed by the *ultimate receiver*. The `role` attribute specifies which processors in this chain are intended to process the header block, the `mustUnderstand` attribute set to `true` specifies that if the header block's processor does not understand it, it must respond with an error message, and the `relay` attribute set to `true` specifies that the header block's processor is to retain the header block in the message instead of removing it.

The SOAP 1.2 specification does not concern itself with the specifics of message transfer. It defines a protocol framework that can be used to specify how an underlying protocol can be 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 [112] and XMPP [26].

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 [106] for SOAP 1.1. The desired characteristics of this attachment feature were first specified on an abstract level [121].

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, there will need to be a Base64 encoding [27], 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 that the XML Protocol WG came up with was XML-binary Optimized Packaging (XOP) [134], a generic mechanism for including binary data in XML. XOP was intentionally limited to the case where data to be optimized is Base64-encoded in the Infoset representation of the XML. XOP 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` [55]. Any binary content inside the Infoset representation is then replaced with a pointer to the corresponding part in the package.

A method of using XOP to optimize SOAP performance with binary data is specified by SOAP Message Transmission Optimization Mechanism (MTOM) [125]. 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 [124] defines how the MIME type [28] 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 manner of defining extensions to the protocol. An extension will specify one or more new header blocks, 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[3] that is chartered with defining how messages are addressed so that they can be delivered to their proper destination. As a basis for this work there exists a submission [122] from a group of W3C members. The Addressing WG has so far produced Candidate Recommendations for the core principles [126] and for a SOAP binding [127].

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.

In addition to 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 [67], which defines a messaging service on top of SOAP 1.1 to support secure and reliable messaging. These reliability and security features have since been further refined by OASIS into Web Services Reliability [70] and Web Services Security [71].

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.

Web Services Security (WS-Security) makes it possible to sign and encrypt parts of SOAP messages. This complements transport layer security solutions such as Secure Sockets Layer (SSL) [29] by allowing true end-to-end security for SOAP messages, since SSL can only be used to secure traffic between SOAP intermediaries. Furthermore, being able to selectively encrypt and sign message parts makes it much easier to compartmentalize processing, since the outward-facing systems of a Web service need not do any security processing, just routing based on (unencrypted) headers.

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

WS-Security specifies a SOAP header that can contain a `Signature` element of XML Signature [114] to indicate signed parts of a message, and an `EncryptedKey` element of XML Encryption [113] that contains an encrypted (symmetric) key and references to message parts encrypted with that key. In addition, it is possible to send security tokens, such as X.509 certificates [39], to authenticate the message sender to the recipient.

2.2.3 Service Description and Discovery

While this thesis concentrates only on SOAP messaging, Web services include much more than just the messaging protocol. The intent is that Web services would be automatically discoverable and that this discovery process would produce information on how to invoke the services, i.e., what is the syntax of the SOAP messages expected by the service. Using XML everywhere and preferring late binding to the interfaces are seen as good options to support evolving of service interfaces (experience has demonstrated that evolving statically defined interfaces is extremely complicated).

To describe Web services, the W3C is currently specifying Web Services Description Language (WSDL) [128, 129]. This language allows the definition of service interfaces, which consist of the messages that the service accepts, responses that it produces, and any protocols that are available to invoke the service. These are all separated into different compartments so that the individual parts can be reused across different services.

The necessary late binding of services means that the WSDL description of a service will typically not be available to an application at compile time. To discover services at run time, OASIS has defined Universal Description, Discovery, and Integration (UDDI) [69], which allows the dynamic discovery of Web services and access to their WSDL descriptions. This description can then be interpreted by the application to construct a proper invocation to the service.

While in theory the specifications are all that is needed, in practice specifications are often implemented incorrectly or only partially. To remedy this, Web Services Interoperability Organization (WS-I), an organization dedicated to promoting Web service interoperability, has defined the WS-I Basic Profile [98], which clarifies the various specifications in an attempt to ensure better interoperability. However, the Basic Profile uses the old versions of SOAP [105] and WSDL [108], so it is of little help for more modern Web service systems.

2.3 The Mobile Environment

In recent years, the capabilities of devices such as mobile phones have increased so that they are now capable of more complex tasks than previous

mobile devices. This includes participating in computer networks as full-fledged members, providing functionality that is only possible through networking.

In this environment, however, there are several issues that are absent in the more typical fixed network setting with desktop computers and servers. The most obvious concern is that due to the required mobility of the devices, their connection to the network needs to be wireless, and one that supports efficient roaming between base stations on the fixed network side.

Commonly used current wireless networking systems include Wireless LAN (WLAN) [37], General Packet Radio Service (GPRS) [12], and Bluetooth [8], with third-generation mobile phone systems like Universal Mobile Telecommunications System (UMTS) [65] intended to supplant GPRS eventually. Of these technologies, Bluetooth is a short-range technology originally planned for replacing home computer system interconnections with wireless communication. However, it can feasibly be used to build small-scale ad hoc networks among independent devices as well [35]. WLANs are mostly suitable for indoor use as a replacement for fixed LANs, as their range of full-speed communication is not very long.

Since modern mobile phones and Personal Digital Assistants (PDAs) support several of these wireless communication technologies, it would also be beneficial to be able to switch between them. For example, when moving outdoors, GPRS is typically the network of choice, as it is most widely available without interruptions. However, when arriving at the office, using the office WLAN is the better option due to the lower speed, much higher latency, and higher cost of GPRS. Similarly, when encountering other devices outdoors, direct communication over Bluetooth is preferable to routing over GPRS through some central server.

Designing programs for mobile devices is different from the case of typical desktop computers. The most visible issue is the requirements that the device's form factor places on the user interface. A typical modern program for desktop computers has a mouse-based Graphical User Interface (GUI) consisting of several different *components*, such as buttons and text entry fields, to control the interaction.

This kind of interface does not work very well on mobile devices. For one, there is no mouse available, but a stylus is often used with PDAs to serve a similar role. A more pressing concern is the size of the screen, which simply cannot accommodate a complex GUI. Instead, style guides suggest reserving the screen for the most frequent commands and relegating less-used commands to menus [72].

However, as we focus on middleware, user interface design is not our concern. Instead, we must consider more the capabilities of the mobile device as compared with a desktop system. The main capabilities to consider are processor speed, memory size, and network characteristics such as bandwidth and latency.

In current mobile phones, processor clock frequency is on the order of 100 MHz and available memory is typically several megabytes. These capabilities are clearly more than sufficient for even sophisticated applications. On the networking side WLAN can achieve bandwidth of up to 54 Mbps with latency of a few milliseconds, which is clearly acceptable. However, GPRS can manage only 56 Kbps with a latency measured in hundreds of milliseconds. While UMTS increases the theoretical bandwidth to 2 Mbps, latency will still be very high.

The most pressing concern for mobile devices, though, is their battery-powered nature. All processing, memory use, and especially network use consume the battery. The battery needs to be recharged periodically, and currently outlets for such are typically available only at home or in the workplace. Furthermore, users will not wish to recharge their device batteries too often. For instance, a typical modern laptop computer can be used continuously for only a few hours before the battery needs to be recharged, which is unacceptable for a device such as a mobile phone that is expected to remain turned on at all times.

The concern for battery usage needs to permeate software design for mobile devices. In particular, transmission of data over a wireless network consumes a lot of energy, so the amount of communication needs to be minimized. Processing time is not quite as crucial, though it is clear that mobile devices are not capable of performing heavy-duty computational tasks. The proper tradeoff between communication and computation is likely to be highly device-dependent, so locking the design to certain figures would be a mistake.

For programming mobile devices there are several possible programming languages available. Our main focus has been on the Symbian OS[4] for mobile phones, for which Symbian C++ [34] and Java Mobile Information Device Profile (MIDP) [91] are the main development platforms. Lately, Python [97] has also become available, but we have no experience with that as of this writing. Of the two main platforms, we see Java as the better option, as the Java MIDP platform is quite similar to the Java Standard Edition [32], making skill transfer and code sharing much easier than with C++. However, skill transfer is not immediate, as there are several new issues to consider when programming for mobile devices [63].

2.4 Review of XML Performance Measurements

The rise of XML for purposes that were previously handled by specific binary formats has naturally raised concerns over the performance of XML compared to existing systems. This concern has been extremely strong in the mobile community, due to the limitations of the environment outlined

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

in section 2.3. There exist therefore several measurements of the effect of XML in various contexts. Below we summarize the work done in this area.

One of the oldest and best-known performance measurements of SOAP was done in the context of Grid computing [16]. This study investigated the bottlenecks that are present in an ordinary SOAP invocation in a typical scientific computing scenario. Various bottlenecks are then optimized, and the resulting system analyzed again.

For XML serialization and parsing this work introduces specialized improvements, especially for the case of handling arrays. The main goal is to process everything with a single pass through the data, all the way between the application and the Input/Output (I/O) buffer. Another improvement was the use of HTTP 1.1 to provide both persistent Transmission Control Protocol (TCP) connections and chunking of content.

The final performance issue, which in the end took 90 % of total processing time, was the marshalling and unmarshalling of floating point data. This kind of data was abundant in the messages due to the investigation being performed in the context of scientific computing. The authors propose extending SOAP with the capability to transfer some data in binary and to negotiate these extensions. Later, a similar desire was driving work in alternate XML serialization formats [133].

In its early years, SOAP and Web services were positioned as an alternative to existing technologies for distributed computing such as Java Remote Method Invocation (RMI) [90] and CORBA [64]. The concept was that SOAP would be usable over the Internet, something that RMI and CORBA had failed to deliver.

Earliest comparisons between these three technologies [19] were concerned with the latency of invocations. It was noted that CORBA and RMI deliver approximately the same performance, and the performance of even the best SOAP implementation was worse by a factor of 10 for a simple invocation.

This is explained by noting that the larger SOAP message needs to be split into several TCP segments, causing TCP's slow start to delay delivery by a network round trip. A further consideration was the Nagle algorithm of TCP: it turned out that SOAP implementations would push data over the network in non-full TCP segments, delaying the sending of any further data.

More complex measurements of this work provide similar or worse performance for SOAP implementations. As was the case with [16], large arrays are again measured as a significant problem in SOAP performance. In particular, the measured SOAP toolkits scale very poorly when array sizes are increased.

Further work in this area [22] looked at how various parameters of the SOAP implementation affect its performance in comparison with CORBA. Again, it was noted that the Nagle algorithm in conjunction with small

TCP segments decreased SOAP performance. Furthermore, the two XML parsers that were used had a factor of 5 difference in performance.

The conclusions of this work are that using HTTP 1.1 with persistent connections may be beneficial, especially over a high-latency connection. Similarly, the choice of the XML implementation can affect performance significantly. By calculating the improvements possible using various techniques, the work concludes that, using technology current at the time, it would have been possible to have SOAP performance only a factor of 7 worse than CORBA, in contrast with the factor of 400 that was initially measured.

A later comparison with RMI [46] examines different ways to implement distributed applications in Java. The benchmarked methods use only values of simple types such as integers and floating point values. The conclusion of the work is that Web services are a factor of 8 slower than RMI, with the SOAP implementation spending a majority of its time in marshalling and unmarshalling.

The above measurements have all concentrated mainly on SOAP processing performance. The networks in all of these have been high-speed LANs. There is little to no consideration of Wide Area Networks (WANs) such as the Internet or wireless networks such as WLAN or GPRS, and no measurements in either environment. From the observations made regarding Nagle's algorithm and TCP slow start, we would expect latency to be a significant issue when using wireless networks.

Our own performance measurements of SOAP [51] tested four different connections: loopback network, hosts on the same LAN, hosts on the same WLAN, and routing from our WLAN to our LAN. These measurements also explored compression of XML messages, using both generic compression and a simple binary format.

From these measurements we concluded that the main bottleneck in our wireless network was the need to open new connections. After network latency achieved a certain limit, adding compression did not worsen performance noticeably. We also noted that compression with a non-persistent connection still sends more data in total than a persistent connection without compression due to the additional TCP segments that are needed for opening of new connections.

Finally, [54] provides Web service measurements over both WLAN and Global System for Mobile communications (GSM), the latter invoking over a public GSM network. Furthermore, measurements were also made on actual mobile phones. Invocation time is split into several components and each component measured separately to better identify bottlenecks.

The conclusion of this work is that for the slowest networks processing time is dominated by network latency. This is observed to be the case even with the weakest processors. Using GSM the time taken by communication is measured to be over 90 % for even a very complex query. In contrast,

using WLAN, the time taken by communication remains under 30 % even in the case where there is little processing involved.

As a conclusion to all of these measurements we can see that SOAP messaging in the mobile environment is problematic in several different ways. Processing XML, especially with off-the-shelf tools, is costlier than processing a binary format. This applies in particular to typed data, which we expect to be present in abundance in SOAP messages. Furthermore, off-the-shelf SOAP toolkits do not appear to consider interaction between HTTP and TCP, causing performance degradation. This is particularly exacerbated by the high latencies in wireless networks.

Chapter 3

Message Transfer Service Overview

Based on the measurements presented in section 2.4 we compiled a set of requirements for an XML-based messaging system for mobile devices. We present these requirements below in section 3.1. Based on these requirements, we designed our XML-based Message Transfer Service (MTS) [48]. The design of the MTS is described in section 3.2 and details of its components in chapters that follow.

The MTS is a component of the Fuego service set[1], a middleware platform for the mobile Internet. In addition to messaging, this platform includes facilities for event notification [94], data synchronization [58], and presence information dissemination. The event notification service builds on top of the messaging, and the synchronization service uses the XML processing API that was originally developed for the MTS.

3.1 Requirements Analysis

As detailed above in section 2.4, several independent measurements indicate that there are two concerns with XML in the mobile environment. The size of XML is a problem because of wireless networks, and processing requirements are a problem because of weak devices. Therefore neither XML compression nor improvements in XML processing technology alone can satisfy these requirements. This is why an *alternate serialization format* based on some XML data model is seen by many as the best approach.

Currently there are several such alternate XML formats, and we cover them in detail in section 5.2 below. At the time of our design, the only public format for which information was available was WAP Binary XML (WBXML) [104]. This could not be adopted as such, as its design was for

[1] http://www.hiit.fi/fuego/fc/

a very specific purpose, and therefore not suitable for general XML-based messaging. Furthermore, while WBXML can be generalized [31], its features are still geared towards a very static form of data, and we wished to support many kinds of use cases efficiently. For these reasons, we decided to develop our own "binary XML" format, described in more detail in chapter 5 below.

The focus of the binary format needs to be on representing application data as SOAP messages for small mobile devices. The characteristics of the device require the implementation to have a *small footprint* so that it fits into available memory, and to be able to *process the format efficiently*, in both *time* and used dynamic *space*. The format itself needs to provide a *compact representation* of the data. As it is only used for interchange, it needs to be *readable and writable directly* between the serialized form and application data. Saving buffer space during processing is also important, so reading and writing should be possible in a *streaming manner*.

We also expect the application data contained in messages to consist of application-defined types at the programming language level. Therefore the format implementation will need to support *efficient encoding and decoding* of such typed data. Furthermore, as a complete or partial schema for messages is often available, a useful feature is to be able to *use this schema information* to improve efficiency. However, to retain some semblance of loose coupling, the schema needs to be *allowed to evolve* in common ways without invalidating existing processors.

In a binary XML format, *compatibility with XML* on some level is typically required. In our view, it is beneficial to achieve this compatibility at a *low-level API*, since that makes directly available all the functionality that has already been implemented for XML. A requirement for the system is therefore to include an abstract model for XML and an API to go with it that allows processing both XML and a binary format.

The ideal would be to be able to use an existing API for this purpose. Indeed, in our original version of the MTS we used the SAX interface [9] for processing XML. However, the needs of messaging are more focused on what is called *data-oriented XML*, meaning XML that mostly consists of structured data. The decoding of such typed data proved to be an arduous task with SAX, so we decided to design our own API to provide better type-handling capabilities.

Still, we wished to preserve compatibility with XML, so we based our API on another actual XML API. Our requirements for this were that it be possible to both read and write XML in a *streaming manner*, to easily encode and decode *typed data*, and to have standard APIs for *both reading and writing*, the latter of which SAX lacks. Our contribution is mainly in extending our selected API with typed data handling and in formalizing the data model associated with it.

Even now, many are of the opinion that an alternate serialization for-

Table 3.1: Requirements on message transfer service components

Component	Requirements
XML API	compatibility with XML, low level, data-oriented, streaming, typed data, input and output
XML Serialization	small footprint, processing time, processing space, compact representation, directly streamable, typed data, schema enhancements, schema evolvability
Message Protocol	asynchronous interface, small headers, sending and receiving

mat for XML is sufficient to solve the issues with XML usage. However, the mobile environment has requirements beyond small message size and efficient processing. The synchronous RPC interface provided by typical SOAP implementations is very wasteful over wireless connections where network round trips can last on the order of seconds. This necessitates the use of *asynchronous interfaces* as the main ones for two-way messaging.

Furthermore, the most commonly used protocol is HTTP. HTTP itself is a very useful protocol, and has some features that make it very suitable in the case of client mobility (we encounter these later in section 6.2). However, typical HTTP usage adds a *large amount of headers* onto each message, potentially doubling the size of a simple SOAP message. Per the law of diminishing returns, an alternate serialization format for XML will not be a significant improvement if most of a message consists of protocol headers.

Another consideration on the protocol layer is its precise semantics. To be a full-fledged member of a larger network, a node needs to be able to *both send and receive* messages. However, typical ways of connecting a mobile device to the Internet use Network Address Translation (NAT) [87], which makes it impossible for the outside to initiate contact with the mobile device. For this reason, the protocol needs to support *two-way communication*, which HTTP as a single-request-response protocol with clearly defined client and server roles does not do.

We summarize our collected requirements in Table 3.1. We note that many of the requirements are shared between the processing API and serialization format. This indicates a potential for coupling their designs very closely. The requirements for the protocol are not very specific to XML, but are applicable to any messaging system for the mobile environment.

3.2 System Architecture

The overall view of the MTS, as currently implemented, is shown in Figure 3.1. The message service component on the upper left binds the com-

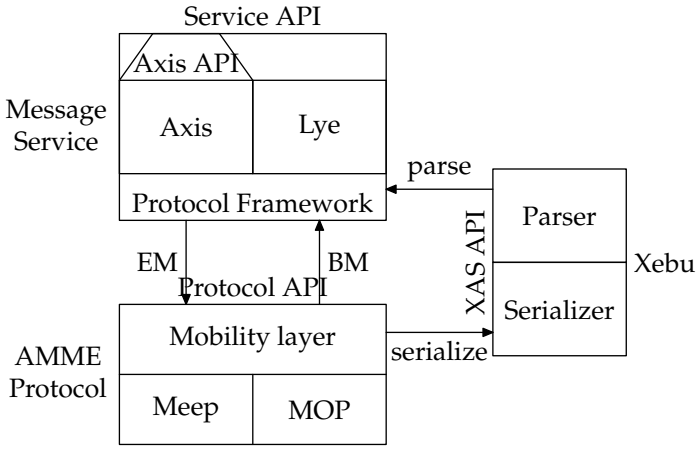


Figure 3.1: The Message Transfer Service architecture

ponents together and is the main interface for applications. We describe this main component in this chapter and leave the internals of other components to later chapters. In the figure, `EM` is an encodable message that will be serialized by the protocol layer, and `BM` is a sequence of bytes that will be parsed by the message service component.

The MTS is divided into three separate components, the *message service*, the *message protocol*, and the *XML serialization*. All of these provide generic interfaces and have at least two implementations each. The message protocol and XML serialization components are the topics of later chapters.

In Figure 3.1 the message service component provides two interfaces to the outside world: the *service API* and the *protocol framework*. The former is for use by messaging applications, and the latter is for pluggable protocols. We have, in fact, implemented several different protocols using the message service's protocol framework, but Abstract Mobile Message Exchange (AMME) is the most featureful of these.

The service API provides a class for messages, instances of which are constructed by applications and passed to the message service for delivery. The data in messages can be specified either as XML or as a collection of name-value pairs. The names in the latter are hierarchical, and also serialized as hierarchical XML. SOAP headers may also be specified for messages, but for them only XML is available.

Various properties required by the MTS to direct and correlate messages are specified in SOAP headers. This is similar to Web Services Addressing [126], except that we use simple strings and numbers instead of URIs. For example, each message gets a unique identifier so that responses to messages can be dispatched to the proper target.

Messages are always directed at *destinations*. In essence, a destination is a Uniform Resource Locator (URL) separated into component parts. Its

components are protocol, server address, server port, and target. The protocol names a Message Transfer Protocol (MTP), and may in addition include *features* that specify additional information on the type of connection required. The server address and port are the same as in normal HTTP URLs. The target is the local name of the target on the server side and is used to dispatch the message.

The two basic message sending operations are `send` for one-way messages and `sendCallback` for asynchronous two-way messages. The basic two-way messaging operation needs a callback object provided by the application that is then invoked when the response arrives. The callback style of two-way messaging is simple, yet powerful, permitting different Message Exchange Patterns (MEPs) to be easily implemented.

The invocation method of the callback interface permits a message to be returned by the application. If the received message was one for which a response was expected, indicated by a specific SOAP header, the service will send the returned message back. As the application can also mark this message as one to which a response is expected, the callback style can easily be used to implement the conversation MEP, which consists of a sequence of messages sent back and forth between the parties.

The service supports two different kinds of callback objects. *Persistent* ones are explicitly registered by the application and they remain known until the application deregisters them. *Transient* ones, on the other hand, are generated by the service for a single MEP and are deregistered after the MEP has completed. Each non-one-way message carries a SOAP header to identify its sender. If this sender is a persistent one, the receiver can store it and use it at any later time as a message target. This can be used to provide the subscribe/notify MEP.

We also provide other semantics for two-way messaging, all implemented generically on top of the callback interface. The other major asynchronous two-way style, polling, is implemented as a *future* object [33] that is registered as the callback. By forcing a synchronization of the future object immediately, it is possible to provide a synchronous two-way invocation. For reasons detailed above, we do not, however, recommend using the synchronous request-response pattern. These other semantics only support request-response interaction, as specifying more flexible semantics for these styles is not feasible.

As with the rest of the system, the service API is a generic one, permitting multiple implementations. We provide two of these, which we call the *Axis service* and the *Lye service*. The Axis service is built around the Apache Axis[2] SOAP implementation, and only alters the protocol processing and serialization performed by Axis. As Figure 3.1 shows, we also provide the standard Axis API to applications to permit some compatibility with stan-

[2] `http://ws.apache.org/axis/`

dard Web services. As Axis is not usable on mobile devices, we implemented from scratch the Lye service for the Java MIDP platform. The Lye service is intended as a very simple one that should be suitable for mobile devices.

Chapter 4

XML Processing Interfaces

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 can be *mixed*, i.e., consisting of both text and elements. 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. Elements are typically rigidly structured, and contain either only other elements or a stringified representation of some programming language data value. Documents can be very small, and the preference is to transform them in a streaming manner between XML and their application-specific form.

4.1 Existing Interfaces

The two best-known interfaces for processing XML data are SAX [9] and DOM [118]. Of these two, SAX is intended for streaming parsing. During SAX parsing the parser is in control and invokes a registered callback handler for each *SAX event* encountered during parsing. DOM, on the other hand, represents the entire XML document in a tree format, and provides a multitude of links needed to navigate the document.

When selecting the XML processing interface for our messaging system, we immediately rejected DOM for consideration. Our interest in XML is purely as a data interchange format, and application-level representation of any transferred data will be tailored specifically to that application. Adopting DOM as the model would therefore require applications to hold two different representations of data in memory, with the DOM version

taking a large amount of additional space to represent the required navigational links. Since we endeavor to save space usage in our system, DOM cannot be considered appropriate for our purposes.

Our first implementation of the MTS used the SAX API for XML processing. Since SAX does not preserve any state specific to itself, it is very well suited to the mobile environment. However, a general message transfer service will need to exchange arbitrary application data structures, so it needs a data binding system [89] that can be extended with application-specific types.

This data binding requirement was ultimately the reason why we had to reject SAX. The SAX processing model keeps the parser in control of the parsing. If the application wishes to construct a complex data structure from XML data, it will need to remember partial data that has been fed by the SAX parser and to finally construct the structure when encountering an element end event. In short, this means that the decoder will need to be implemented with state that is updated as callbacks from the parser are received.

In contrast, a pull-style API where the application is in control and requests the parsed events from the parsing API one at a time provides a much more natural way for decoding structured data. In this case the program counter is sufficient to tie the element start events to the content of their respective elements, and required state is naturally kept in local variables. Furthermore, processing distinct elements appearing in arbitrary order is no more difficult than it is with a push-style parser such as SAX.

Later well-known APIs include JDOM[1] and Streaming API for XML (StAX) [7], the former of which is similar to DOM and the latter to SAX. As we rejected DOM for fundamental reasons, we did not even consider JDOM. StAX, having a pull model, would have been more suitable to our needs, but it did not appear early enough to be considered. However, our adopted solution is a precursor to StAX, so a future migration to StAX as the more standard API is not ruled out.

4.2 The XAS Data Model

For the reasons above, we decided to base our data model on a pull-style event-based API. As the underlying API we chose XmlPull [84], in part because that API is used in the kXML[2] implementation of XML parsing and serialization for mobile phones.

Our model, XAS [47], formalizes the implicit data model provided by the XmlPull API. The basic object in the model is an *event*. This represents a single event as detailed in Table 4.1. These types are the same as those

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

Table 4.1: Event types of the XAS data model

Event	Data
DOCUMENT START	none
DOCUMENT END	none
PREFIX MAPPING	name
ELEMENT START	name
ELEMENT END	name
ATTRIBUTE	name, value
CONTENT	value
TYPED CONTENT	name, value
COMMENT	value
PROCESSING INSTRUCTION	value
ENTITY REFERENCE	value

provided by the low-level `getEventType` method of the XmlPull's `XmlPullParser` interface, and the data in each event is what is provided for the `XmlSerializer` interface.

In Table 4.1 the *name* in event data refers to a pair consisting of a namespace URI and a local name. The only representation of namespace prefixes in XAS is present in the PREFIX MAPPING event that contains a namespace URI and the prefix to map it to. This decision was made because XML Namespaces distinguishes names based on their namespaces and not on any prefixes that the namespaces might be mapped to.

The XAS data model leaves out document type declarations. Since our main purpose is to use this model for SOAP messaging, this is reasonable because SOAP prohibits these declarations. Furthermore, we consider validation better to perform after converting a document into its XAS representation. The model also leaves out CDATA sections, since their existence (as well as that of the standard entities `<`, `&`, etc.) is an artifact of the XML serialization format, and has nothing to do with an abstract model.

A new event type in the XAS model is the TYPED CONTENT event. This represents typed data that will need to be encoded according to the rules of the serialization format. This event was created because our messaging system supports alternate serialization formats, which might have more efficient ways to encode common data types. Hence, typed data needs to be represented abstractly at the data model level, so that it can stay in its internal representation until it needs to be serialized. In a TYPED CONTENT event the name is the name of the type in the manner of XML Schema [109] and the value is the data itself, represented in a manner appropriate to the language used.

A complete XML document is represented as a sequence of XAS events. The first and last events of this sequence are a DOCUMENT START event and

a DOCUMENT END event, respectively. An ELEMENT START *block* consists of a sequence of PREFIX MAPPING events, followed by an ELEMENT START event and a sequence of ATTRIBUTE events. Either of the sequences may be empty. An ELEMENT START block therefore corresponds to the start tag of an element in an XML document. An *element* is a sequence starting with an ELEMENT START block and ending with an ELEMENT END event of the same name as the block's ELEMENT START event. The scope of the namespace prefix mappings in an ELEMENT START block is the element started by it.

Between an ELEMENT START block and its corresponding ELEMENT END event is *element content*, a sequence of elements and CONTENT events. Two consecutive CONTENT events are equivalent to a single CONTENT event whose value is the concatenation of the two events. Element content may also be a single TYPED CONTENT event. Of the other event types, COMMENT and PROCESSING INSTRUCTION events are permitted between any two events in the sequence except within an ELEMENT START block. ENTITY REFERENCE events are permitted in the same places as CONTENT events are.

An example XML document and its mapping to a sequence of XAS events are shown in Figure 4.1 where we represent names as pairs of a prefix and a local name to save space. In reality, each `p`, apart from the one in the PREFIX MAPPING event, would be `http://example.org/people`. For the TYPED CONTENT events we use the prefix `xsd` to refer to the XML Schema datatypes namespace.

4.3 The XAS API

XML processing in our messaging system is based on an API derived from the XAS data model. The XAS event maps to a Java class `Event`, which is a discriminated union of the possible XAS event types. The class contains fields for all possible data contained in a XAS event and accessor methods for these. A sequence of events is represented by the interface `EventSequence`. This interface has several alternate implementations, each appropriate for a different use.

At the lowest layer are the XmlPull APIs, `XmlPullParser` and `XmlSerializer`, both extended with handling of TYPED CONTENT events described above. When using XAS, the former is typically wrapped inside an `EventStream`, an `EventSequence` implementation that (lazily) calls on the parser to produce new events only when the application demands them.

The other `EventSequence` implementations are `EventList` and `EventSerializer`, both of which are used for application-controlled `EventSequence` construction. The `EventList` class implements methods similar to Java's standard `List` interface, and the `EventSerializer` class implements

```
<?xml version="1.0" encoding="UTF-8"?>
<composers xmlns:p="http://example.org/people">
  <p:person p:nationality="DE">
    <p:name>
      <p:first>Richard</p:first>
      <p:last>Wagner</p:last>
    </p:name>
    <p:occupation>Composer</p:occupation>
    <p:born>1813-05-22</p:born>
    <p:died>1883-02-13</p:died>
  </p:person>
</composers>
```

DS(UTF-8) PM(http://example.org/people,p) ES(,composers)
ES(p,person) A(p,nationality,DE) ES(p,name) ES(p,first)
TC(xsd,string,Richard) EE(p,first) ES(p,last) TC(xsd,string,Wagner)
EE(p,last) EE(p,name) ES(p,occupation) C(Composer) EE(p,occupation)
ES(p,born) TC(xsd,date,1813-05-22) EE(p,born) ES(p,died)
TC(xsd,date,1883-02-13) EE(p,died) EE(p,person) EE(,composers) DE

Figure 4.1: An example XAS event sequence

XmlPull's `XmlSerializer` interface to allow an application to produce an `EventSequence` in lieu of outputting an XML document.

All of these classes provide only an event-by-event view of an XML document. To provide a more XML-like view, we implemented two higher-level classes, `XmlWriter` and `XmlReader`. The former is a wrapper around an `XmlSerializer` and provides methods for inserting complete elements. The latter is basically a cursor into an `EventSequence`, but also includes methods for accessing complete elements at the cursor's position.

One intent of the XAS API was to allow simple chaining of various XML processors as classes that wrap and implement the `EventSequence` interface. We implemented an abstract `TransformedEventSequence` class that worked acceptably for this purpose, but mostly on the input side. This class wraps an underlying `EventSequence` and defines an abstract `transform` method that can perform arbitrary *m*-to-*n* transformations of the events from the underlying sequence.

As the XAS API was originally designed for our messaging system, it also includes the possibility of selecting a serialization format different from XML. The serializers and parsers for the actual serialization format are accessed through a collection of *factories* [30], which are registered based on the MIME type associated with their serialization format (like `text/xml` or `application/soap+xml` for XML data).

As many existing XML-using systems process XML through more standard APIs such as SAX or DOM, we also implemented compatibility inter-

faces for both of these. The SAX interface is a two-way converter that converts SAX events into XAS events and vice versa (note that an element start event in SAX also contains the element's attributes, and therefore is converted to several XAS events). The DOM compatibility interface takes an `EventSequence` representing a complete document and converts this into a DOM tree, or vice versa.

4.4 Typed Data in the XAS API

One reason for designing a new API for XML processing was to integrate typed data handling into the system. Our main purpose in doing this tight integration was to allow the serialization format described in chapter 5 to handle typed data efficiently. Otherwise typed data would have to be converted to and from a string representation, which would have eliminated any performance benefits gained from the binary form used by our format.

In the XAS API, the value in a TYPED CONTENT event is a Java `Object`, so it can be any value representable in Java, including one defined by the application programmer. The XAS system includes a mapping between XML Schema types and Java classes so that the typed data handler can always determine an appropriate type when converting.

Encoding and decoding of typed data are handled by objects of classes `ContentEncoder` and `ContentDecoder`, respectively. Such objects are installed into the serializer and parser, and whenever typed data needs to be handled, the installed encoder or decoder is invoked. Typically these objects are *chained*: if one does not support the given type, it will pass its arguments unchanged to the next one in the chain. Only if the last processor in the chain does not recognize the type, is an error raised.

In our view, typed data can be divided into two classes, *primitive* and *complex*. Primitive typed data is data that would appear as the (sole) text content of a single element whereas complex typed data is everything else. Our preference is that the serialized form of complex typed data would always consist of a sequence of elements, and not have mixed content.

The cause of this division is that encoders and decoders for primitive typed data need to be implemented separately for each distinct serialization format. On the other hand, if our preference for complex typed data is followed, their encoders and decoders will be independent of the underlying serialization format. The system, as currently implemented, does not support application programmers defining new kinds of primitive typed data, except by directly modifying the source code of every format implementation.

The encoding interface is very simple. There is one method, `encode` that takes as arguments the type name and the object to be encoded. The method is also passed the serializer to use. As the serializer interface does

not allow access to the output stream, it is not possible for the `encode` method to directly output any data, so primitive typed data cannot be handled.

The decoding process is more complex. It is handled by two mutually recursive methods, `decode` and `expect`. Of these, the `expect` method's implementation is shared by all `ContentDecoder` objects whereas `decode` needs to be implemented for each type separately. The arguments of the `decode` method are the type name and an `XmlReader` where the cursor is positioned after the ELEMENT START block of the element to be decoded. The method is expected to decode the element's content as a typed object, return this decoded object, and to leave the cursor immediately before the ELEMENT END event ending the decoded element.

The `expect` method starts at an ELEMENT START block. It reads the type attribute from the ELEMENT START block, and then calls the `decode` method giving this type as an argument, with the cursor positioned correctly. For primitive typed data our parsers provide the TYPED CONTENT events directly so there is nothing for the `decode` method to do. For complex typed data the code of the `decode` method typically consists of a sequence of calls to `expect` for each of the components of the data to decode.

We used the `TransformedEventSequence` class to implement a transformation of a regular event sequence into a typed event sequence. This `TypedEventSequence` class takes an `EventSequence` and a `ContentDecoder`, and produces an `EventSequence`, which contains TYPED CONTENT events for all the types that the supplied decoder understands.

4.5 Example of Typed Data Handling with XAS

We next provide an example of typed data encoding and decoding. An example Java class and an XML encoding of its instance are shown in Figure 4.2. We will further assume that the `Person` and `Name` classes follow the Java Beans framework [88] by providing `public` accessor and mutator methods for each of their `private` fields. We also assume a constructor for both classes that takes all the components as arguments. We will omit all namespace handling, as that would just clutter the example without providing any additional insight.

The encoding process in Figure 4.3, shown in Java-like pseudocode, is straightforward. The encoder is provided with the object to be encoded, the name of its type, and the serializer to use. An `XmlWriter` is constructed from the serializer to make the encode process a simple sequencing of the individual components of the object.

The decode process in Figure 4.4 is not much more complicated. However, we have here omitted all error handling. If the `expect` method does not manage to decode any typed data, it will return `null` and not advance

```
public class Person {
  private Name name;
  private String nation;
  private String work;
  private Calendar born;
  private Calendar died;
}

public class Name {
  private String first;
  private String last;
}
```

```
<person nation="DE">
  <name>
    <first>Richard</first>
    <last>Wagner</last>
  </name>
  <work>Composer</work>
  <born>1813-05-22</born>
  <died>1883-02-13</died>
</person>
```

Figure 4.2: An example Java class and its XML-encoded form

PERSON-ENCODE(*o*, *type*, *ser*)

```
1  if type = "person"
2     then
3          XmlWriter writer ← new XmlWriter(ser)
4          Person person ← (Person) o
5          writer.addEvent(Event.attribute("nation", person.getNation()))
6          writer.typedElement("name", "name", person.getName())
7          writer.typedElement("work", "string", person.getWork())
8          writer.typedElement("born", "date", person.getBorn())
9          writer.typedElement("died", "date", person.getDied())
10    else
11         chain.encode(o,type,ser)
12
```

NAME-ENCODE(*o*, *type*, *ser*)

```
1  if type = "name"
2     then
3          XmlWriter writer ← new XmlWriter(ser)
4          Name name ← (Name) o
5          writer.typedElement("first", "string", name.getFirst());
6          writer.typedElement("last", "string", name.getLast());
7     else
8          chain.encode(o, type, ser)
9
```

Figure 4.3: Example encoding code

```
PERSON-DECODE(type, reader)
 1  if type = "person"
 2     then
 3          Event at = reader.advance()
 4          String nation = (String) at.getValue()
 5          Name name = (Name) expect("name", reader)
 6          String work = (String) expect("work", reader)
 7          Calendar born = (Calendar) expect("born", reader)
 8          Calendar died = (Calendar) expect("died", reader)
 9          return new Person(name, nation, work, born, died)
10     else
11          return chain.decode(type, reader)
12
```

```
NAME-DECODE(type, reader)
1  if type = "name"
2     then
3          String first = (String) expect("first", reader);
4          String last = (String) expect("last", reader);
5          return new Name(first, last);
6     else
7          return chain.decode(type, reader)
8
```

Figure 4.4: Example decoding code

the `reader`. Any implementation of the `decode` method needs to behave in the same manner.

We can see from these examples that writing encoding and decoding code can be tedious and repetitive, as the procedure to be followed is essentially the same in all cases. Furthermore, the error handling that we omitted is also the same everywhere. This is why we designed a language called Object Representation Language (ORL), which can be used to describe the structure of a Java class. To accompany this, we implemented a program that generates the appropriate encoder and decoder classes from the ORL definition. The ORL definition of our example is shown in Figure 4.5.

The ORL syntax is intentionally very simple. The `type` keyword introduces the name of a structured type. The content of the type is enclosed in braces `{}`, and consists of pairs of type name and component name. The

```
type name {
  string first
  string? middle
  string last
}

type person {
  name name
  string nation
  string? work
  date born
  date? died
}
```

Figure 4.5: An example ORL file

type name may be either a predefined one, such as `string`, or a structured type defined in ORL, such as `name`. Additional syntax not shown here is available to handle namespaces for the types and Java packages for the generated encoders and decoders. Our code generator produces essentially the code in Figures 4.3 and 4.4, but with error handling, from the file in Figure 4.5.

The example in Figure 4.5 shows a feature that our encoding and decoding example did not cover. In ORL, a component can be marked with a `?` to indicate optionality and with a `*` to indicate repetition. The generated encoder and decoder will automatically deal with these cases. Furthermore, the decoding process using ORL allows more flexibility, as the construction of the object is left to the application. This permits using either a constructor as in our example above, mutator methods, or factories.

We note that an ORL definition for the encoding of a datatype essentially provides a schema for the part of the document consisting of that datatype. In light of the techniques presented later in section 5.4 it could be useful to integrate ORL into any schema-handling present in our system, and to allow such an application-defined schema to also be used for optimization.

Chapter 5

Alternate XML Serialization

The idea of an alternate serialization format for XML is not a new one. As one design principle of XML, as listed in 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 second half of this chapter presents our own serialization format.

5.1 XML Compression

XML documents have much textual redundancy, so they compress very well with generic text compression tools. However, XML has structure beyond the linear one expected by a generic compressor. For instance, it could be expected that elements with the same name (e.g., multiple `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 than the very popular general-purpose compressor gzip [20], which implements the Lempel-Ziv compression algorithm [139].

One of the best-known XML-specific compressors was XMill [57]. 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 reads an XML document using SAX and splits the generated events into different *streams*. There is one stream for the structure (tags), and a number of data streams. A user can specify the names of elements that are included in each data stream. Default data streams are

then constructed individually for each element type that was not included in the user's definitions. The structure stream also contains pointers to the data streams so that the XML document can be reconstructed after the transform.

XMill allows the user to specify *semantic compressors* for data streams. 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 against the data value to eliminate common parts directly.

In the final phase, gzip is applied individually to each stream (to the data streams after semantic compression), and the streams are concatenated to form the final document. Measurements on XMill reported in [57] indicate that XMill performs better than gzip on many kinds of XML data, and furthermore that an original text document converted to XML and compressed with XMill is smaller than the original document compressed with gzip. Timing measurements indicate that XMill is approximately as fast as gzip, both compressing and decompressing.

However, while XMill performs well when combined with gzip, there exist better algorithms for textual data compression. A currently-popular one is bzip2[1], which uses the Burrows-Wheeler transform [11] to preprocess the data into a more compressible form. Bzip2 achieves a compression ratio comparable with state-of-the-art compressors while being much faster than them.

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 [15].

After noticing that XMill's modeling of XML data is not sufficient, the author of [15] proposes a technique called *Multiplexed Hierarchical Modeling (MHM)*, which is based on the well-known Prediction by Partial Matching (PPM) compression technique [17]. 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; this 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 mod-

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

els even further. This process has been implemented in the XMLPPM[2] tool.

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 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 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 [44, 14].

Even if we do not assume a session, there may still be a WSDL description of a service endpoint. This description can be used to create a template message, to which differential encoding is applied [100]. 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.

5.2 XML Binary Characterization

However, 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 is typically the case where the term "binary XML", as first articulated by WBXML [104], is mentioned. This term refers to a binary serialization format that is designed to be compatible with XML and according to the same principles, permitting streaming between the serialized form and an application data model.

The concept of binary XML has become popular in the recent years. The W3C has followed the situation, and in September 2003 organized a workshop on Efficient Interchange of XML Information Item Sets [117]. Several participants in this workshop presented their own binary formats, and as

[2] http://xmlppm.sourceforge.net/

a result, the W3C chartered the XML Binary Characterization (XBC) WG[3] (the author of this thesis 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 [130], supported by use cases [133], required format properties derived from the use cases [132], and ways to measure the properties [131]. The findings were that a binary format that supports the 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[4] to provide such a format, either by developing one itself or by adopting an existing format (our format described in this chapter has been submitted to the EXI WG for consideration).

For our purposes, the most interesting one of the use cases identified by the XBC WG is called *Web Services for Small Devices*. This use case coincides with our application area very closely, and the analysis of requirements in the use case matches our own, presented in section 3.1, nearly exactly. However, our analysis lifted as necessary properties also *Schema Extensions and Deviations* to permit evolution of message schemas and *Specialized Codecs* to permit integration of application-specific data structures.

Binary XML techniques can roughly be divided into Infoset-based and schema-based [74]. 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.

5.3 Tokenization Techniques

One basic concept of binary XML formats that has been used by many of the existing well-known formats is called *tokenization*. This is similar to what generic compressors like gzip [20] 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 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 takes quite a bit of processing power, the tokenization performed by binary XML formats is much more efficient. This

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

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

is because the 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.

5.3.1 Existing General-Purpose Formats

The oldest format, WBXML [104], 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 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 Wireless Application Protocol (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 [31] 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 binary format is indubitably Fast Infoset [79]. This format represents the information items of XML Infoset in an Abstract Syntax Notation One (ASN.1) schema [41]. Then, it is possible to use the well-established *encoding rules* of ASN.1 [40, 42] 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.

Another somewhat similar general-purpose format is XBIS [85]. 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 is again tokenization. Names of elements and attributes are always tokenized, while tokenizing text and attribute values is optional. A document is serialized 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.

5.3.2 Basic Xebu Format

Our format, Xebu [50], is derived directly from the XAS model presented in section 4.2. Each event of XAS maps directly to a serialized form in Xebu. The serialization of an event begins with 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. This is followed by the content of the event.

Each string in an event's content is given 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. In our messaging system these tokens can be specified beforehand, and dynamic tokenizations can persist across messages in a single communication channel.

Xebu includes four separate *token mappings*, for namespaces, names, values, and text. Namespaces are simply the namespace 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. We considered the case where the same local name belongs to two different namespaces to

be semantically insignificant to optimize for.

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 various 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. Because of this, a technique like Adaptive Replacement Cache (ARC) [59] that provides two classes of tokens, persistent and temporary, could be beneficial.

At the moment we are considering a three-level split of the token space where a temporary token can either persist until replaced or be invalidated when the depth of the processed tree goes above the one where the token was assigned. The latter kind would allow self-contained XML fragments to be serialized, while still retaining most of the benefits of tokenization. As mentioned above, the design of Xebu makes experimenting with alternate policies very simple, since only the serializer side needs to be modified.

Another Xebu feature, also common in other binary XML formats, is the binary encoding of known data types. The TYPED CONTENT event of XAS was designed to allow this kind of alternate encoding without going through a string representation. This will save space in many cases and can also improve performance for certain data types.

5.4 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 structure presented in section 2.2. Furthermore, in many cases there will be schema information on existing 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.

5.4.1 Existing Schema-Based Formats

Existing binary formats that can take advantage of schema information include BiM of MPEG-7 [62], Fast Web Services [78], and XML Schema-based Binary Compression (XSBC) (formerly called Cross-Format Schema Protocol (XFSP) [82]). There is also a schema-optimized version of Millau [92]. These all take a slightly different approach to using the schema, and we present these approaches next.

XSBC [82] is a very simple format. Its approach is basically pre-tokenization based on the element names in the schema combined with encoding typed data specially, determining the correct encoding from schema information. In this it resembles the original version of our Xebu format [49]. 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 [6] 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 [92] 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 reported in [92] 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 optimization, requiring that the DTDs used with this technique do not have too many choices available.

Fast Web Services [78], 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 [43] 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) [42], 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 in [78] 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 increases 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 [18] indicate that ASN.1 PER can achieve up to 50-fold improvement in document size compared to XML. However, timing measurements were not included in this work.

The BiM format [62] was designed for use with the MPEG-7 metadata format [5] 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. At each point in an XML document there is typically only a very small number of possible following items, and the BiM automaton transitions can then output the minimal number of bits required to distinguish between these alternatives. Measurements [18] indicate that BiM is capable of achieving over 10-fold reduction in document size.

None of the formats above permit deviations from the given schema, but XSBC at least would be simple to extend for this. We see this rigidity as a liability, since in real use it is not uncommon that schemas are not universally available or are not exactly the same everywhere. Furthermore, different use cases may require different schemas to be applied to the same XML document at various times.

Reputedly, Efficient XML [81] is capable of performing schema-based optimizations without sacrificing the ability to serialize any XML document. This is based on principles of information theory [83] by noting that a document conforming to a schema has less entropy with respect to the schema than other documents. However, there is little technical information available of Efficient XML so these claims cannot be evaluated.

5.4.2 Schema Optimization Design

Our approach to schema-based optimization is similar to that of BiM with its automata. We construct what we call a *Codec Omission Automaton (COA)*, which is a pair of automata, *Encoding Omission Automaton (EOA)* for the serializer side and *Decoding Omission Automaton (DOA)* for the parser side.

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

Their input and output are both XAS event sequences, 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 established for modeling XML as XAS sequences, which were described in section 4.2. A sufficient requirement for the format is that the serialization of a XAS event is *context-free*, i.e., a serialized XAS event can be read without knowing the events that were read previously.

XML itself is not context-free as we have defined the term. One reason is that recognizing an attribute requires first the processing of the attribute's start tag, and therefore an attribute cannot be reliably recognized if its ELEMENT START event is omitted. Of the formats covered above, we believe that at least XSBC satisfies the requirements for context freedom.

The schema optimization that we perform is simply the omission of events from the input event sequence of the EOA. Since we perform only a transformation to another XAS 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 XAS event sequences.

The omission of events introduces issues that are not covered by the XAS model, but will need to be handled by the serialization. An issue that could break the system is the coalescence of CONTENT events. The XAS model allows an element's text content to be represented as multiple consecutive CONTENT events. However, if events are omitted so that two separate text contents become adjacent, the parser side will need to recognize where the first content ends. To solve this, we introduce a *coalesce flag* into CONTENT events; this flag determines whether a CONTENT event is a part of the same text content as an immediately preceding CONTENT event.

The other case where we extend the XAS model is not a matter of correctness, but simply an optimization. In the Xebu format a TYPED CONTENT event is typically not length-prefixed since the decoder is written so that it reads a correct number of bytes. If now event omission brings two TYPED CONTENT events next to each other, these would normally be serialized as separate TYPED CONTENT events. To improve space usage we introduce a TYPED MULTICONTENT event, which gathers a sequence of encoded typed data elements and prefixes these with the length of the whole sequence. This eliminates the need for separate discriminator tokens for each piece of data, which especially helps, e.g., the case of lists of integers.

As our schema language we chose RELAX NG, mainly because it has, in addition to XML syntax, a standardized compact syntax [68] more resembling traditional programming languages. This compact syntax is both easier for humans to handle and more amenable to traditional parsing tech-

niques.

Our language of choice for implementing the COA generator was Standard ML [60], whose features are a good match for implementing compilers [3]. 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. We also used the combinator technique for our parser [25], 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 this has not yet materialized to even a design.

5.4.3 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 a XAS event sequence, and their transitions on these XAS events have specifications on what XAS events to output. In both automata transitions also have, in addition to a XAS 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 XAS event. In the case of a XAS 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 XAS 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 provide 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 are otherwise the same as those 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.

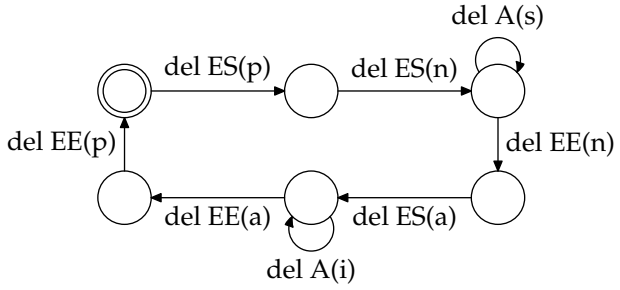
An example RELAX NG schema and its associated generated COA are given in Figure 5.1. The schema (a) says that a `person` element is a sequence of elements `name`, whose content is a string, and `age`, whose content is an

```
start = element person {
      element name { xsd:string },
      element age { xsd:int }
}
```

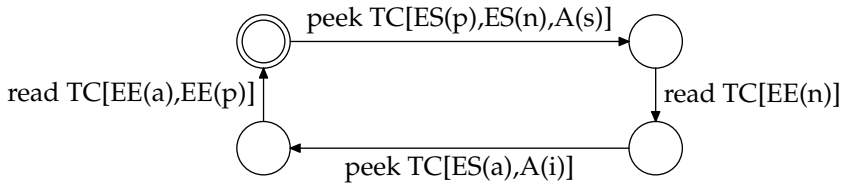
(a) Schema

ES	ELEMENT START
EE	ELEMENT END
A	ATTRIBUTE
TC	TYPED CONTENT
p	`person`
n	`name`
a	`age`
s	`type=xsd:string`
i	`type=xsd:int`

(b) Legend



(c) EOA



(d) DOA

Figure 5.1: An example COA

integer. The legend (b) provides some abbreviations for the EOA in (c) and the DOA in (d).

From Figure 5.1 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 for the type to allow the parser to decode the TYPED CONTENT event properly, since Xebu does not contain explicit typing information. The transitions then insert the omitted events around the decoded TYPED CONTENT event.

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.

5.4.4 Schema Optimization Implementation

Our RELAX NG parser constructs an abstract syntax tree from its input RELAX NG schema. Our implementation then performs some of the simplifications specified by RELAX NG [66]; 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 [4] 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 RELAX 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. 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 what events to omit cannot be made fully when processing a piece of syntax.

An example of this is illustrated in Figure 5.2. 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.

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

Figure 5.2: Selecting whether to enter a subautomaton

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

Figure 5.3: A problematic use of the star construct

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 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. This choice was made because using states was simpler, but it was not sufficient for the more complex process required of the DOA construction.

However, using states as entry and exit points introduces the problem of chaining the subautomata. To solve this, we introduce 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.

As we mentioned before, repetition constructs also have some interesting points. An example is provided in Figure 5.3, 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 its ELEMENT END event, 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 evolv-

ability. 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.

5.4.5 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 more interesting 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 in the figures will also show whether their known or unknown entry and exit points are used. These are indicated with a k or a u at the point, 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 those of an optional construct on the DOA side with a q. These markings appear only where they are introduced in the construction.

We begin by showing the `element` construct in Figure 5.4. In this figure, as in all the rest, we shorten all event names, transition types, etc., to a single letter whose meaning should be clear from the context. We show both the normal case and the case where the subautomaton is an open 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 rarely produce events in the transitions. Apart from the `element` construct, only the construction processes for the `attribute` and `data` constructs produce events for transitions; the others may transform existing transitions, but will not produce new ones.

The next one we cover is the `group` construct in Figure 5.5. On the EOA

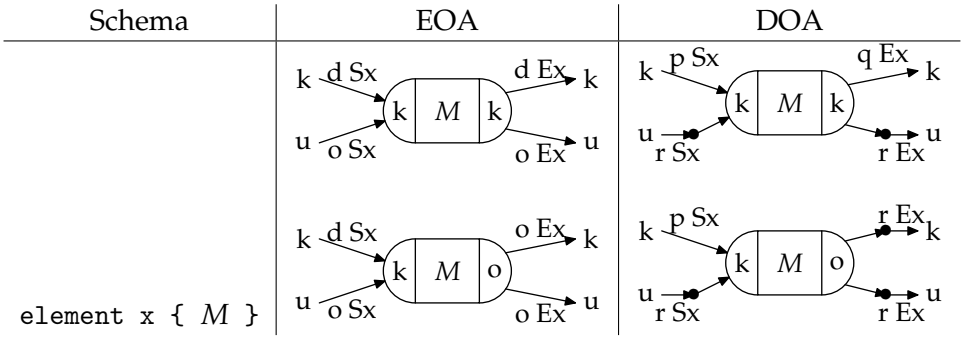


Figure 5.4: Subautomaton construction for `element`

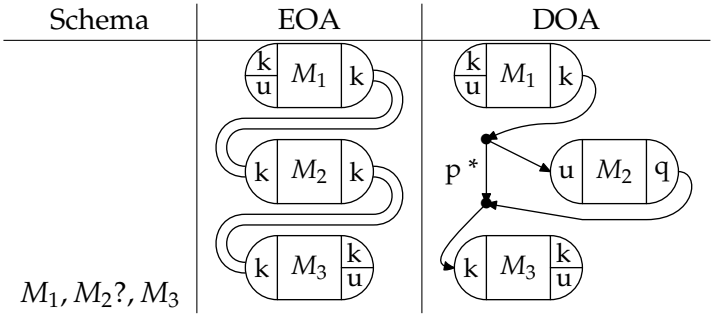


Figure 5.5: Subautomaton construction for `group`

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 has been marked optional. We do not show this in the EOA construction, but the result is that in the EOA M_2's entry and exit points would be 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 extremely complex. In our implementation it takes approximately 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

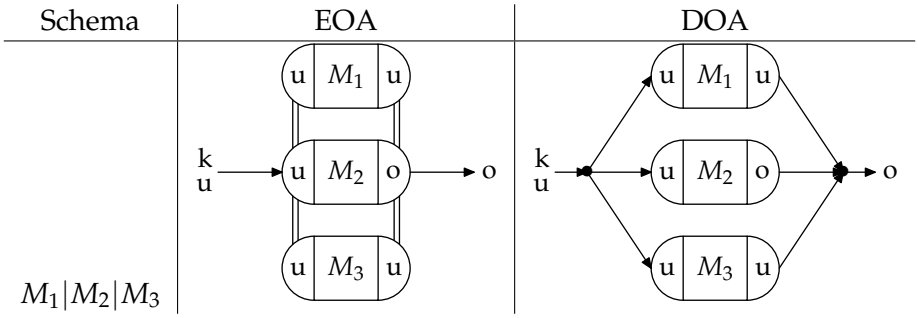


Figure 5.6: Subautomaton construction for `choice`

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 5.6. 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 respectively. 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 the DOA's 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.

Chapter 6

Message Transfer Protocol

Improving the processing of application messages helps only to the extent that the processing is a bottleneck of the system. A messaging system needs to consider also the protocol used for transferring messages. We noted in our measurements [51] 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.

Our implemented protocol is divided into two layers, the *Transfer layer* and the *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.

6.1 Basic Protocol Semantics

The basic purpose of the protocol is to be flexible enough to accommodate a variety of messaging styles. As noted in section 3.2, 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.

6.1.1 Protocol Requirements

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 on the service layer using SOAP headers, as is done in WS-Addressing [126]. There-

fore 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 desirable. 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.

6.1.2 The Transfer Layer

Our original message protocol implementation used Blocks Extensible Exchange Protocol (BEEP) [75] directly as its underlying protocol, since the capabilities of BEEP matched the above requirements well. In BEEP a *session* is opened between two peers. Such a session is then divided into *channels*, each of which can be opened from either side of the connection.

BEEP itself does not specify what transport protocol is used underneath, but the only standard mapping is on top of TCP [76], so that was what we used. At the time that we made our decision to use BEEP, there was quite a bit of interest in it, and also a standardized SOAP binding [73]. However, none of the available BEEP implementations reached release status, and interest in BEEP seems to have mostly waned. This is somewhat regrettable, since in our opinion BEEP is a well-designed protocol with many applications.

However, we could not use BEEP on mobile phones, as version 1.0 of the MIDP API, which we targeted, only supports HTTP for communication. Therefore we decided to implement the Transfer layer to provide BEEP-like semantics on top of various other protocols and to implement the more sophisticated features of our original protocol generically on top of this.

The message syntax of the Transfer layer is the same as that used by HTTP and BEEP, namely a header consisting of name-value pairs followed by an opaque body as shown in Figure 6.1. Note that the actual representation in the Transfer layer is abstract, and the format shown in the figure is simply the serialization format chosen by HTTP and BEEP. For simplicity

```
Content-Type: application/x-ebu+item
Content-Length: 1245
...
<body data>
```

Figure 6.1: The AMME message syntax

of implementation we map the Transfer layer headers to headers in HTTP and BEEP, but this is not a requirement of AMME. It is merely sufficient to specify how the headers are represented on lower layers.

On the Transfer layer communication happens over point-to-point connections. One party of a connection is always designated as the client and the other the server. This distinction has significance only during connection opening where the client is the party actively initiating the connection and the server is passively waiting for connection attempts.

The actual opening of a Transfer layer connection happens by the client initiating connectivity with the protocol under the Transfer layer. After this the client and server exchange a single request-response pair of messages. These messages do not contain any data, but may contain headers specified by higher layers to, e.g., negotiate parameters for the connection.

After a Transfer connection is established, messages can be sent by either party at any time. Transfer layer connections are divided into pipes; each message will be sent through a specified pipe. This provides multiplexing of connections for higher layers without requiring the opening of new connections. At the Transfer layer messages are strictly one-way and there is no acknowledgement mechanism.

6.1.3 Transfer Layer Mappings

We have implemented four different mappings of the Transfer layer. The underlying protocols and their source lines of code[1] are shown in Table 6.1. Code that is shared between all mappings comes to an additional 285 lines by the same measurement. Of these protocols, the TCP mapping is a very simple one that we built to have a Transfer layer implementation quickly, and the Bluetooth L2CAP mapping is a modified version of the TCP mapping to act more as a proof of concept than as something that gets used. We do not consider these two mappings further.

The Transfer layer is also responsible for interpreting the protocol features mentioned in section 3.2. Currently the only specified feature is called `enc`, and it indicates that the connection should be encrypted. For HTTP this is accomplished with SSL [29]. For TCP we implemented a Java inter-

[1] Measured using David Wheeler's `sloccount` tool

Table 6.1: Implemented Transfer layer mappings with code line counts

Protocol	Line count
BEEP	354
HTTP	612
TCP	280
L2CAP	270

face on top of the native Host Identity Protocol (HIP) API [53] (also developed in the Fuego Core project), and used that for encryption. BEEP also includes a native capability to use SSL, but our main interest was in providing the HIP API to Java applications, so we left the `enc` feature unimplemented for the BEEP mapping.

Since the Transfer layer design was inspired by BEEP, its mapping is very straightforward. A connection on the Transfer layer is mapped to a BEEP session and a pipe to a BEEP channel. An implication of this is that opening a new pipe in the Transfer layer requires a network round trip in this mapping. The mapping of the header-body structure is also straightforward, since BEEP uses the exact same semantics.

While BEEP uses a control channel, this control channel is not available to applications. Therefore a BEEP data channel needs to be opened to perform the AMME connection establishment. This channel is later repurposed to carry AMME messages.

The HTTP mapping is much more complex, since the basic semantics of HTTP is a single synchronous request-response pair, and the required semantics of the Transfer layer is continuous asynchronous one-way messaging in both directions.

The initial request-response pair is handled by the client issuing an HTTP GET request to a known URL handled by the server. The body of the server's response will contain a unique URL for this particular Transfer connection. HTTP headers in these messages are used to carry any metadata provided by higher layers.

For actual communication the client will use the unique URL provided by the server. In our implementation the client will now start a number of threads that will handle the messaging. On a phone the client uses two threads, on a desktop computer between 4 and 8. Half of these threads will be *token threads* and the other half will be *data threads*. The purpose of the token threads is to permit the server to send messages to the client, but they also act as a rudimentary flow control device.

Each token thread will begin its execution by sending an empty HTTP request, a *token message*, to the server. These messages contain a header that lets the server know they do not contain any data. The server will let these wait for a response. If the server needs to send a message to the client, it

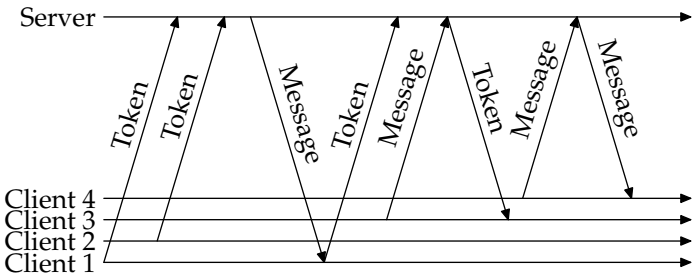


Figure 6.2: Token and data messages in HTTP Transfer mapping

will send it as a response to one of these waiting requests. This is similar to the PAOS reverse HTTP binding for SOAP [56], but our solution is a more general one.

The process is shown in Figure 6.2 with four client threads, the first two of which are token threads. The client begins by sending two token messages. Next, when the server has a message to send, it sends it as a response message to one of the tokens. The token thread receiving this response will pass the message to higher layers, and then resends a token message to the server. The next two cases illustrate the client sending an actual message. In the first case, the server has no data to send, so it responds with a token message, which the client knows to ignore. In the second case the server has data to send, so it can send it as a response to the client's message.

One consideration, which surfaced especially with mobile phones, was the time that a request could remain unanswered. HTTP client implementations will break the underlying TCP connection if the server does not respond sufficiently quickly; this time can be as low as 5–10 minutes on mobile phones. For this reason the server has a timeout, after which it will respond to a token message with a token message of its own. The client knows not to process this, but the token thread receiving it will resend its token, thereby resetting the timeout.

6.2 Extension Modules for AMME

The header-body split offers a way to extend AMME by defining new headers and their semantics. We implemented several such extension headers, which we divide into separate *modules*; each module supports certain behavior and specifies headers to achieve this. The main considerations in the extension modules were to improve the quite weak semantics of the Transfer layer and to provide supporting functionality for mobile clients.

As mentioned before, the split into Mobility and Transfer layers was made because we wished to utilize several different underlying protocols, but did not wish to implement essentially the same functionality for each.

The sensibility of this approach is also partially validated by noting that the amount of Mobility layer code, counting all the extensions described below, is 1197 lines, almost the same as is taken by our two main Transfer mappings, BEEP and HTTP.

The splitting of the extra functionality into independent modules is also beneficial because different underlying protocols can give different guarantees to the Transfer layer. If an underlying protocol provides some useful functionality, we can disable the module providing the same functionality when the Mobility layer is used in conjunction with that mapping.

Note that, for clarity, we provide readable header names for all of the modules below. In our actual implementation using these names would waste bandwidth needlessly, so the header names going over the network are only two characters long. Furthermore, any numbers or lists of numbers appearing in headers are encoded in a compact binary form.

6.2.1 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.

Use of individual acknowledgements typically indicates lost messages, so upon reception the receiver should resend all unacknowledged messages. However, we have noticed that especially the HTTP mapping with more than one data thread is prone to messages arriving out of order, so this should not be an immediate trigger.

The Mobility layer also passes all received messages to the application in the order of their sequence numbers. The service components of our messaging system preserve this order, thus giving applications a guarantee of ordered delivery. Furthermore, the messaging service 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.

6.2.2 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 this connection. If the client later wishes to continue this previous connection, it will send this identifier in the 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.

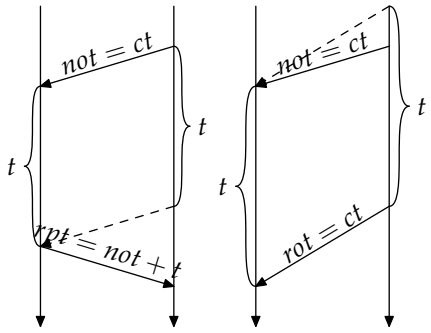
This feature is also useful to applications, for two reasons. The first is that applications, both at the client and server, will see a unique persistent identifier for any communicating peer, and can use this identifier instead of using an IP address, which will change when the other end is mobile. The second benefit is that we are able to retain Xebu state, specifically the tokenizations, across mobility, and do not need to rebuild it at every connection break.

We note that this module is not needed with the HTTP mapping, since the unique URL given at the initial request already provides a unique identifier. Furthermore, connection persistence, as implemented by this module, is mostly usable for connection-oriented Transfer layers which provide notices of disconnection to applications. As it stands, it is not meaningful to speak of an HTTP-based Transfer connection closing or breaking.

6.2.3 Message Compaction Modules

The Mobility layer also contains some 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 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-level 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



(a) Round-trip time computation (b) Timestamp update

Figure 6.3: Computing round trip times in AMME

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.

6.2.4 Measuring Round-Trip Time

The final module of the Mobility layer provides round-trip time measurements. 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 round-trip time.

The precise formulas used in calculating timestamps are

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

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 round-trip time.

A graphical demonstration of how these equations work to compute the round-trip time is given in Figure 6.3(a). Here we see the right side sending the original message at time $not = ct$. After time t has passed, the left side sends a message (this can be independent of the message sent by the right side), containing the PEER-TIMESTAMP value of $spt = not + t$. When this

message arrives at the right side, the time that has actually passed from *not* is t plus the round-trip time. Hence a subtraction of the received value from the current time gives the round-trip time.

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

Figure 6.3(b) shows how this works. The second message sent by the right side contains its current time in an OWN-TIMESTAMP header. The left side 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.

Chapter 7

Experimental Results

During the course of this work we have continually run experiments on the system. In addition to writing examples and test cases for nearly all of the functionality we have performed extensive performance measurements to determine how usable the system is in our target environment. Naturally several of these measurements have been performed on mobile phones in real network conditions.

The experiments that we have performed consist of both experiments on individual components of the MTS as well as experiments on the whole system. Many of these measurements have been published elsewhere in some form [47, 48, 50], but the below exposition should provide more detail.

7.1 Experimental Platforms and Data

For our measurements we had several different machines available. This partly reflects the fact that the measurements were performed over a long period of time, during which our available computing systems were upgraded. We provide the names and characteristics of each of the platforms in Table 7.1. We ran the Java Virtual Machines (JVMs) mostly at default settings, but for some experiments needed to increase the maximum heap size.

The different networks that we measured and the machines on each network are shown in Table 7.2. All network experiments terminated with Beagle being the server. We also show ICMP round-trip times (measured with the `ping` program and shown in its minimum/average/maximum format) and hop counts from the client machine to Beagle (measured with the `traceroute` program).

For the most part we are interested in the speed of the components that we measure. However, as we noted in section 2.3, memory consumption is also an issue, so for XML processing we include measurements of the

Table 7.1: The platforms used in the experiments

Platform	Description
Beagle	Desktop PC, 1333 MHz AMD Athlon processor, 512 MiB of main memory, operating system Debian GNU/Linux 3.1, Sun Java 2 SDK 1.4.2
Clement	HP Omnibook laptop, 500 MHz Intel Pentium III processor, 512 MiB of main memory, operating system Debian GNU/-Linux 3.1, Sun Java 2 SDK 1.4.2
Mekong	IBM ThinkPad R51 laptop, 1.6 GHz Intel Pentium M processor, 1 GiB of main memory, operating system Debian GNU/Linux 3.1, Sun Java 2 SDK 1.4.2
Lugburz	Desktop PC, 3 GHz Intel Pentium 4 processor, 1 GiB of main memory, operating system Debian GNU/Linux 3.1, Sun Java 2 SDK 1.5.0
3660	A Nokia 3660 model mobile phone supporting MIDP 1.0
7610	A Nokia 7610 model mobile phone supporting MIDP 2.0

Table 7.2: Networks used in experiments

Network	RTT (ms)	Hops	Machines
LAN	0.1/0.1/0.2	1	Beagle, Clement
WLAN	2.8/3.7/21.1	5	Clement
GPRS	690/830/1330	12	Clement, 3660

amount of memory that is spent in total during the processing. For networking experiments we measure the amount of data that is transmitted over the network, as that is one of the most crucial pieces of information for messaging applications.

For XML serialization and parsing experiments we collected three different data sets from different components of the Fuego middleware platform, as shown in Table 7.3. The Flood and Event sets are intended to reflect the expected use of the messaging system while the Syxaw set is for testing whether the system works for large XML documents. The Event-C set is one for which we have a complete schema available, and we use that only for a part of the Xebu experiments.

All of these data sets exist as individual XML files in the file system. In the experiments we load all files into memory. For the experiments on Xebu we also parse them first into XAS event sequences and use these event sequences as input to the serialize-parse cycle of the experiments. Everything is always done inside memory for these experiments; no I/O time is included in the measurements.

We also need to eliminate various anomalies caused by the JVM. The first of these is just-in-time (JIT) compilation, which is eliminated by run-

Table 7.3: The data sets for XML processing experiments

Name	Amount	Size (B)	Origin
Flood	2000	1874168	The flood example application for the messaging system
Event	2647	5821016	The restaurant example application for the event system
Event-C	698	2117437	The notifications of the Event data set
Syxaw	1	13223476	A large example directory hierarchy from the Syxaw file system

Table 7.4: The APIs in the XAS measurements

Format	Description
SAX	The Xerces parser using the SAX API
DOM	The Xerces parser using the DOM API
XAS	The regular XAS API with the kXML parser
XASSAX	The SAX compatibility API of XAS
XASDOM	The DOM compatibility API of XAS

ning a long enough loop of the experiment before starting the measurement, so that no JIT compilation happens during the actual measurement phase. The length of this loop was determined experimentally. The second issue is garbage collection. We force this to happen at selected points in the execution where timing is turned off so that it does not interfere with the measurement itself. We also compensate for garbage collection in our memory consumption measurements by including collected memory in our figures. In memory measurements we follow recognized best practices [77].

7.2 Indicative Measurements of the XAS API

We performed some measurements of the XAS API, mostly to make sure it was not unacceptably slow. We do not consider these measurements to show anything, just to give rough indications of performance. They may be useful in conjunction with the measurements of section 7.3 to show some idea of the differences between XML parsers.

The tests for the Flood and Event data sets were made on Beagle with the maximum heap size increased to 256 MiB. The test on the Syxaw data set was run on Mekong with the maximum heap size increased to 768 MiB, as the memory ran out with smaller heap sizes. We used a total of five different APIs, all described in Table 7.4. The SAX experiments discarded the results whereas all the other experiments left their results in memory.

Table 7.5: XAS processing measurements

Processor	Time (ms)	Memory (MB)	Left-over (MB)
	Flood		
SAX	4860, 5370	176, 198	0.2, 0.2
DOM	9880, 7820	315, 328	138, 63
XAS	1590, 1710	50, 52	46, 46
XASSAX	710, 1020	58, 84	0.2, 0.2
XASDOM	5700, 5970	220, 234	14, 14
	Event		
SAX	6860, 7910	261, 309	0.3, 0.3
DOM	12960, 12720	446, 491	190, 107
XAS	3810, 4420	105, 107	84, 84
XASSAX	1970, 2790	136, 185	0.3, 0.3
XASDOM	11680, 12500	401, 424	52, 52
	Syxaw		
SAX	1700, 3240	2.6, 2.9	0.002, 0.002
DOM	3730, 7320	88, 131	85, 117
XAS	4960, 6180	105, 105	102, 102
XASSAX	5380, 6720	107, 110	0.003, 0.003
XASDOM	11440, 12740	154, 154	119, 119

The processing in each case was the same. The XML document was parsed as XML from memory, and then optionally serialized. The complete results are shown in Table 7.5. Each column shows two figures, the first one with just parsing and the second one with serialization included. The memory measurement refers to total memory used during the experiment and the left-over measurement to the memory that is still spent after a full garbage collection.

We note that for XAS the total processing time appears to be directly dependent on the size of the processing and not at all on the number of documents. In contrast, Xerces does much better with the large single Syxaw document than with the many small Flood and Event documents. This is indicative of a large startup cost for the Xerces parser.

We also note an anomaly in the DOM results: adding serialization decreases the time that is spent and the memory that persists for the sets of small documents. While the former is not explainable, a cause for the latter could be that during document traversal the DOM tree is also optimized in some manner by the serializer. This is a likely explanation as the experiment without serialization does not traverse the resulting data structure.

Finally, we note that with the DOM compatibility API of XAS we get much smaller data structure sizes than with Xerces DOM, as indicated by the amount of left-over memory with the sets of small documents. We

Table 7.6: Formats for the Xebu experiments

Format	Description
Xerces	XML with the Xerces SAX parser
kXML	XML with the kXML parser and XAS API
FI	Fast Infoset with the SAX API
Xebu	Xebu with the XAS API

did not investigate the differences in the DOM trees built by the two approaches, but we suspect that the Xerces DOM building is much more sophisticated. However, the XAS compatibility API does build a correct DOM tree for XML as we have verified with extensive test data, so the difference is not caused by XAS omitting some necessary information.

7.3 Xebu Performance

Xebu performance testing was done on the Lugburz machine and on the two mobile phones. Here we used only the Event data set, as we felt that to be the closest to real-world data. Furthermore, we had a complete schema available for the Event-C data set, which we could use to construct a COA and test that.

We measured several different implementations, which we refer to as formats, that are given in Table 7.6. In addition, we suffix a format with Z to indicate the use of gzip on top of the serialized form. The Xebu measurements also use two other suffixes. F indicates *forgetful* processing, i.e., the token mappings are not preserved from one document to the next. The Fast Infoset was used only in a forgetful mode. The S suffix for Xebu indicates that the COA was used.

We collect the performance measurements made on Lugburz for all the formats in Table 7.7. This table shows the final document sizes, the times and memory spent in processing, as well as the throughput in messages per second for each format. All values are average values for a single document. Sizes are shown both as absolute values and as a percentage of the XML document size. We do not show error ranges, as the memory consumption did not vary at all, and timing deviations were all between 0.01 and 0.02 ms.

We also measured data binding speed for all the formats, i.e., the time that was taken to process primitive typed data, but there was no difference between Xebu's binary representation and the text representation of XML. This is because all typed data in these documents consisted of strings and small integers, for which there is little performance difference between text and binary. A quick experiment confirmed that for date and floating point values the binary encoding of Xebu is approximately 2–3 times faster than

Table 7.7: Performance of XML serialization formats

Format	Size (B)	Size (%)	Serialization Time (ms)	Serialization Thr (1/s)	Serialization Mem (kB)	Parsing Time (ms)	Parsing Thr (1/s)	Parsing Mem (kB)
Xerces	3033	100.0	0.45	2201	39.77	0.76	1315	113.23
XercesZ	675	22.3	0.74	1356	42.45	0.87	1144	114.22
FI	1689	55.7	0.35	2825	13.88	0.48	2072	37.63
FIZ	687	22.7	0.62	1608	17.67	0.55	1833	39.11
kXML	3033	100.0	1.06	941	141.23	0.93	1078	73.69
kXMLZ	674	22.2	1.36	736	141.45	0.95	1050	74.79
XebuF	1304	43.0	0.53	1874	55.51	0.83	1198	58.48
XebuFZ	695	22.9	0.93	1079	56.04	0.91	1094	60.14
Xebu	807	26.6	0.45	2230	38.63	0.76	1309	50.87
XebuZ	390	12.9	0.64	1565	41.20	0.82	1219	52.26
XebuFS	493	16.3	0.56	1794	38.92	0.79	1264	55.78
XebuFSZ	312	10.3	0.61	1644	43.68	0.80	1254	59.68
XebuS	493	16.3	0.42	2357	34.52	0.77	1299	47.13
XebuSZ	312	10.3	0.55	1809	34.96	0.80	1251	48.76

the text encoding of XML.

The results indicate that Xebu achieves quite a good message size, even in forgetful mode when compared to Fast Infoset. However, while Xebu clearly defeats the kXML implementation in speed and memory use, Xerces achieves performance similar to Xebu. Fast Infoset is markedly better than Xerces or Xebu in these figures. We would expect that the Fast Infoset implementation has seen much more optimization work than our Xebu implementation.

The results after applying gzip are interesting. In this case when comparing between the forgetful binary formats and XML we see that after compression there is little difference in size. As the tokenization is fundamentally a similar operation to that performed by gzip, this is to be expected. The results for regular and COA-using Xebu versions are obviously better, as both of these formats remove additional redundancy from the documents before gzip sees them.

We also note that there is no difference in document size between forgetful and regular Xebu when schema optimizations are used. As the schema optimizations include pre-tokenization of strings appearing in the schema, this is clear. Namely, we have the complete schema available, so there will not appear any unknown names in the documents. In fact, some of our experiments indicate that we could also turn the dynamic tokenization of Xebu completely off in this case without it affecting the document size.

We also measured on both of our phones. As the Xerces and Fast Infoset implementations are written for Java Standard Edition, we could only use

Table 7.8: Performance of XML serialization formats on mobile phones

Format	Serialization Time (ms)	Thr (1/s)	Mem (kB)	Parsing Time (ms)	Thr (1/s)	Mem (kB)
7610						
kXML	56.0± 3.1	17.9	23.2±3.3	134.0± 5.5	7.5	50.5±2.8
Xebu	60.0± 5.1	16.7	33.1±2.2	134.1± 4.4	7.5	50.9±3.5
XebuS	45.0± 4.1	22.2	25.5±2.0	123.7± 6.7	8.1	49.4±5.1
3660						
kXML	250.0± 6.4	4.0	8.1±0.3	527.5± 2.9	1.9	29.3±3.0
Xebu	211.1±15.6	4.7	22.9±2.5	503.8±43.9	2.0	30.7±4.1
XebuS	159.5± 7.6	6.3	18.6±0.3	406.9±11.9	2.5	25.7±0.3

kXML and Xebu. All of Xebu's schema optimizations are also available for the MIDP platform. The results for the phones are presented in Table 7.8. Sizes are the same as in Table 7.7, so they are omitted, but we observed more fluctuation in the results in this case, so we include error limits at one standard deviation.

We note that in this case the kXML implementation performs significantly better. As profiling support on the MIDP platform is not at all good, we cannot offer precise causes, but simply note that the JVMs clearly differ between the desktop and the phones. Otherwise the results are consistent with those of Table 7.7.

As we noted in section 2.3, one concern stemming from the available memory on the mobile devices is application footprint. As we expect XML processing to be an integral component of future messaging, the footprint of the processor implementation needs to be very small. We therefore measured the footprint of each implementation by adding together the sizes of all the classes of that implementation that were loaded into memory during a single run of the experiment.

The footprints are shown in Table 7.9. The Foot column gives the normal footprint measured on the desktop experiment. For the formats usable on mobile phones, we also obfuscated the implementation with Proguard[1], which would be done in real deployment. We then added together the sizes of the same classes that we did without obfuscation.

In addition to the actual classes, we also include the size of data encoding and decoding code that we wrote for the non-Xebu formats. As Xebu includes this functionality as an integral part, including the size of that code for all implementations makes the measurement more realistic. For the other formats this code comes to 7.6 kB normally and 4.8 kB in obfuscated form.

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

Table 7.9: Footprints of XML serialization format implementations

Format	Foot (kB)	Obf (kB)
Xerces	510.5	–
kXML	36.7	21.1
FI	199.4	–
Xebu	52.1	22.0
XebuS	87.7	44.3

From the footprints we can see that Xerces is completely unsuitable for mobile devices, even if the implementation could be rewritten for MIDP. The efficiency of Fast Infoset appears to come at the cost of a very complex implementation; much of this may be spent for general ASN.1 processing, but we did not investigate the implementation closer.

Xebu's footprint, especially when obfuscated, approaches that of the kXML implementation, indicating that Xebu is also suitable for mobile devices. For the schema-based optimization we note that our current implementation builds different Java classes to implement the COA for each different schema. The COA could also be implemented generically, and we estimate that such a generic implementation would take approximately 10 kB (5 kB obfuscated), with the automata of our measurements requiring 10 kB of dynamic memory during execution.

7.4 AMME Functionality

There is little that can be tested of AMME performance, but it is possible to verify various pieces of functionality. AMME guarantees ordered delivery of messages, including ordered delivery of response messages, and provides resending of messages for reliability. In addition, AMME includes a new method for round-trip time estimation that does not require an actual round trip to be performed. These features can all be tested.

Our test application consisted of a client and a server where the client periodically sends a message to the server and processes the response asynchronously. We ran the server on Beagle and the client on Clement, using all three available networks, LAN, WLAN, and GPRS.

The reliability guarantee was tested by implementing a new Transfer layer that dropped messages randomly. Since this is acceptable behavior for the Transfer layer, the reliability module of the Mobility layer is expected to cope with it. Indeed, we verified that all messages were eventually received by the server in this case.

For testing ordered delivery it was sufficient to use the HTTP mapping with a 0-second delay between messages, as due to multithreading in the client the Transfer layer can in this case deliver messages out of order. This

Table 7.10: Actual and AMME-measured round-trip times

Conn	Client AMME	Client BEEP	Server AMME	Server BEEP
LAN	25.1±8.1	39.7±4.6	26.0±12.0	47.6±5.1
WLAN	25.0±10.9	48.6±5.0	26.9±14.0	51.1±10.1
GPRS	2628.4±508.7	2675.2±536.9	2591.6±288.7	2697.5±358.2

will especially be the case if different TCP connections are used for different HTTP requests, which will be the case unless both ends support persistent connections and pipelining. We verified this out-of-order delivery to happen at the Transfer layer by observing network traffic directly. The Mobility layer performed the correct reordering in all cases.

Finally, for the round-trip time estimation we set the client's delay between messages to be sufficiently large so that only a single message would be in transit at a time. We inserted code in the BEEP mapping implementation to measure actual round-trip times by printing timestamps at message sending and acknowledgement times. As the BEEP mapping acknowledges each message immediately, this will provide accurate results.

The results we got are shown in Table 7.10 with mean times and standard deviations, both from the client and the server side. As the BEEP protocol is symmetric, we could get these measurements from both sides. The first measurement was excluded as that included the time to set up the BEEP connection, and was therefore several times as large as the other measurements.

We can see that AMME's measurement is apparently an accurate way of getting round-trip times. The figures are consistently lower than those measured by BEEP. This is explained by Figure 6.3(a), which shows that AMME's measurement only takes into account the time taken on the network. The timestamp difference method that we used with BEEP also includes the remote processing to send the acknowledgement. We verified by observation that this remote processing took approximately the difference that is shown in Table 7.10.

7.5 General Messaging Performance

The test scenario that we used for the full MTS performance test consisted of a client on Clement and a server on Beagle. Again, we used all of the three available networks. We compared the BEEP mapping of AMME against regular Apache Axis, and Apache Axis using persistent HTTP connections, as shown in Table 7.11.

In our scenario the client sends messages as quickly as it can. A single message consists of a string, a date, and a floating point value that all remain constant throughout the test, and a sequence number that increases

Table 7.11: Protocols of the MTS experiments

Protocol	Description
HTTP	The default HTTP 1.0 shipped with Apache Axis using XML with synchronous invocations
PHTTP	A version of HTTP 1.0 hacked to keep connections persistent using XML with synchronous invocations
AMME	The MTS with the BEEP mapping of AMME using Xebu with asynchronous invocations

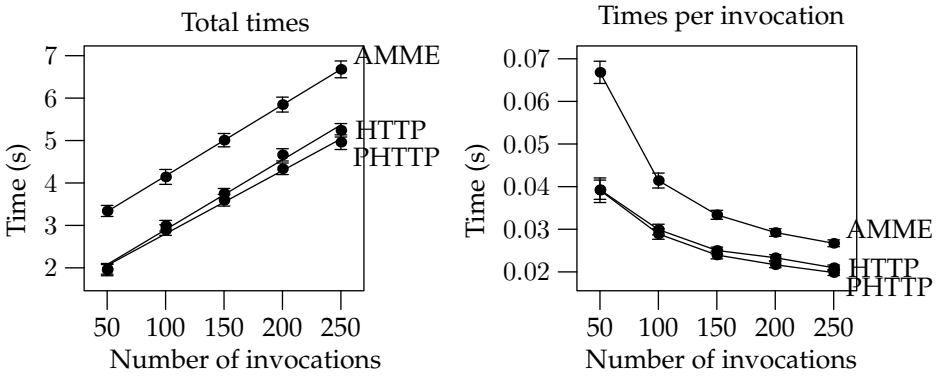


Figure 7.1: Per-invocation times over the LAN connection

for each message. The server verifies that it receives the messages in sequence and responds with the sequence number. The number of messages that we used varied between 50 and 250.

The first results in Figures 7.1, 7.2, and 7.3 show the total time for the experiment as well as the total time divided by the number of invocations for each of the three networks. The total times are drawn as regression lines whereas for the per-invocation times the average values are connected. In both cases, error bars are shown at one standard deviation.

The MTS has a significant overhead over the LAN connection when compared to Axis. As the implementation of the MTS we used was a wrapper around Axis, this is as expected. However, even on the WLAN connection we are starting to see the benefits of asynchronous invocations when AMME catches up to even PHTTP, and would for a larger number of messages surpass it.

The deviation in the times for HTTP over WLAN is very large. We note that a lost TCP SYN segment will, with the implementation in Linux, cause a 1.5-second timeout before retrying. As the HTTP implementation needs to constantly open new connections, the likelihood of this happening grows. With the small latencies of the WLAN, such delays effect a large variation in the results.

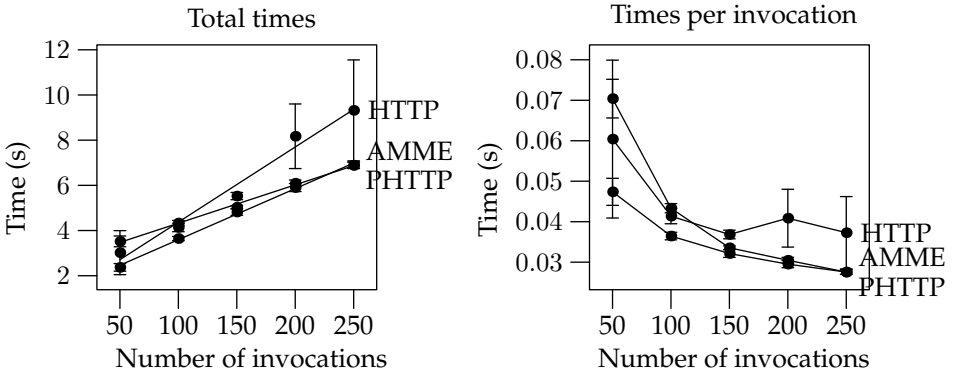


Figure 7.2: Per-invocation times over the WLAN connection

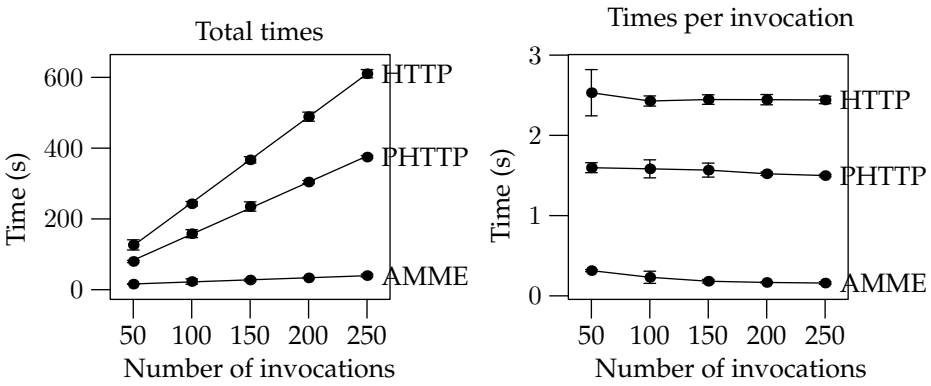


Figure 7.3: Per-invocation times over the GPRS connection

In the measurements over GPRS the benefits of asynchronous invocations materialize most clearly. Here the dominant factor in the timing is network latency. As the MTS uses both asynchronous invocations and the message bundling of AMME, the effects of network latency are much smaller for it. Comparing the two HTTP protocols we see that one network round trip is spent by plain HTTP to open each connection.

We also measured the total amount of data sent in the 250-message experiment, shown in Figure 7.4. The data is split into three parts, the application data (indicated as XML), the data used by the application protocol (HTTP or AMME), and finally TCP segments containing no application data.

Clearly the amount of XML data sent by AMME is much lower, since it uses Xebu instead of XML. We also note that the amount of protocol overhead for AMME is much lower than for the two HTTP protocols. The two HTTP protocols send approximately the same amount of application data, but the constant opening of new TCP connections makes HTTP send significantly more data in total.

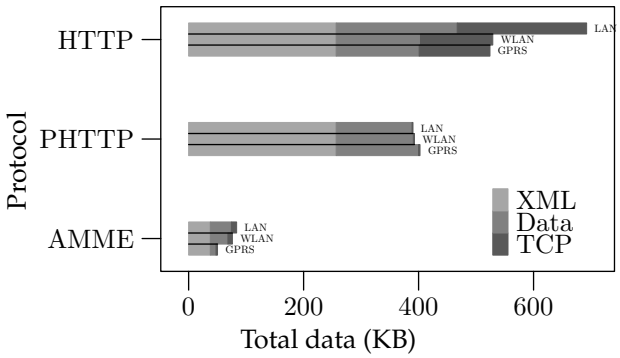


Figure 7.4: Amounts of total data sent

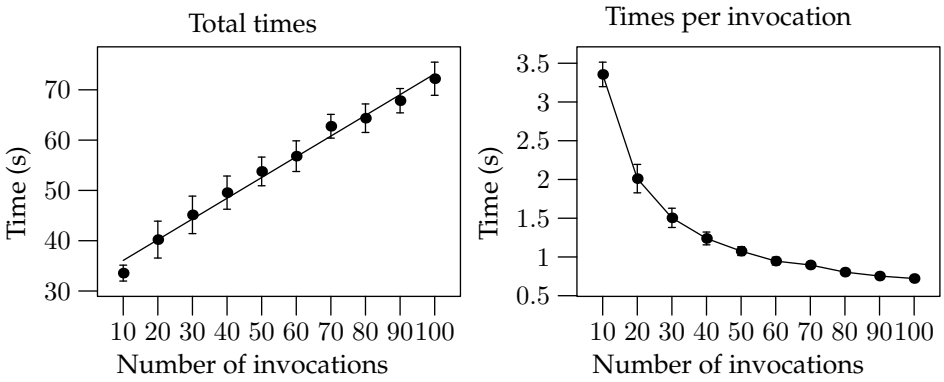


Figure 7.5: Per-invocation times using a mobile phone

We can see that the amount of overhead caused by the AMME protocol in this scenario is only approximately 25 % of the total data sent, even at its highest over the LAN. This appears to be similar to the overhead caused by HTTP for XML. However, taking absolute figures, we can see that Xebu combined with HTTP would make the protocol overhead to be approximately 75 % of the total message size. This therefore provides validation for our observation in section 3.1 that a binary format for XML is not sufficient on its own.

Finally, we ran a similar scenario with the 3660 phone as the client using the HTTP mapping of AMME. There is no comparison point for these measurements, so Figure 7.5 shows only the total time and time per invocation for this single case. Note that we used a fewer number of messages for this experiment.

By extrapolating the figure for total times we can see that the overhead of the experiment is approximately 30 seconds. As we started each experiment from a clean slate, this includes all network setup that was required for the phone, and is therefore understandable. If network connectivity had

been available from the start, this delay would have been much smaller.

The average time per invocation appears to settle to around 0.5 seconds, which is much higher than it was for AMME in Figure 7.3. This is explained by the use of the HTTP mapping with its thread-based flow control. Since in our phone implementation only one message can be in transit in either direction at the same time and message size is bounded, the largeness of this figure is also explained.

Chapter 8

Conclusions

We have used the MTS described here as a component of the mobile middleware platform built by the Fuego Core project. While we consider it usable for basic XML messaging, further experience with real-world scenarios has revealed certain areas of improvement. We first describe what we have found useful and then consider some simple enhancements of the current system as well as larger areas of future work.

8.1 Useful Ideas

The selection of Java as the programming language was useful in getting the system to work on mobile phones without too much effort. The similarity of the language and especially of its use on all platforms was a crucial enabler for this. Frequent compilation for the phone quickly revealed any code which accidentally used features not available on the MIDP platform. We intend to continue using Java in our future work on the system.

We believe that our decision to make the basic messaging interfaces asynchronous to be a correct one. As we noted in section 3.1, the latencies involved in wireless networks make mandatory synchronous programming infeasible. In addition, as we noted in section 3.2, the callback style makes it simple to implement different MEPs. However, so far we have little experience in advanced uses of these asynchronous APIs, so final evaluation needs to be postponed.

The pull-style sequence-based interface for XML processing appears to be more natural than the alternatives. After all, this kind of interface is essentially what programmers have always been using for serialization of structured data, though more typically with byte- or character-based output. The usage experiences reported in [47] support this view.

Based on the measurements in section 7.3 we note that while Xebu achieves the best results only in document size, its other qualities are well balanced. Both Xerces and Fast Infoset have very large implementations,

and the small kXML uses more time and dynamic space than any of the others. Therefore Xebu can be said to be a very good fit for the mobile environment.

Compared to other existing binary XML formats, Xebu has been designed to be more loosely coupled, in a sense. The two main reasons for claiming this are the explicit presence of assigned tokens in the serialized form and the default transitions of the COA. The explicit presence of tokens permits us to keep the space of available tokens limited without needing to specify any token replacement policy.

The default transitions in the COA were designed to accommodate certain schema changes. At least in certain cases it is possible to add elements or attributes not present in the schema, and the definition and our construction would allow slightly more of this than our current implementation. This is in contrast to other formats which require explicit preparation for extensibility when designing the schema. However, we have not performed a formal analysis of the allowable schema extensions, and suspect an exact analysis is not even feasible.

Message transfer protocols have not been our focus in this research. We especially have not considered ad hoc communication or multicasting, both of which have come up as potential enhancements. In our view, the peer-to-peer model of BEEP is better suited to this environment than the request-response style of HTTP, but so far we do not have sufficient experience to give definite conclusions.

8.2 Proposed Enhancements

While the use of Java permitted the same implementation to be used on both desktop machines and mobile phones, this was not without its downside. The implementation was mostly tested on desktop-class machines, so the mobile phone platform was not given the attention that it deserved. One indication of this are the measurements in section 7.3 where the performance of kXML was clearly lower than Xebu's on the desktop but comparable on mobile phones.

We still consider the interfaces of the system to be a good fit for the phones too. However, the implementation, especially the XAS system, is perhaps too massive and split too finely to be most efficiently usable on the phone. As mentioned in [63], good application design and suitability for mobile phones may be at odds with each other.

While the XAS API served its purpose well, it turned out that its intended purpose had a much smaller scope than its eventual requirements. We noted in [47] that extending XAS with an indexing scheme to provide tree-like handling could make it competitive with DOM, and are currently working on such a scheme. Furthermore, messaging applications seem to

need much better handling of XML fragments, an area where DOM dominates. As we do not consider DOM-like APIs to be suitable, we will need to extend XAS to provide these capabilities.

The Xebu format itself appears to be acceptable. Our measurements indicate that there are still some performance improvements to be made, but we do not believe these to be infeasible. We might consider extensions to the schema optimization to make it handle more cases (e.g., some improved handling of choice and interleave, as well as considerations of recursive elements), and it would also be useful to chart exactly how the schema can be extended without breaking the COA.

The protocol layer we do not see as needing much improvement. We intend to write a light-weight version of it to better work on mobile phones. Furthermore, it might be useful to reconsider addressing, since a messaging target may be able to use several different underlying protocols (e.g., WLAN, GPRS, Bluetooth), and it might be useful to provide a near-transparent way of selecting the most appropriate one.

8.3 Future Work

Our future work has three main directions. First, while low-level API compatibility with XML has proved beneficial in integrating a binary format into the XML stack, the required string processing is still a source of inefficiency. Providing a way for binary-aware applications to use the XAS API to directly access the tokens could give a performance boost. However, this will most likely complicate the API significantly, and needs to be evaluated carefully.

A major topic for all binary XML formats are security features such as XML Encryption [113] and Signatures [114]. Since these rely directly on the serialized form for interoperability, API compatibility does not help. As secure messaging will likely be important in the future, it would not be acceptable to require XML there and leave binary XML only for the non-secure uses.

Security processing will also require a way of handling XML documents as trees and of processing XML fragments. As we noted above, the current XAS API is not suited for this type of work. However, we believe that it is possible to extend XAS to cover these cases while still retaining the efficient sequence-based processing model.

Finally, the Fuego Core project has done work on efficient content-based routing [93], but this work has focused on simple filters. In XML messaging content-based routing is typically handled using the much more complex XPath. The concept of matching several XPath expressions against the same XML document simultaneously has received much attention [2, 21], but these systems are limited in the kinds of XPath expressions that can

be handled. Furthermore, the propagation of filters and the necessity of covering optimizations shown in [93] are not addressed at all.

As a final statement, our experience with the MTS suggests that XML messaging is not incompatible with mobile devices. While we have identified several issues with our current implementation above, none of these appear to be fundamental problems. Rather, they are specific to our implementation, and are typical of application development where the full requirements are revealed only after a system has seen actual use. Our future work should address all of these concerns in a manner that provides an efficient XML-based messaging system for the needs of future communication applications.

Bibliography

[1] Bob Aiken, John Strassner, Brian E. Carpenter, Ian Foster, Clifford Lynch, Joe Mambretti, Reagan Moore, and Benjamin Teitelbaum. *RFC 2768: Network Policy and Services: A Report of a Workshop on Middleware*. Internet Engineering Task Force, February 2000. (Cited on page 1.)

[2] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *26th International Conference on Very Large Data Bases*, pages 53–64. Morgan Kaufmann Publishers, September 2000. (Cited on page 83.)

[3] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, United Kingdom, 1998. (Cited on page 49.)

[4] Lex Augusteijn. Sorting morphisms. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, Heidelberg, Germany, September 1998. (Cited on page 52.)

[5] Olivier Avaro and Philippe Salembier. MPEG-7 systems: Overview. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(6):760–764, June 2001. (Cited on page 47.)

[6] Matthew E. Bayer. Analysis of binary XML suitability for NATO tactical messaging. Master's thesis, Naval Postgraduate School, Monterey, California, USA, September 2005. (Cited on page 46.)

[7] BEA Systems Inc., San Jose, California, USA. *JSR 173: Streaming API for XML*, October 2003. (Cited on page 30.)

[8] Bluetooth SIG. *Specification of the Bluetooth System, Core Package version 2.0*, November 2004. (Cited on page 17.)

[9] David Brownell. *SAX2*. O'Reilly, Sebastopol, California, USA, January 2002. (Cited on pages 10, 24, and 29.)

[10] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-05, Hong Kong University of Science and Technology, April 2001. (Cited on page 9.)

[11] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Systems Research Center, Digital Equipment Corporation, May 1994. (Cited on page 40.)

[12] Jian Cai and David J. Goodman. General packet radio service in GSM. *IEEE Communications Magazine*, 35(10):122–131, October 1997. (Cited on page 17.)

[13] Stefano Campadello. *Middleware Infrastructure for Distributed Mobile Applications*. PhD thesis, University of Helsinki, Department of Computer Science, Helsinki, Finland, April 2003. http://ethesis.helsinki.fi/julkaisut/mat/tieto/vk/campadello/. (Cited on page 2.)

[14] Stephen L. Casner and Van Jacobson. *RFC 2508: Compressing IP/UDP/RTP Headers for Low-Speed Serial Links*. Internet Engineering Task Force, February 1999. http://www.ietf.org/rfc/rfc2508.txt. (Cited on page 41.)

[15] James Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Data Compression Conference*, pages 163–172, March 2001. (Cited on page 40.)

[16] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the limits of SOAP performance for scientific computing. In *11th IEEE Symposium on High Performance Distributed Computing*, pages 246–254, July 2002. (Cited on page 19.)

[17] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984. (Cited on page 40.)

[18] Michael Cokus and Daniel Winkowski. XML sizing and compression study for military wireless data. In *XML Conference and Exposition*, Baltimore, USA, December 2002. (Cited on page 47.)

[19] Dan Davis and Manish Parashar. Latency performance of SOAP implementations. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 377–382, May 2002. (Cited on page 19.)

[20] L. Peter Deutsch. *RFC 1952: GZIP File Format Specification Version 4.3*. Internet Engineering Task Force, May 1996. http://www.ietf.org/rfc/rfc1952.txt. (Cited on pages 39 and 42.)

[21] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems*, 28(4):467–516, December 2003. (Cited on page 83.)

[22] Robert Elfwing, Ulf Paulsson, and Lars Lundberg. Performance of SOAP in Web service environment compared to CORBA. In *Ninth Asia-Pacific Software Engineering Conference*, pages 84–93, December 2002. (Cited on page 19.)

[23] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. (Cited on page 12.)

[24] Roy Fielding, James Gettys, Jeffrey Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul Leach, and Tim Berners-Lee. *RFC 2616: Hypertext Transfer Protocol — HTTP/1.1*. Internet Engineering Task Force, June 1999. http://www.ietf.org/rfc/rfc2616.txt. (Cited on page 2.)

[25] Jeroen Fokker. Functional parsers. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, Heidelberg, Germany, May 1995. (Cited on page 49.)

[26] Fabio Forno and Peter Saint-Andre. *JEP-0072: SOAP Over XMPP*. Jabber Software Foundation, October 2005. http://www.jabber.org/jeps/jep-0072.html. (Cited on page 14.)

[27] Ned Freed and Nathaniel Borenstein. *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. Internet Engineering Task Force, November 1996. http://www.ietf.org/rfc/rfc2045.txt. (Cited on page 14.)

[28] Ned Freed and Nathaniel Borenstein. *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. Internet Engineering Task Force, November 1996. http://www.ietf.org/rfc/rfc2046.txt. (Cited on page 14.)

[29] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The SSL Protocol Version 3.0*. Netscape Communications, November 1996. http://wp.netscape.com/eng/ssl3/draft302.txt. (Cited on pages 15 and 59.)

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, Massachusetts, USA, 1995. (Cited on page 33.)

[31] Marc Girardot and Neel Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. In *Ninth International World Wide Web Conference*, May 2000. http://www9.org/w9cdrom/154/154.html. (Cited on pages 24 and 43.)

[32] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Boston, Massachusetts, USA, 3rd edition, June 2005. (Cited on page 18.)

[33] Robert Halstead, Jr. New ideas in parallel Lisp: Language design, implementation. In Takayasu Ito and Robert Halstead, Jr., editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 2–57. Springer-Verlag, Heidelberg, Germany, October 1990. (Cited on page 27.)

[34] Richard Harrison. *Symbian OS C++ for Mobile Phones Volume 1*. Symbian Press, April 2003. (Cited on page 18.)

[35] Leping Huang, Hongyuan Chen, T. V. L. N. Sivakumar, Tsuyoshi Kashima, and Kaoru Sezaki. Impact of topology on Bluetooth scatternet. *Journal of Pervasive Computing and Communications*, 1(2):123–134, June 2005. (Cited on page 17.)

[36] IBM. *MQSeries Everyplace for Multiplatforms Version 1, Release 2*, 2002. (White paper), http://www-3.ibm.com/software/ts/mqseries/everyplace/v12/whitepaper.html. (Cited on page 1.)

[37] Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, USA. *IEEE Std 802.11 — Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, March 1999. (Cited on page 17.)

[38] International Organization for Standardization, Geneva, Switzerland. *ISO 8879:1986. Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*, 1986. (Cited on page 5.)

[39] International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. *Public-key and attribute certificate frameworks*, March 2000. ITU-T Rec. X.509. (Cited on page 16.)

[40] International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. *Abstract Syntax Notation One (ASN.1) Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, 2002. ITU-T Rec. X.690. (Cited on page 43.)

[41] International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. *Abstract Syntax Notation One (ASN.1) Specification of Basic Notation*, 2002. ITU-T Rec. X.680. (Cited on page 43.)

[42] International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. *Abstract Syntax Notation One (ASN.1) Specification of Packed Encoding Rules (PER)*, 2002. ITU-T Rec. X.691. (Cited on pages 43 and 46.)

[43] International Telecommunication Union, Telecommunication Standardization Sector, Geneva, Switzerland. *Mapping W3C XML Schema Definitions into ASN.1*, 2004. ITU-T Rec. X.694. (Cited on page 46.)

[44] Van Jacobson. *RFC 1144: Compressing TCP/IP Headers for Low-Speed Serial Links*. Internet Engineering Task Force, February 1990. http://www.ietf.org/rfc/rfc1144.txt. (Cited on page 41.)

[45] Rick Jelliffe. *The Schematron Assertion Language 1.5*. Academia Sinica Computing Centre, October 2002. http://xml.ascc.net/resource/schematron/Schematron2000.html. (Cited on page 9.)

[46] Matjaz B. Juric, Bostjan Kezmah, Marjan Hericko, Ivan Rozman, and Ivan Vezocnik. Java RMI, RMI tunneling and Web services comparison and performance analysis. *ACM SIGPLAN Notices*, 39(5):58–65, May 2004. (Cited on page 20.)

[47] Jaakko Kangasharju and Tancred Lindholm. A sequence-based type-aware interface for XML processing. In Mohamed H. Hamza, editor, *Ninth IASTED International Conference on Internet and Multimedia Systems and Applications*, pages 83–88. ACTA Press, February 2005. http://www.cs.helsinki.fi/u/jkangash/xml-interface.pdf. (Cited on pages 30, 67, 81, and 82.)

[48] Jaakko Kangasharju, Tancred Lindholm, and Sasu Tarkoma. Requirements and design for XML messaging in the mobile environment. In Nikos Anerousis and George Kormentzas, editors, *Second International Workshop on Next Generation Networking Middleware*, pages 29–36, May 2005. http://www.cs.helsinki.fi/u/jkangash/xml-messaging-mobile.pdf. (Cited on pages 23 and 67.)

[49] Jaakko Kangasharju and Kimmo Raatikainen. Byte-efficient representation of XML messages. In W3C [117]. http://www.w3.org/2003/08/binary-interchange-workshop/08-xebu.pdf. (Cited on page 46.)

[50] Jaakko Kangasharju, Sasu Tarkoma, and Tancred Lindholm. Xebu: A binary format with schema-based optimizations for XML data. In Anne H. H. Ngu, Masaru Kitsuregawa, Erich Neuhold, Jen-Yao Chung, and Quan Z. Sheng, editors, *6th International Conference on Web Information Systems Engineering*, volume 3806 of *Lecture Notes in Computer Science*, pages 528–535, New York, USA, November 2005. Springer-Verlag. Short paper, http://dx.doi.org/10.1007/11581062_44. (Cited on pages 44 and 67.)

[51] Jaakko Kangasharju, Sasu Tarkoma, and Kimmo Raatikainen. Comparing SOAP performance for various encodings, protocols, and connections. In Marco Conti, Silvia Giordano, Enrico Gregori, and Stephan Olariu, editors, *Personal Wireless Communications*, volume 2775 of *Lecture Notes in Computer Science*, pages 397–406, Venice, Italy, September 2003. Springer-Verlag. http://www.cs.helsinki.fi/u/jkangash/soap-performance.pdf. (Cited on pages 20 and 57.)

[52] John C. Klensin. *RFC 2821: Simple Mail Transfer Protocol*. Internet Engineering Task Force, April 2001. http://www.ietf.org/rfc/rfc2821.txt. (Cited on page 2.)

[53] Miika Komu. Application programming interfaces for the Host Identity Protocol. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, September 2004. http://hipl.hiit.fi/hipl/hip-native-api-final.pdf. (Cited on page 60.)

[54] Mikko Laukkanen and Heikki Helin. Web services in wireless networks — what happened to the performance? In Liang-Jie Zhang, editor, *Proceedings of the International Conference on Web Services*, pages 278–284, June 2003. (Cited on page 20.)

[55] Edward Levinson. *RFC 2387: The MIME Multipart/Related Content-type*. Internet Engineering Task Force, August 1998. http://www.ietf.org/rfc/rfc2387.txt. (Cited on page 14.)

[56] Liberty Alliance Project. *Liberty Reverse HTTP Binding for SOAP Specification, Version 1.0*, 2003. (Cited on page 61.)

[57] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, May 2000. (Cited on pages 39 and 40.)

[58] Tancred Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *Third ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 93–97, September 2003.

http://www.hiit.fi/fuego/fc/papers/mobide03-pc.pdf. (Cited on page 23.)

[59] Nimrod Megiddo and Dharmendra S. Mocha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, March 2003. (Cited on page 45.)

[60] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, USA, 1997. (Cited on page 49.)

[61] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages 2001*, August 2001. http://www.extrememarkup.com/extreme/2001/index.htm. (Cited on page 9.)

[62] Ulrich Niedermeier, Jörg Heuer, Andreas Hutter, Walter Stechele, and Andre Kaup. An MPEG-7 tool for compression and streaming of XML data. In *IEEE International Conference on Multimedia and Expo*, pages 521–524, August 2002. (Cited on pages 46 and 47.)

[63] Nokia, Espoo, Finland. *Efficient MIDP Programming Version 1.1*, March 2004. (Cited on pages 18 and 82.)

[64] Object Management Group, Needham, Massachusetts, USA. *Common Object Request Broker Architecture (CORBA/IIOP), version 3.0.3*, March 2004. (Cited on pages 1 and 19.)

[65] Tero Ojanperä and Ramjee Prasad. An overview of third-generation wireless personal communications: A European perspective. *IEEE Personal Communications*, 5(6):59–65, December 1998. (Cited on page 17.)

[66] Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA. *RELAX NG Specification*, December 2001. http://www.relaxng.org/spec-20011203.html. (Cited on pages 9 and 52.)

[67] Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA. *Message Service Specification, Version 2.0*, April 2002. http://www.oasis-open.org/committees/ebxml-msg/documents/ebMS_v2_0.pdf. (Cited on page 15.)

[68] Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA. *RELAX NG Compact Syntax*, November 2002. http://www.relaxng.org/compact-20021121.html. (Cited on page 48.)

[69] Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA. *UDDI Version 3.0*, July 2002. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm. (Cited on page 16.)

[70] Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA. *Web Services Reliable Messaging: WS-Reliability 1.1*, August 2004. http://docs.oasis-open.org/wsrm/2004/06/WS-Reliability-CD1.086.pdf. (Cited on page 15.)

[71] Organization for the Advancement of Structured Information Standards, Billerica, Massachusetts, USA. *Web Services Security: SOAP Message Security 1.0*, March 2004. http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0. (Cited on page 15.)

[72] Jean Ostrem. Palm OS user interface guidelines. Document 3101-001-HW, PalmSource Inc., Sunnyvale, California, USA, February 2003. (Cited on page 17.)

[73] Eamon O'Tuathail and Marshall T. Rose. *RFC 3288: Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP)*. Internet Engineering Task Force, June 2002. http://www.ietf.org/rfc/rfc3288.txt. (Cited on page 58.)

[74] Santiago Pericas-Geertsen. Binary interchange of XML Infosets. In *XML Conference and Exposition*, Philadelphia, USA, December 2003. (Cited on page 42.)

[75] Marshall T. Rose. *RFC 3080: The Blocks Extensible Exchange Protocol Core*. Internet Engineering Task Force, March 2001. http://www.ietf.org/rfc/rfc3080.txt. (Cited on page 58.)

[76] Marshall T. Rose. *RFC 3081: Mapping the BEEP Core onto TCP*. Internet Engineering Task Force, March 2001. http://www.ietf.org/rfc/rfc3081.txt. (Cited on page 58.)

[77] Vladimir Roubtsov. Java tip 130: Do you know your data size? On the JavaWorld Web site. http://www.javaworld.com/javaworld/javatips/jw-javatip130.html. (Cited on page 69.)

[78] Paul Sandoz, Santiago Pericas-Geertsen, Kohuske Kawaguchi, Marc Hadley, and Eduardo Pelegri-Llopart. Fast Web services. On Sun Developer Network, August 2003. http://developer.java.sun.com/developer/technicalArticles/WebServices/fastWS/index.html. (Cited on pages 46 and 47.)

[79] Paul Sandoz, Alessandro Triglia, and Santiago Pericas-Geertsen. Fast Infoset. On Sun Developer Network, June 2004. http://java.sun.com/developer/technicalArticles/xml/fastinfoset/. (Cited on page 43.)

[80] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001. (Cited on page 1.)

[81] John Schneider. Theory, benefits and requirements for efficient encoding of XML documents. In W3C [117]. http://www.agiledelta.com/EfficientXMLEncoding.htm. (Cited on page 47.)

[82] Ekrem Serin. Design and test of the cross-format schema protocol (XFSP) for networked virtual environments. Master's thesis, Naval Postgraduate School, Monterey, California, USA, March 2003. (Cited on page 46.)

[83] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27, 1948. (Cited on page 47.)

[84] Aleksander Slominski. On using XML pull parsing Java APIs. On XmlPull Web site, March 2004. http://www.xmlpull.org/history/index.html. (Cited on page 30.)

[85] Dennis M. Sosnoski. XBIS XML Infoset encoding. In W3C [117]. http://www.w3.org/2003/08/binary-interchange-workshop/09-Sosnoski-position-paper.pdf. (Cited on page 43.)

[86] C. M. Sperberg-McQueen. XML and semi-structured data. *ACM Queue*, 3(8):34–41, October 2005. (Cited on page 11.)

[87] Pyda Srisuresh and Matt Holdrege. *RFC 2663: IP Network Address Translator (NAT) Terminology and Considerations*. Internet Engineering Task Force, August 1999. http://www.ietf.org/rfc/rfc2663.txt. (Cited on page 25.)

[88] Sun Microsystems Inc., Santa Clara, California, USA. *JavaBeans*, August 1997. (Cited on page 35.)

[89] Sun Microsystems Inc., Santa Clara, California, USA. *JSR 31: Java Architecture for XML Binding (JAXB)*, January 2003. http://jcp.org/aboutJava/communityprocess/final/jsr031/index.html. (Cited on page 30.)

[90] Sun Microsystems Inc., Santa Clara, California, USA. *Java Remote Method Invocation Specification*, 2004. (Cited on page 19.)

[91] Sun Microsystems Inc. and Motorola Inc. *Mobile Information Device Profile Version 2.0*, November 2002. (Cited on page 18.)

[92] Neel Sundaresan and Reshad Moussa. Algorithms and programming models for efficient representation of XML for Internet applications. In *Tenth International World Wide Web Conference*, pages 366–375, May 2001. http://www10.org/cdrom/papers/542/index.html. (Cited on page 46.)

[93] Sasu Tarkoma. *Efficient and Mobility-aware Content-based Routing Systems*. Licentiate thesis, University of Helsinki, Department of Computer Science, Helsinki, Finland, June 2005. (Cited on pages 83 and 84.)

[94] Sasu Tarkoma, Jaakko Kangasharju, and Kimmo Raatikainen. Client mobility in Rendezvous-Notify. In *Second International Workshop on Distributed Event-based Systems*, pages 1–8, June 2003. http://www.eecg.toronto.edu/debs03/papers/tarkoma_etal_debs03.pdf. (Cited on page 23.)

[95] Unicode Consortium. *The Unicode Standard, Version 4.0*. Addison-Wesley, Boston, Massachusetts, USA, August 2003. (Cited on page 6.)

[96] Eric van der Vlist. *RELAX NG*. O'Reilly, Sebastopol, California, USA, December 2003. (Cited on page 10.)

[97] Guido van Rossum and Fred L. Drake, Jr. *The Python Language Reference Manual*. Network Theory Ltd., September 2003. (Cited on page 18.)

[98] Web Services Interoperability Organization. *Basic Profile Version 1.1*, August 2004. http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html. (Cited on page 16.)

[99] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993. (Cited on page 1.)

[100] Christian Werner, Carsten Buschmann, and Stefan Fischer. Compressing SOAP messages by using differential encoding. In *IEEE International Conference on Web Services*, pages 540–547, July 2004. (Cited on page 41.)

[101] Dave Winer. *XML-RPC Specification*, June 2003. http://www.xmlrpc.com/spec. (Cited on page 12.)

[102] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Extensible Markup Language (XML) 1.0*, February 1998. W3C Recommendation, http://www.w3.org/TR/1998/REC-xml-19980210. (Cited on page 6.)

[103] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Namespaces in XML*, January 1999. W3C Recommendation, http://www.w3.org/TR/REC-xml-names/. (Cited on page 7.)

[104] World Wide Web Consortium, Cambridge, Massachusetts, USA. *WAP Binary XML Content Format*, June 1999. W3C Note, http://www.w3.org/TR/wbxml/. (Cited on pages 23, 41, and 43.)

[105] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Simple Object Access Protocol (SOAP) 1.1*, May 2000. W3C Note, http://www.w3.org/TR/SOAP/. (Cited on pages 13 and 16.)

[106] World Wide Web Consortium, Cambridge, Massachusetts, USA. *SOAP Messages with Attachments*, December 2000. W3C Note, http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211. (Cited on page 14.)

[107] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Canonical XML Version 1.0*, March 2001. W3C Recommendation, http://www.w3.org/TR/xml-c14n/. (Cited on pages 11 and 12.)

[108] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Web Services Description Language (WSDL) 1.1*, March 2001. W3C Note, http://www.w3.org/TR/wsdl. (Cited on page 16.)

[109] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Schema Part 1: Structures*, May 2001. W3C Recommendation, http://www.w3.org/TR/xmlschema-1/. (Cited on pages 9 and 31.)

[110] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Schema Part 2: Datatypes*, May 2001. W3C Recommendation, http://www.w3.org/TR/xmlschema-2/. (Cited on page 9.)

[111] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Exclusive XML Canonicalization Version 1.0*, July 2002. W3C Recommendation, http://www.w3.org/TR/xml-exc-c14n/. (Cited on page 12.)

[112] World Wide Web Consortium, Cambridge, Massachusetts, USA. *SOAP Version 1.2 Email Binding*, June 2002. W3C Note, http://www.w3.org/TR/2002/NOTE-soap12-email-20020626. (Cited on page 14.)

[113] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Encryption Syntax and Processing*, December 2002. W3C Recommendation, http://www.w3.org/TR/xmlenc-core/. (Cited on pages 12, 16, and 83.)

[114] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Signature Syntax and Processing*, February 2002. W3C Recommendation, http://www.w3.org/TR/xmldsig-core/. (Cited on pages 12, 16, and 83.)

[115] World Wide Web Consortium, Cambridge, Massachusetts, USA. *SOAP Version 1.2 Part 1: Messaging Framework*, June 2003. W3C Recommendation, http://www.w3.org/TR/soap12-part1/. (Cited on page 13.)

[116] World Wide Web Consortium, Cambridge, Massachusetts, USA. *SOAP Version 1.2 Part 2: Adjuncts*, June 2003. W3C Recommendation, http://www.w3.org/TR/soap12-part2/. (Cited on page 13.)

[117] World Wide Web Consortium. *W3C Workshop on Binary Interchange of XML Information Item Sets*, September 2003. http://www.w3.org/2003/08/binary-interchange-workshop/Report.html. (Cited on pages 41, 89, and 93.)

[118] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Document Object Model (DOM) Level 3 Core Specification*, April 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/. (Cited on pages 10 and 29.)

[119] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Extensible Markup Language (XML) 1.0*, 3rd edition, February 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xml-20040204/. (Cited on pages 2, 5, 6, and 8.)

[120] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Extensible Markup Language (XML) 1.1*, February 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xml11-20040204/. (Cited on page 6.)

[121] World Wide Web Consortium, Cambridge, Massachusetts, USA. *SOAP 1.2 Attachment Feature*, June 2004. W3C Note, http://www.w3.org/TR/2004/NOTE-soap12-af-20040608/. (Cited on page 14.)

[122] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Web Services Addressing (WS-Addressing)*, August 2004. W3C Member Submission, http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/. (Cited on page 15.)

[123] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Information Set*, 2nd edition, February 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xml-infoset-20040204/. (Cited on page 10.)

[124] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Describing Media Content of Binary Data in XML*, May 2005. W3C Note, http://www.w3.org/TR/2005/NOTE-xml-media-types-20050504. (Cited on page 14.)

[125] World Wide Web Consortium, Cambridge, Massachusetts, USA. *SOAP Message Transmission Optimization Mechanism*, January 2005. W3C Recommendation, http://www.w3.org/TR/2004/REC-soap12-mtom-20050125/. (Cited on page 14.)

[126] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Web Services Addressing 1.0 — Core*, August 2005. W3C Candidate Recommendation, http://www.w3.org/TR/2005/CR-ws-addr-core-20050817/. (Cited on pages 15, 26, and 57.)

[127] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Web Services Addressing 1.0 — SOAP Binding*, August 2005. W3C Candidate Recommendation, http://www.w3.org/TR/2005/CR-ws-addr-soap-20050817/. (Cited on page 15.)

[128] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, August 2005. W3C Last Call Working Draft, http://www.w3.org/TR/2005/WD-wsdl20-20050803/. (Cited on page 16.)

[129] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*, August 2005. W3C Last Call Working Draft, http://www.w3.org/TR/2005/WD-wsdl20-adjuncts-20050803/. (Cited on page 16.)

[130] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Binary Characterization*, March 2005. W3C Note, http://www.w3.org/TR/xbc-characterization. (Cited on page 42.)

[131] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Binary Characterization Measurement Methodologies*, March 2005. W3C Note, http://www.w3.org/TR/xbc-measurement. (Cited on page 42.)

[132] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Binary Characterization Properties*, March 2005. W3C Note, http://www.w3.org/TR/xbc-properties. (Cited on pages 11 and 42.)

[133] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Binary Characterization Use Cases*, March 2005. W3C Note, http://www.w3.org/TR/xbc-use-cases. (Cited on pages 19 and 42.)

[134] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML-binary Optimized Packaging*, January 2005. W3C Recommendation, http://www.w3.org/TR/2004/REC-xop10-20050125/. (Cited on page 14.)

[135] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Path Language (XPath) 2.0*, November 2005. W3C Candidate Recommendation, http://www.w3.org/TR/2005/CR-xpath20-20051103/. (Cited on page 10.)

[136] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XQuery 1.0: An XML Query Language*, November 2005. W3C Candidate Recommendation, http://www.w3.org/TR/2005/CR-xquery-20051103/. (Cited on page 10.)

[137] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, November 2005. W3C Candidate Recommendation, http://www.w3.org/TR/2005/CR-xpath-datamodel-20051103/. (Cited on page 10.)

[138] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XSL Transformations (XSLT) Version 2.0*, November 2005. W3C Candidate Recommendation, http://www.w3.org/TR/2005/CR-xslt20-20051103/. (Cited on page 10.)

[139] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977. (Cited on page 39.)