NTNU
Innovation and Creativity

Efficient Web Services on Mobile Devices

Lars Johnsrud

Master of Science in Communication Technology
Submission date: August 2007
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Dinko Hadzic, FFI

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Web Services are becoming a de facto technology for implementing applications based on a Service Oriented Architecture (SOA), and achieving interoperability between different systems. Today, Web Services are usually realized on computer systems where processing resources and network band width are not a limitation.

It is also desirable to allow mobile devices to access the growing Web Services infrastructure. Although mobile devices are getting more advanced and powerful, the large information overhead of Web Services is a significant challenge and limitation in the context of mobile devices.

The thesis should identify and evaluate techniques that can improve the performance of Web Services on mobile phones, and discuss the potential advantages and disadvantages of such techniques. Binary XML, currently being standardized by W3C, and general data compression seem to be promising methods to reduce the processing, memory and network band width requirements, thus allowing more efficient realizations of Web Services on mobile devices.

A prototype application should also be developed, and it may be tested both in an emulated environment, and on real mobile device. The prototype case is a company's internal phonebook, made available to employees through a Web Service mobile application.

Assignment given: 25. January 2007
Supervisor: Peter Herrmann, ITEM

Abstract

Efficient solutions for Web Services on mobile devices would allow truly global, platform independent and interoperable information access, anywhere and at any time. While Web Services are continuously gaining ground, they are commonly reserved for use on personal computers and high-capacity servers, even though mobile devices are continuously becoming more advanced in terms of processing resources and wireless communication capabilities.

This thesis identifies several challenges of accessing Web Services from mobile devices, and evaluates and discusses methods for creating more efficient solutions. Some of the challenges are the limited bandwidth and high communication latency. Reducing the size of XML information transferred and optimizing the communication protocol stack are identified as possible solutions to overcome these challenges. Additionally, as the communication cost correlates with the amount of data transferred, more efficient Web Services solutions are clearly beneficial for the end-user.

The approaches described to reduce the size of XML files are traditional compression, alternative representation of the files, and binary XML. Binary XML is a compact representation of information that keeps the desirable structure of XML intact. The Efficient XML Interchange format currently being standardized by W3C is studied in more detail. Furthermore, optimizing the protocol stack has also been evaluated.

A prototype Web Service system has been developed and tested in both simulated environments and in real surroundings using GPRS, EDGE and UMTS network connections. The results from the measurements show that both compression and the use of binary XML reduce the size of the information significantly and thereby the cost. Time needed to transfer the information is also reduced, and this effect is most apparent when the original files are large. Binary XML may however be the desirable format since it enables direct interaction with the information and keeps the memory footprint small. To reduce the time needed to transfer the information further, removing the HTTP protocol and optimizing the transport protocol, seems to minimize the effect of the latency.

Preface

This master thesis is written at the Department of Telematics at the Norwegian University of Science and Technology (NTNU) in the spring semester of 2007. It was carried out at the Norwegian Defence Research Establishment (FFI).

I would like to use this opportunity to thank my academic supervisor at the Department of Telematics, Professor Peter Herrmann. Special thanks go to my supervisor at FFI, Dinko Hadzic, for his guidance and suggestions during the work with this thesis

Furthermore, I would like to thank Anne Kibsgaard for her proofreading effort.

Hamar, August 2007

Lars Johnsrud

Table of Content

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
ADSL	Asymmetric Digital Subscriber Line
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CLDC	Connected Limited Device Configuration
CPU	Central Processing Unit
EDGE	Enhanced Data for GSM Evolution
EFX	Efficient XML
EXI	Efficient XML Interchange
FFI	Norwegian Defence Research Establishment
GB	Gigabyte
GPRS	General Packet Radio Service
HSDPA	High-Speed Downlink Packet Access
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ITU	International Telecommunication Union
ITU-T	International Telecommunication Union - Standardization
Java ME	Java Micro Edition
JDK	Java Development Kit
JSR	Java Specification Request
KB	Kilobyte
LAN	Local Area Network
MB	Megabyte
MIDP	Mobile Information Device Profile
MIME	Multipurpose Internet Mail Extensions
NIST	National Institute of Standards and Technology
NOK	Norwegian Krone
NTNU	Norwegian University of Science and Technology
OMTP	Open Mobile Terminal Platform
OS	Operating System
PC	Personal Computer
RFC	Request for Comment
RTT	Round Trip Time
SDK	Software Development Kit
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
W3C	World Wide Web Consortium
WIFI	Wireless Fidelity
WSDL	Web Services Description Language
WTP	Web Tools Platform

XML	eXtensible Markup Language
XSD	XML Schema Definition

1 Introduction

1.1 Background and Motivation

Today, people expect to have access to information anywhere and at anytime. Web Services have proven themselves to be a good solution for achieving this when using computers connected to high bandwidth networks. By developing applications based on Serviced Oriented Architecture (SOA) as Web Services, one may achieve the desirable goals of interoperability, platform independency and reuse of services.

Mobile devices have the potential of enabling global, wireless access to services from small, handheld, battery powered devices. Even though mobile devices are becoming more powerful almost every day, they still are subject to some major challenges. Bandwidth and cost are two areas that are subject to limitations when adopting Web Services for mobile devices. Another thing to be considered is processing resources, in terms of Central Processing Unit (CPU), memory resources and battery consumption.

A challenge with bandwidth is the lack of it in many situations. Even though mobile devices may be connected to Universal Mobile Telecommunications System (UMTS) or even Wireless Fidelity (WIFI) networks in populated areas, they may encounter network connection of Enhanced Data for GSM Evolution (EDGE) or General Packet Radio Service (GPRS) in less populated areas. This may again lead to a significant loss in performance of many services. Even though the services are still functioning as intended, the time needed to transfer the data may lead to a bad user experience, and hence the services may be considered to be useless. One solution to this problem might be to reduce the amount of data being transferred, and by this reducing the time used to transfer the data needed by the service. By sending and receiving less data, lower battery consumption may also be achieved.

Even if the Web Services are working properly, there are other reasons for reducing the amount of data being transferred, namely the cost. Since the cot of transferring data is determined by the amount of data being transferred, the network operators would prefer if their costumers transfer large data amounts, while the costumers would prefer services that transfer as little data as possible. This picture gets even clearer when the abroad costs are taken into consideration. Currently (spring 2007), the Norwegian telecommunication company Telenor is charging 43,75 Norwegian Krone(r) (NOK) for each Megabyte (MB) transferred in Western European countries [1]. NOK is the currency in Norway, and one Euro is approximately eight NOK. These rates are even higher in many countries outside Europe.

With this in mind, the motivation for developing solutions that lead to less bandwidth consumption should be clear. If the amount of data that has to be transferred can be reduced, it would make Web Services a more suitable technology for bringing services to the mobile user.

1.2 Thesis Scope and Limitations

The focus of this thesis is on eXtensible Markup Language (XML) Web Services and methods for improving their usability on mobile devices. Mobile devices usually connect to Web Services over wireless links. These wireless links lead to higher response times due to limited bandwidth and higher latency compared to wired links. In addition, connection over these links is in many cases charged based on the amount of data transferred. This thesis identifies and studies methods that reduce the amount of data being transferred via wireless links between a Web Service and a Web Service client on a mobile device. Doing this may lead to an improved response time due to less impact of the limited bandwidth. Secondly, the cost will be reduced on links that are charged on a per byte basis. Stack optimization is also evaluated as it also may yield better response time, and also possible cost savings.

A prototype phonebook service will be developed to get a real-life experience of the effect of reducing the amount of data being transferred. Based on this prototype several measurements will be performed both in a simulated environment and on wireless links used by real-life mobile devices. The obtained results will be discussed and evaluated both in terms of response time and reduced cost. Stack optimization is not explored in the prototype and measurements.

Although this thesis focuses less on processing resources on a mobile device, and thus the related considerations, these are issues that have to be taken into account when developing services targeted at these devices. The writer is well aware of the importance of these areas, but the time constraint gives no room to look thoroughly into these areas.

1.3 Related Work

Accessing Web Services from mobile devices are continuously getting more attention. At the same time the problems related to the wireless links used to achieve this, are becoming more evident. In [2] the performance of Simple Object Access Protocol (SOAP) over wireless links are studied, one of the main facts it identifies, is that protocols underlying SOAP can be improved significantly in conjunction with wireless links. The same observation is also done in [3] where the effect of sending SOAP messages over User Datagram Protocol (UDP) is exploited. It reports that this scheme gives an increase in throughput compared to the more commonly used SOAP over Hypertext Transfer Protocol (HTTP).

Another extensive work was performed by World Wide Web Consortium (W3C) in their job with standardizing a Binary XML format. In [4] they present measurements related to processing efficiency and other properties for a set of formats they evaluated as a basis for a Binary XML format. In this thesis the possible benefits of such a format is studied in terms of reduced response time of Web Services on mobile devices.

1.4 Outline

CHAPTER TWO describes several challenges related to Web Services on mobile devices. It also presents the technology that enables access to Web Services from mobile devices.

CHAPTER THREE focuses on two different ways to optimize Web Services for mobile devices; reducing the size of XML information and protocol stack optimizations.

CHAPTER FOUR presents the design of the prototype Phonebook service. In this chapter the framework for the measurements of performance of compact XML representations is also presented.

CHAPTER FIVE gives a brief description of the most important parts of the implementation.

CHAPTER SIX presents the measurements and results related to reducing the size of XML for use with Web Services on mobile devices. It also discusses these results.

CHAPTER EIGHT gives a conclusion of the study, and outlines possible future work.

2 Web Services on Mobile Devices

2.1 Web Services and the Big Picture

SOA is an architectural concept that recommends and advises the use of loosely coupled services. SOA is more than just services, SOA is about business functionality and how the information technology could be integrated into this [5]. The Web Services technology is commonly used to realize SOA.

Web Services work well today in the way that they form a base for services that are interoperable, platform independent and have clearly defined service interfaces. In [6] it is stated like this:

> "Based on XML standards, Web Services can be developed as loosely coupled application components using any programming language, any protocol or any platform."

By using Web Services instead of legacy systems, a business can be more adaptable. New services can be created by combining old ones, and inefficient services can be rewritten as long as the service interface is kept unchanged. In big monolithic legacy systems it is often easier to upgrade hardware than to make the software perform better. However, in today's world this is not the way to do it. One has to make the business service logic, easy to use and easy to access from a vide variation of devices. In addition to this, the adaptation has to be quick to perform.

Since the service interfaces are clearly defined and the communication is performed over standardized protocols, the way that a service should be accessed is thus defined. The result of this is that the code needed to access a service from other platforms or a mobile device can be automatically generated based on the interface and protocol. This again leads to a quick and easy way to combine services into new services and to access services from all kinds of devices.

There are, however, reasons to believe that the designers of Web Services did not predict that mobile devices that were initially meant to be a peripheral voice call devices, has now become a small computer that is expected to do almost anything anywhere [7]. The fact that Web Services and mobile devices that initially were meant to exist apart from each other, now have to work together has led to great challenges for application developers. Now we will have a closer look at these challenges.

2.2 Mobile Devices

Even though the mobile devices have become very advanced, they are still very limited compared to ordinary computers. To get a clear picture of this it can be useful to compare the

mobile device with a spaceship. They can both go to almost any place, but their performance at these locations is restrained by their initial capabilities. Moreover, there is no way for the spaceship to add more fuel as it is impossible for the mobile device to get more battery power at a location, and one can not add another person at a remote planet to perform more tasks, in the same way as one can not add more processing power to the mobile device.

So let us now take a look at some of the challenges related to mobile devices. Compared to personal computers, the two most obvious limitations are the size of the screen and the keyboard. Both of these two limitation post challenges on the user interface of an application. The mobile devices also have less available processing power, so computations will take longer time than on a Personal Computer (PC). These devices also depend on wireless data connections, which lead to both reduced bandwidth for data transfer and higher latency, in addition to the transfer of data via such links is charged based on the amount of data transferred. Finally, the mobile devices are battery powered, which naturally constraints the operational time of the devices.

Table 2.1 shows the characteristics of a mobile device compared to a typical PC. One of the things that should be noticed is that the CPU in the Nokia N95 is 32 bit while the one in the computer is 64 bit. In addition, it should be noted that Nokia N95 is one of the most advanced mobile phones on the market today. In other words, the PC will have at least 10 times faster CPU, 20 times more memory and 100 times more storage than the currently most advanced mobile phone. Even though the mobile devices are becoming very advanced they are still quite limited compared to a PC.

Table 2.1 Comparison of Mobile device and PC Characteristics.

Model	**NOKIA N95 [8]**	**PC typical**
CPU	330 MHz	2 GHz
Memory	64 MB, ~ 22 MB free	1024 MB
Storage	160 MB + 2 GB optional	250 GB

The huge amount of different devices and platforms leads to yet other challenges when developing applications targeted at mobile devices. The mobile devices have different capabilities, for instance some may have a camera and some may not. Above this again they run different Operating Systems (OS). Table 2.2 below shows some of the OS that are run on mobile devices today. The result of this diversity is that the application developer has to deal with many different platforms to adopt the application to.

Table 2.2 Listing of mobile device OS's [9].

Mobile Device OS name
Symbian OS
Linux
Windows Mobile
RIM BlackBerry
Palm OS

One way to deal with this problem is to develop applications based on the Java Micro Edition (Java ME). The Connected Limited Device Configuration (CLDC) specification [10] urges to "Focus on application programming rather than systems programming". The meaning of this is that the Java ME platform hides the complexity of the underlying system, and provides the developers with a programming environment that is independent of the platform and the device. This is illustrated in Figure 2.1 below.

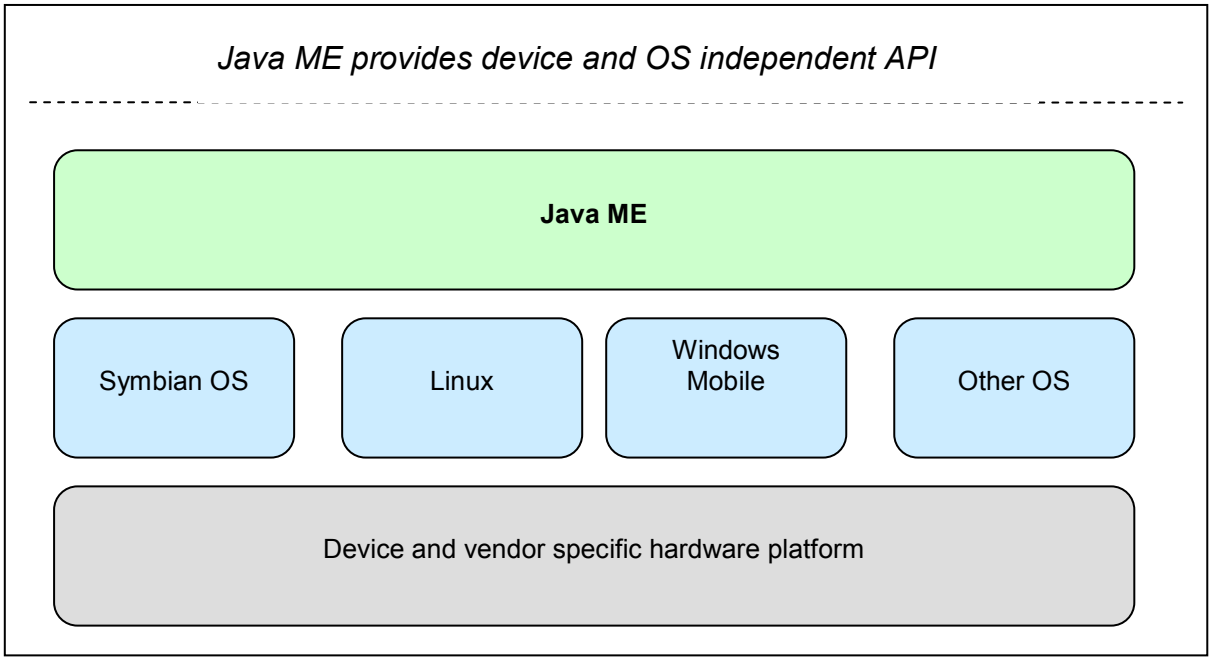

Figure 2.1 Illustration of how Java ME hides the underlying OS and device.

The facts mentioned above make the Java ME platform an excellent environment for developing applications for mobile devices. In addition, the platform is massively deployed already; in a paper [11] from January 2006 Open Mobile Terminal Platform (OMTP) points out that it was estimated that there were over 800 million mobile devices supporting the Java ME platform. It also states that applications developed for this platform has generated revenues of over $1 billion annually. By now, it can be expected that both of these numbers have increased. With the introduction of the optional Web Service package it should be expected that new markets will adopt this platform, and thereby increase the revenues substantially. Last but not least, the Java ME platform is currently becoming open source. The process of releasing the open source code is expected to be completed by the end of 2007 [12].

Based on the advantages mentioned in this chapter the Java Me platform was chosen to be used in the work described in this report.

2.2.1 The Java ME Platform

Now let us have a look at the architecture that makes up the Java ME platform that will be used. The Java ME technology stack consists of the configuration, profile and optional packages in combination with a java virtual machine [13]. For mobile devices the configuration is called CLDC. It consists of the virtual machine and some low-level libraries that form a standardised layer against the mobile devices that support this configuration. CLDC is intended for devices with as little as 192 KB of total memory and upwards. Other characteristics of typical devices are low processing capabilities, they are battery powered and have low bandwidth wireless network connection [10]. The Mobile Information Device Profile is a layer above the CLDC and provides a set of Application Programming Interfaces (APIs) for a set of devices with the same capabilities. In this way, the application developers are provided with a common way to develop applications for a set of devices, and optional packages can further extend the functionality. Figure 2.2 below shows the relationship between the different parts of the stack.

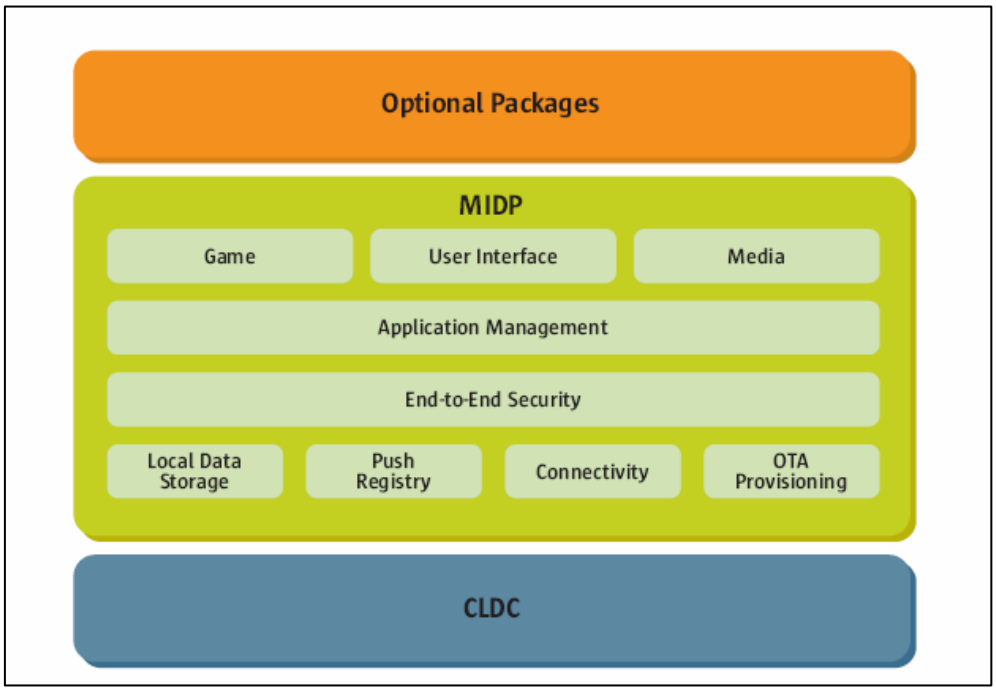

Figure 2.2 Java ME technology stack [14].

The applications developed will be placed on top, and use libraries both from the Mobile Information Device Profile (MIDP) profile and the optional packages. Applications targeted at this platform are called MIDlets. These applications have some special constraints related to their lifecycle, but they are not important here, so the interested reader is advised to go to other literature for information.

The Java Specification Request (JSR) 172 specifies the Java ME Web Services API [15]. This is an optional package that enables access to Web Services from mobile devices. JSR 172 has been derived from the original API for the Java Standard Edition, and provides a subset of

that API [15]. This reduction enables it to run on Java Me devices on the cost of some features.

In order to develop applications for this platform, a wireless toolkit was needed. This is a set of tools that enables easier development towards mobile devices. One important tool is the emulator, which lets the developer emulate the target device on the computer and test the application without uploading it to the device. Since the target device for this work was a Sony Ericsson device, the toolkit was downloaded from their site. The toolkit used to implement the prototype phonebook service in this thesis was the *Sony Ericsson* Software Development Kit *(SDK) 2.2.4 for the Java(TM) ME Platform* [14].

Above the most essential technologies to access Web Services from mobile devices have been presented. In order to develop and run the fully functional phonebook prototype Web Service, more software and hardware are needed. Since these are of less interest and are highly related to the implementation of the phonebook Web Service, they are presented in chapter 5, Implementation.

2.3 Focus of this Thesis

This thesis focuses on the challenges related to the wireless links used by mobile devices to access Web Services. In this chapter some facts about these links and comparison with wired links will be presented to clarify these challenges.

Problems that are introduced by wireless links are illustrated in Figure 2.3. The wireless link that is illustrated is a UMTS link. The characteristics for UMTS are obtained from [16], and for Asymmetric Digital Subscriber Line (ADSL) the number are based on the writer's access to the Internet. Latency is obtained by using the ping command on a Windows system. It should also be clear that this parameter is related to the distance the data will be transferred.

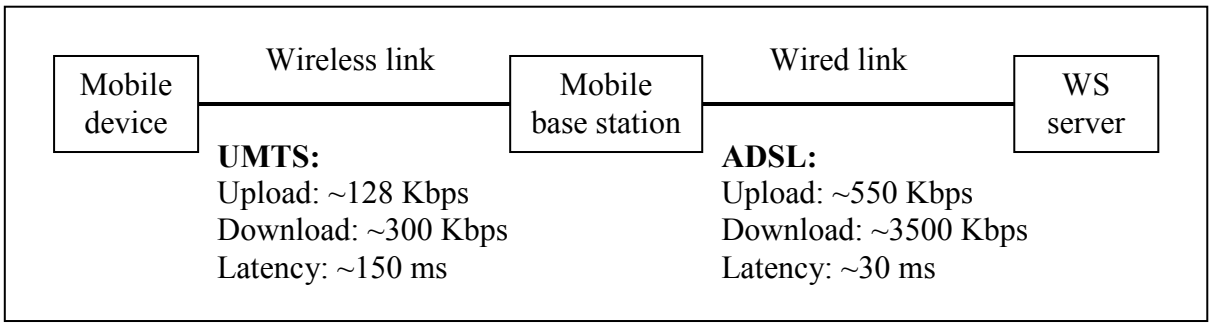

Figure 2.3 The link characteristics related to a mobile device that access a Web Service.

Today, the UMTS is the best link that is available in Norway if one does not have access to a WIFI network. However, High-Speed Downlink Packet Access (HSDPA) is currently being deployed and this network performs quite a bit better than UMTS. HSDPA is the next technology used in mobile networks for data transmission, and provides download bandwidth

up to 3,6 Mbps [16]. There are also situations where one will experience networks that perform worse than UMTS, and this situation will persist for a long time to come.

This thesis will look at the effect of the limited bandwidth and the high latency introduced by wireless links. Other characteristics like bit errors etc., will only be studied when they have an impact in connection with the limitations mentioned. In the next chapter, techniques that may reduce the effect of these limitations will be proposed. In later chapters, one of these techniques will be applied in an implementation of a prototype service. Measurements based on this prototype will then be carried out to illustrate the possible benefits of this technique. The limited processing capabilities and small screens are also constraints that are very interesting and pose challenges that should be investigated thoroughly. However the timeframe of this thesis do not permit this.

3 Methods for Optimizing Web Services

3.1 Possible Approaches

The previous chapter identifies limited bandwidth and high latency of wireless links as limitations when accessing Web Services from a mobile device. These factors lead to high response time in addition to data transfer being charged by the amount of data transferred. In this chapter we will look at two approaches to try to make Web Services more suitable for wireless links. These two approaches overlap partially, but their main focus is quite different from each other, as will be clear throughout this chapter. It should be mentioned that none of the two approaches are believed to be the only solution, but rather a combination of the two.

3.1.1 Compression Vs. Binary XML

The first approach that was taken was to reduce the size of XML files. It is a fact that XML documents contain a huge overhead. By overhead it is meant the information that does not carry actual information, but rather describes the actual information. Overhead is sometimes called meta information, and the tags in XML described below are examples of overhead. If the size of the XML documents transferred could be reduced, it should lead to better response time and cheaper transfer.

To better understand the problem with overhead, the XML address format developed and described in chapter 4.5 will be used. This format is utilized in the Phonebook prototype service presented in chapter four through five. As can be seen in Figure 3.1 the tags used in XML lead to a huge overhead. One of the design goals stated in Extensible Markup Language 1.0 [17] published by W3C is as follows; “XML documents should be human-legible and reasonably clear”. The consequence of this statement is documents with more information than needed. Another requirement for XML is that every tag also shall have a corresponding end tag. An example from the figure is the pair: <contacts> and </contacts>. This leads to a large overhead and a high amount of redundancy. Every *contacts* element below has about twice as much metadata as data. In other words, it is fair to say that in our example; only about one third of the XML file size is carrying “real” information.

```
<contacts>
      <firstName>Elin</firstName>
      <lastName>Borgen</lastName>
      <phoneNumber>95903263</phoneNumber>
      <eMail>borgen88@stud.ntnu.no</eMail>
      <Address>Hordnesvegen 293</Address>
      <PostalCode>5244</PostalCode>
      <Town>Fana</Town>
</contacts>
.
.
<contacts>
      <firstName>Elin</firstName>
      <lastName>Gudmundsen</lastName>
      <phoneNumber>94864723</phoneNumber>
      <eMail>gudmundsen14@gmail.com</eMail>
      <Address>Munkedamsveien 98</Address>
      <PostalCode>0270</PostalCode>
      <Town>Oslo</Town>
</contacts>
```

Figure 3.1 XML file which represent address information.

The most obvious way to reduce the XML file size would be to reduce the size of the tags. For example the *contact* tag could be reduced to *<c>* and *</c>*, the *phoneNumber* tag could be *<pN>* and *</pN>* etc. This solution alone would most likely lead to a significant reduction of the file sizes. However, the suggested solution would also dramatically reduce the "readability" which is an important objective for XML. The readability of XML is in fact one of the reasons why XML has become such a popular format. We will thus look at other ways to reduce the size of XML files, namely compression and binary XML.

Compression means preserving all the information, but reducing the size of it. The original file size can be obtained by reversing the compression algorithm. This method is described in chapter 3.2 below.

The second method that will be looked into is alternative ways to represent the information in order to obtain a more compact format that is faster to process. In contrast to compression, the original representation can usually not be obtained. Another fact about these formats is that they do not add the processing overhead introduced by compression and decompression. If these representations are well formed and have a good structure, they are often referred to as Binary XML. It can be said that Binary XML is a format that relates to XML but does not meet all the requirements of XML.

3.1.2 Stack Optimization

Next, a solution that both deals with latency and bandwidth will be looked at, namely stack optimization. By taking away or changing one or more layers in the protocol stack of Web Services, the number of messages that have to be exchanged between the server and the client may be reduced. If the number of messages is reduced, the effect of the latency of the wireless

links will also be reduced. If it is possible to remove one or more layer of the stack, the amount of data that have to be transferred is also reduced since every layer adds overhead. This would be advantageous in relation to the limited bandwidth. This approach is presented in chapter 3.5.

3.2 Compression

The basic idea behind compression is that information by applying an encoding algorithm can be represented in a smaller way. By applying the reversed algorithm for decoding, the original information can be obtained again. The fact that makes this possible is that information usually contains a lot of redundant information. One example of this is the abbreviation OS for Operating System used in this thesis. This can be said to be a compression since the amount of data used to represent it is reduced. Figure 3.2 below illustrates this example. As can be seen the amount of data being transferred between the encoding and decoding is much smaller than the original information.

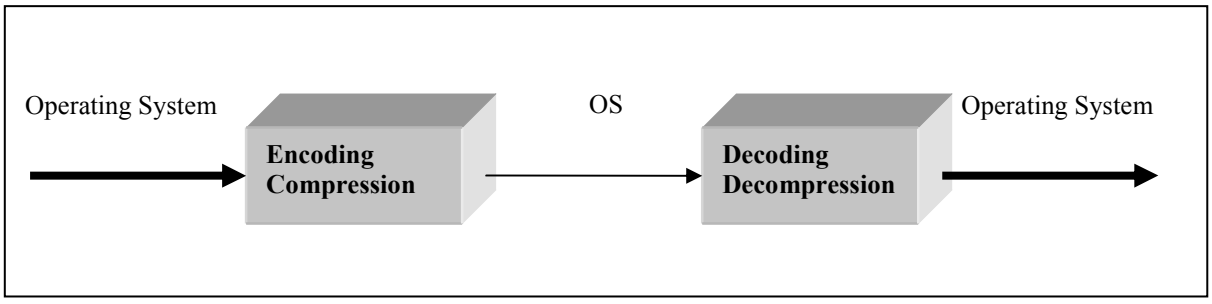

Figure 3.2 Illustration of how compression works.

3.2.1 Deflate Compression

The Deflate Compression method was introduced in the PKZIP 2 archiving tool released in 1993 [18]. After this it has become the de facto standard for lossless data compression and the method is used by many compression formats, amongst others the widely used ZIP format. It has later been specified in the Request for Comment (RFC) 1951 [19]. The compressed data will consist of a set of arbitrary seized blocks. Each block is compressed using a combination of the LZ77 algorithm and Huffman coding. A brief overview of the RFC is given below.

The LZ77 compression algorithm was introduced by Ziv and Lempel in the article: A Universal Algorithm for Sequential Data Compression [20]. This algorithm works in the way that it looks for duplicated strings in the previous data, and if a match is found a back reference pair is inserted instead of the matching string. This pair consists of the *length* and the *backward distance*. The backward distance indicates where the previous identical string started and the length indicates how long it is. The deflate format limits the backward distance to 32K bytes and the length to 258 bytes [19], so the implementation must keep track of the

last 32K bytes of uncompressed data to perform the matching. The matching can however be performed across the compressed blocks.

Each block of compressed data is represented using Huffman coding. For each block a new Huffman tree is built, and this tree gives the coding for the block it is embedded in. Huffman coding is a coding scheme where the symbols or characters that are most frequently used in the data are coded with few bits and the ones that do not appear that often are coded with more bits [21]. This technique alone will in many cases give great compression. To decode the data, the reversed step has to be applied. The Huffman coding is removed and the references are replaced with the actual data.

This is by no means a complete description of the deflate format, but it describes the method in a short form. This method seems to have great potential since it searches for duplicated information. In conjunction with the tags in XML which introduce a great amount of duplicated data, this should reduce the size quite a bit. In addition the savings of the Huffman coding alone can be expected to be between 20 % and 90 % [21]. The *deflate* method described here is used in many compression utilities, amongst others ZIP, GZIP and ZLIB [22].

3.2.2 GZIP and ZLIB

GZIP was initially developed to produce a compression utility that was independent of any patents, and its file format is specified in RFC 1952 [23]. The main idea behind the compression is the deflate method described above. ZLIB is an abstraction of GZIP, and it is specified in RFC 1950 [24]. ZLIB is a free library that can be used to perform compression in other applications, and is currently used in more than 500 applications [22]. Both GZIP and ZLIB were written by Jean-loup Gailly (compression) and Mark Adler (decompression). Because of easy integration into applications through its library, ZLIB was chosen as the compression format that would form the basis for the compression to be used in the implementation of the prototype Web Service described later in this report.

3.2.3 XMill

XMill is a compression tool specially targeted at compressing XML, and its implementation is a result of work done at the AT&T Labs Research in New Jersey, USA, in 1999 [25]. The basic idea in XMill is to group XML content with similarities, such content can for example be all the integers in a document. These similarities can then be exploited to achieve better compression. In [25] it is reported that this method may lead to a compression that is down to half the size of the same XML document compressed with GZIP. However, this improved compression comes at the cost of added processing time, so there has to be a tradeoff between compression ratio and processing time.

3.3 Binary XML

As mentioned in the introduction to chapter 3, another solution in order to compact data could be to represent the XML information in different ways. If this representation is well formed, it can be said to be Binary XML. The most basic alternative representation could be a semicolon-separated string. The first contact from Figure 3.1 would then look like this:

;Elin;Borgen;95903263;borgen88@stud.ntnu.no;Hordnesvegen 293;5244;Fana;

This would be a much more compact representation than the original XML. A format on this form would, however, lack many of the good qualities of the XML format. The first thing to notice is that a developer who wishes to use this format in his/her application would have to have additional information regarding the format. It would be essential to know what kind of information could be expected and how to process it. XML on the other hand will be fairly self-describing if the *element* names are wisely chosen, and thereby there is no need for more documentation to understand it. An additional fact is that the format illustrated above is a flat format. This makes it both hard to represent in an application and hard to operate on. By hard to operate on, it is meant that operations like search, insert, delete, alter, etc. would demand an extensive amount of processing. XML on the other side is represented through a well formed hierarchic structure. This makes it a lot easier to build a data model in an application, and the operation mentioned above is much easier to perform. All the facts mentioned here make the XML less prone to errors than the semicolon-separated format illustrated above.

XML has been widely adopted and has proven itself to be a good way to represent and exchange information. However, one size does not fit all, and XML is not an ideal format for message exchange between a Web Service and a mobile device. To overcome this challenge different proprietary formats have surfaced from a number of developers and organizations. This has lead to a number of alternative representations that are not interoperable [26].

The W3C XML Binary Characterization Working Group [27] was created to investigate the need and possibility for a more efficient representation of XML, or in other words, Binary XML. In XML Binary Characterization [28] Binary XML is defined as follows:

> "a format which does not conform to the XML specification yet maintains a well-defined, useful relationship with XML. By "useful" we mean that practical systems may take advantage of this relationship with little additional effort. For example, it may be useful to convert a file from XML to Binary XML."

The XML Binary Characterization Working Group ended successfully with a series of publications in mach 2005. In their document XML Binary Characterization [28] it is concluded that there is a need for Binary XML, and that it is feasible to develop such a format. This working group was followed up by the W3C Efficient XML Interchange (EXI) Working Group [29], which should specify such a format based on the conclusion from the XML Binary Characterization Working Group. In July 2007 they published the first public working draft of the EXI Format specification [30]. The Efficient XML format developed by AgileDelta [31] forms the basis for this format. This format is examined in chapter 3.3.1, whereas chapter 3.3.4 presents the International Telecommunication Union - Standardization (ITU-T) format Fast Infoset [32] which is another example of Binary XML.

These formats may satisfy the needs in conjunction with mobile devices for a format that is more compact, smaller in size and at the same time has many of the properties that has made XML to the huge success it has become

3.3.1 Efficient XML Interchange Format

W3C is currently standardizing a format to meet some of the shortcomings of XML. This format has been given the name EXI Format [33]. The first W3C Working Draft for EXI was published in July 2007, and is the only current document. Hence this specification is not complete, but the overall idea is ready.

The main idea with EXI is to encode the part that is most likely to appear in an XML document with the fewest bit and hence obtain a more compact representation. The event codes in the EXI stream are encoded with codes that are similar to Huffman code, but simpler [33]. The algorithm that performs the encoding is kept as simple as possible and the number of data types is kept low. These facts should lead to a format that is compact and possible to process on devices with limited capabilities. Additionally EXI is kept highly interoperable with the current XML technologies to facilitate an easy adoption.

3.3.2 Efficient XML

Efficient XML (EFX) was one of the formats the W3C XML Binary Characterization Working Group [28] investigated during their work with requirements for a binary XML format. It was later adopted by the W3C Efficient XML Interchange Working Group as the basis for the specification of the EXI format. EFX was developed by AgileDelta [31] and provides a very compact representation of XML information. AgileDelta states that it produces considerably smaller files, as much as 370 times smaller than standard XML and 27 times smaller than WinZip in the best cases [34]. It is also stated that it is faster compared to compression since applications can read and write it directly and do not need to compress and uncompress.

3.3.3 Abstract Syntax Notation One

Abstract Syntax Notation One (ASN.1) is the most used notation for defining protocols [35]. Today ASN.1 is covered by the International Telecommunication Union (ITU) X.680 series of specifications. In short the ASN.1 defines a set of universal types, meaning that an ASN.1 type compiled on one machine will be the same when it is decompiled on another. These types can then be combined to get more complex structure [35]. In ITU-T Rec. X.693 [36] ASN.1 is described as “a notation for definition of messages to be exchanged between peer applications”. It should now be clear that ASN.1 and XML have many of the same properties, and by this they are related. In fact the X.693 recommendation gives the rules for encoding ASN.1 types using XML. In addition the X.694 [37] recommendation specifies how W3C XML Schema Definition (XSD) can be mapped into an ASN.1 schema.

To sum up, ASN.1 can be used to make a compact binary representation of XML, and this would be highly preferable in conjunction with limited bandwidth. It would also be faster to process than traditional XML. However, it leads to a format that lacks the self-describing feature of XML [38]. This feature is one of the reasons XML has become such a popular format, and would be a significant loss.

3.3.4 Fast Infoset

Fast Infoset is specified in ITU-T rec. X.891 [32], and is an ASN.1 representation that integrates with XML. It is a binary representation based on ASN.1 that retains the self-describing property of XML. The result of this is a format that is more compact and faster to parse than traditional XML [39].

The Fast Infoset format can be used as the basis for information exchange in the Web Service framework. Currently it is possible to choose it as the format to be used in the Java Web Service Developer Pack [39] and hence to make Web Services more efficient. It should be noted that Fast Infoset was one of the formats that W3C considered as the basis for their standardization of a binary XML format. It was, however, not considered to meet W3Cs requirements for a Binary XML format as well as EFX.

3.4 Final Words about Compression and Binary XML

Even though this report focuses on the limited bandwidth and high latency of wireless links, other considerations have to be taken into account. From the link's point of view, the internal structure of the information is of no interest. The less data that has to be transferred, the faster it can be done. However, in conjunction with mobile devices, the processing of the data has to be considered. The problem with compressed data is that it is an intermediate format which has to be compressed and decompressed before and after sending. This adds time used to processing and thereby to higher battery consumption. In addition, the uncompressed data has a memory footprint as big as the original data, which is undesirable considering the limited memory of mobile devices. For these reasons, Binary XML may be preferred over compression, even though compression gives the smallest data.

3.5 The Web Service Protocol Stack

All applications which communicate over a network, needs some common understanding of how the communication should be performed, and this is what protocols are all about. Each protocol defines some rules that dictate the communication when that protocol is used. Different protocols perform different tasks and have different characteristics. For that reason, a combination of protocols will in most cases be needed to achieve the desired communication. The combination of different protocols is often referred to as the protocol stack.

The protocol stack used for communication between the Web Service and its client will be examined in this chapter, the aim being to make the stack perform better over wireless links used by mobile devices. The Web Service and Web Service client communicate by using the SOAP protocol. SOAP is an application layer protocol which is used to exchange XML messages between peer applications [40]. Though SOAP is independent of the underlying transport protocol, HTTP is by far the most commonly used transport protocol [41]. HTTP again is in most cases delivered over Transmission Control Protocol (TCP) and Internet Protocol (IP). Below the IP layer there are necessary protocols to make the physical transmission between adjacent nodes. Figure 3.3 shows the protocol stack used in most Web Service implementations today.

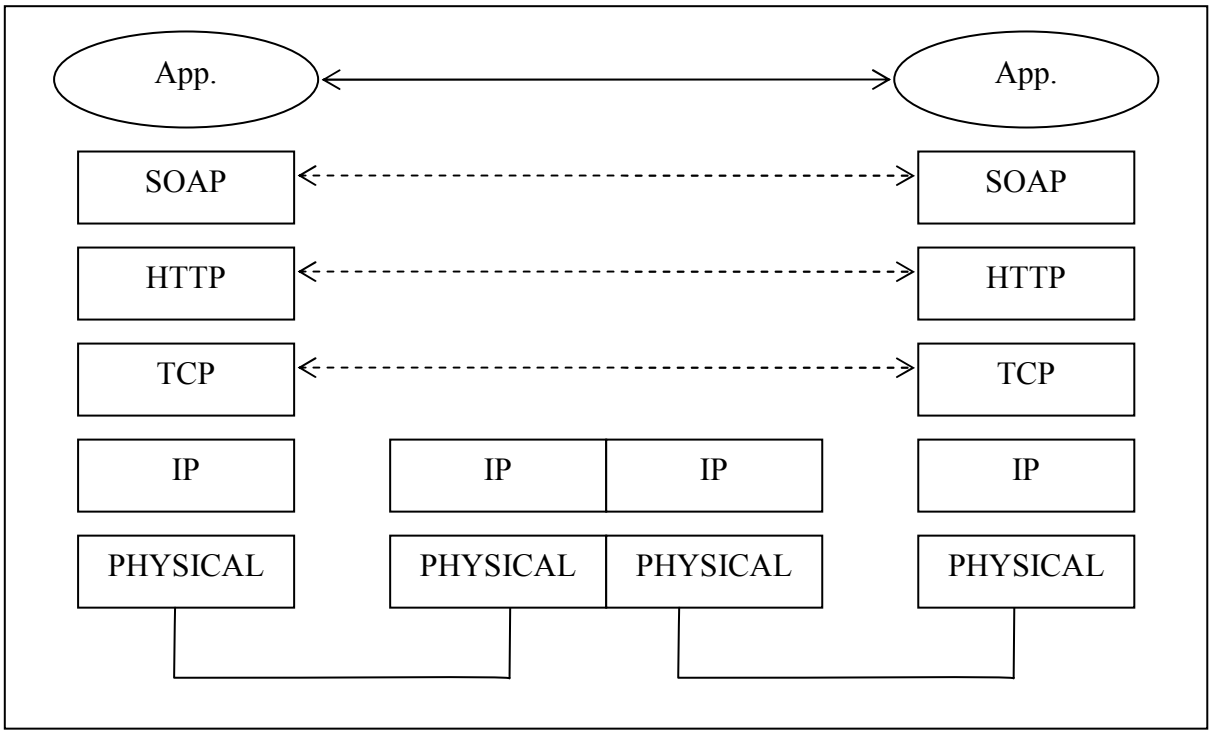

Figure 3.3 The protocol stack most commonly used for Web Services.

3.5.1 IP and the Layers Below

The IP protocol gives the necessary information to route a data packet between the peer applications. Since it is connectionless, every packet carries address-information that has to be processed on every node along the way from the sender to the receiver [35]. This is illustrated in the middle of Figure 3.3 in the previous section. The IP layer does not provide any end-to-end functionality, and it has no way to now if packets get lost or arrive at the receiver out of order etc. Such functionality has to be provided by the layer above IP in the stack. However, as IP and the layers below are necessary to transmit data packets from the sender to the receiver and moreover can not easily be changed or altered, they will not be examined closer in this study.

3.5.2 The Application Layer

The topmost arrow represents the communication that is needed for the peer applications to execute the proper behavior. In this case the applications are a Web Service and a Web Service client. This communication is performed by using the SOAP protocol. This layer is required by Web Services so it can not be removed. As described above it is based on XML, and for that reason it should be possible to encode it as Efficient XML or Fast Infoset which are described in chapter 3.3.2 and 3.3.4. Such encoding would significantly reduce the size of the messages exchanged and hence increase the performance of Web Services.

The protocol below SOAP is HTTP which itself is an application protocol. The most common argument for using HTTP as a tunneling protocol is its ability to traverse firewalls [2]. By using this tunneling, SOAP can be passed through firewalls without additional work. This is also mentioned as one of the reason why SOAP has been widely adopted. In [2] it is shown that removing the HTTP layer and ship SOAP directly over TCP is possible, but this approach will need additional work to give a satisfying solution. One benefit of such solution is that the overhead introduced by HTTP is removed, and hence less data has to be transferred.

3.5.3 The Transport Layer

The Transport Layer is the lowest layer that may provide end-to-end functionality. The most common protocol on this layer is TCP. TCP provides a connection-oriented service to the layer above [35], and thereby it detects lost and corrupted packets, and packets that arrive out of order. In other words, it performs the necessary tasks to deliver the complete and right data to the layer above. The TCP could be replaced by UDP, which is a lighter protocol that provides a connectionless service to the layer above. However, the UDP does not guarantee that complete or perfect data is given to the layer above. This is acceptable for voice-like services where imperfect data is acceptable in contrast to data that arrives later because of retransmission.

Wireless links that are used by mobile devices are subject to quite high bit error rates compared to wired links, and this leads to both lost and corrupted data packets. Because of this, UDP seems to be unsuitable in conjunction with Web Services since it does not guarantee the delivery of perfect data. The problem is that TCP does not perform well under these conditions either. Most TCP implementations has been optimized for wired networks with very low bit error rate, and this has given a congestion control algorithm that performs bad on wireless links [42]. When this algorithm times out because packets have not arrived, most of the implementations assume that this is because there are congestion in the network. It will then reduce the sending of data in order not to worsen the situation. This is a good solution when the transmission is made only over wired links where congestion is a more likely reason than lost packet due to bit error. However, on a wireless link the most likely reason for time outs are lost packets due to bit errors, and the algorithm described above leads to an unnecessary decrease in performance. What is desired is that the lost data is retransmitted as soon as possible in stead of a reduction of data transmission because it is assumed that there is congestion in the network.

In [42] two solutions to overcome this problem are presented: The first solution, called indirect TCP, brake the TCP connection in two at the base station for the wireless link. On the wireless link, lost packets should be assumed to be a result of bit errors, and on the wired line out of the base station, congestion should be assumed. The disadvantages of this approach are the added processing at the base station and the violation of the end-to-end integrity of the transport protocol. The second solution caches the TCP information at the base station and has a snooping agent that investigates this information. If the snooping agent concludes that information is lost on the wireless link, it will try to recover from this before the congestion control at the end node times out. This solution does not violate the end-to-end integrity of the TCP connection, but if the wireless link leads to much lost data, the congestion-control at the end node will be triggered. A more extensive explanation can be found in the referred to literature.

3.5.3.1 Reflection about the Transport Layer

A transport protocol that delivers complete and correct information to the layer above is needed on the transport layer. TCP does this job, but as described earlier it has potential performance problems on wireless links. Methods to overcome this problem are proposed, but it is uncertain whether or not they are feasible and give the desired results. With the increasing number of mobile devices that connects to the Internet over wireless links, there might be a need for a new transport protocol that deals better with lost data. One way to do this could be to have a better error-correction code that would give less retransmission.

3.5.4 Remarks on Stack Optimization

Several ways to optimize the Web Service protocol stack for mobile devices have been proposed. It should be noticed that not all are guaranteed to give an increase in performance. They do, however, all seem to give a reduction in the response time for Web Services on mobile devices. Measurements are needed to give any conclusion on the possible benefits of the proposed changes.

4 Architecture and Design of the Prototype Service

The main objective of the prototype service developed was to demonstrate a way to optimize Web Services for use on mobile devices. The service should also provide functionality to measure the effect of the implemented optimization. The prototype service implemented to demonstrate this concept was a company's internal phonebook. Throughout this chapter the architecture and design of the service will be presented.

4.1 The chosen Optimization Method

In chapter 3, reducing the size of XML information and optimizing the protocol stack are identified as methods that may improve the performance of Web Services on mobile devices. After considering the two methods, reducing the size of XML information was chosen as the one to be studied more thoroughly in the prototype service. The reason for this choice was that it seemed to potentially both reduce the cost and the time of the transfer. The cost should be reduced since the wireless links utilized by mobile devices are usually charged based on the amount of data transferred, thus the smaller the transferred data size is, the less is the cost of transfer. The time needed for the transfer should also be reduced since this also is correlated to the amount of data transferred. Although the time should be reduced on both wired and wireless links, the effect is more obvious on wireless links. This is due to the fact that wireless links have limited bandwidth, meaning that it will take longer time to transfer an amount of data compared to a wired link.

As reducing the size of XML information was chosen as the optimization method, it was time to turn the attention to which techniques to apply in order to achieve this reduction. Chapter 3 outlines two fundamentally different techniques; compression and binary XML. To get a good fundament for comparison, both techniques were explored in the prototype service in addition to original XML.

ZLIB was chosen as the compression method. The reason for this was that it performs approximately equal to any other general purpose compression, and was available as an open source library. It should however be noted that compression utilities like XMill that are optimized for XML, achieve better compression. These utilities are, however, not that common, and add some processing overhead which is one of the disadvantages related to compression on mobile devices.

The choice of Binary XML format was between Efficient XML (EFX) and Fast Infoset studied in chapter 3.3. They are both backed by serious standardization organizations, namely W3C and ITU. Seeing that W3C rejected Fast Infoset in favor of EFX as their basis for standardizing their Binary XML format, EXI, EFX is thought to integrate better with XML. For this reason EFX is chosen as the Binary XML format in this implementation. Based on this connection between XML and EFX in conjunction with the success of XML, it can be expected that the format standardized by W3C will be the most widely adopted in the future.

The last format that was implemented, was EFX which had been compressed. This format was given the abbreviation EFX_ZIP.

With the optimization methods in place, it was time to turn the attention to the architecture and design of the phonebook prototype service that should implement these methods.

4.2 Architecture

The starting point of the service development was to identify the main parts of the service and the infrastructure needed to achieve the desired service goal. Figure 4.1 presents the main architecture of the phonebook application. We can take a closer look at each part of the architecture to get a clearer picture of the service. At the right hand side of the figure we have the data repository. It is responsible for storing all the address information the company keeps. The phonebook service is a Web Service that offers the ability to search the data repository for phone numbers. In principal this can be realized in many ways. Two possible solutions would be either to realize the data repository as a database on another machine or to implement it as a part of the Web Service. To the left in the figure we can see the mobile device which has a phonebook application installed. This application performs requests of phone numbers to the phonebook Web Service.

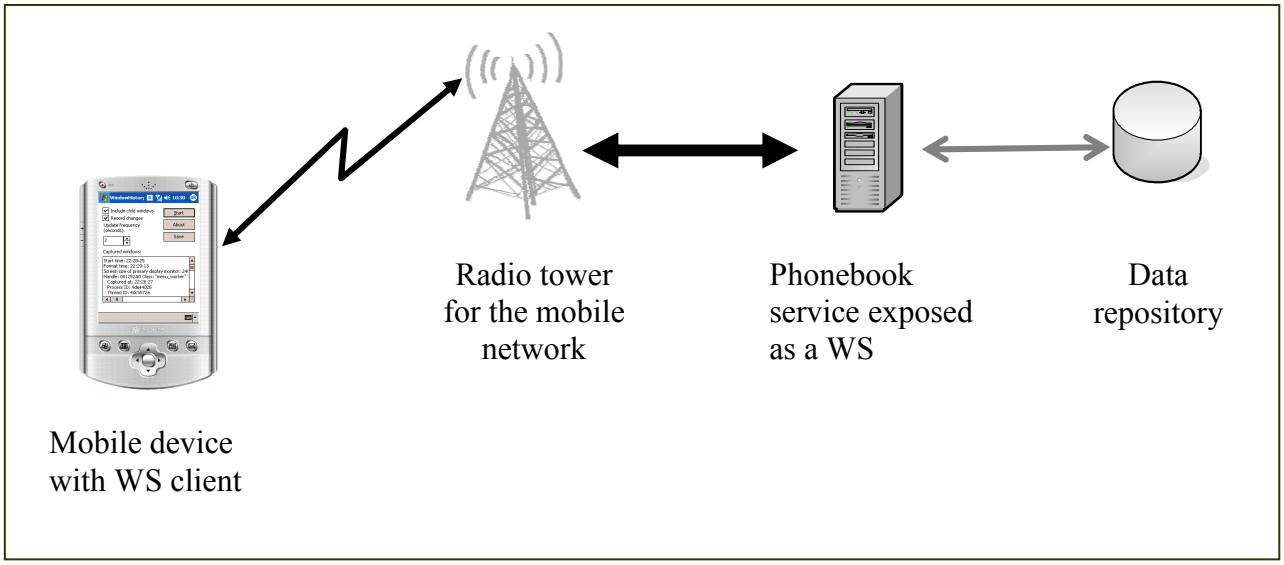

Figure 4.1 The architecture of the phonebook Web Service.

In Figure 4.1 there are three arrows which all indicate possible network links. The one to the right may or may not exist, and if it exists it should be a high performance link with almost no impact on the complete implementation. The arrow in the middle indicates a link between the phonebook Web Service placed on a server on the company's site and the mobile network. This link will in most cases be over the Internet and should be expected to have great performance, meaning that the bandwidth will not cause any problems, but as with all links some delay has to be expected. Under heavy load, the performance on this link could be poor.

The link that causes some of the challenges related to mobile devices is the one to the left in the figure. It is a wireless link, indicated by the broken arrow. Wireless links will in most

cases have considerable larger delay and lower bandwidth than wired links. A lot of the links intended for mobile devices perform poor from a data transfer point of view.

4.3 Basic Functionality of the Service

The prototype service used in this study was a phonebook Web Service. The service provides an interface which takes a name as input and returns the phonebook entries which satisfy that name, to the Web Service client. Figure 4.2 shows this usage pattern. The figure uses the term AddressBook rather than PhoneBook since this is a more general term, and services like this in most cases will provide more than just phone numbers. It will in fact be necessary in most cases to provide addresses in conjunction with the phone number to enable a positive identification of the right person.

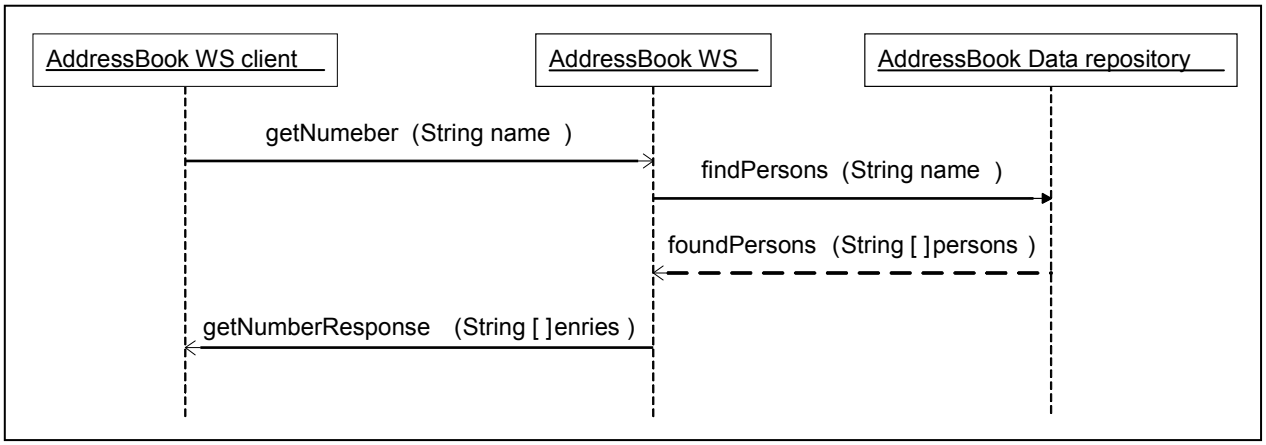

Figure 4.2 Usage pattern of the AddressBook Web Service

Now that basic functionality for the service is explained, we can have a closer look at how the desired functionality was achieved, and how the proposed optimization methods were realized.

4.4 Prototype Service Goals

The objective of the service was to show how different techniques to reduce the size of XML information affect the performance of Web Services on mobile devices. Though there has been less focus on developing a fully functional service, the service should provide functions that emulate a real life service. A phonebook service, as chosen, forms a good platform for these objectives. First, it is realistic, and therefore the information transferred from the service to the mobile device forms a great base for comparison between formats. On the other hand it is fairly simple, and it was easy to keep the functionality at a minimum and remaining focused on the measurements that should be performed. There are several different methods and

techniques to reduce the size of XML. The methods that were chosen to implement were original XML, ZLIB, EFX and EFX_ZIP as described in sub-chapter 4.1.

One decision that had to be made was the type of the returned result. The original XML data fit easily into the XML based SOAP messages, whereas the three other formats are all binary, which leads to problems related to Web Services on mobile devices as described below. In most cases this would be a problem that would be solved during implementation, but because of the impact this has, it was chosen to deal with it at an earlier stage. The target platform for the implementation on the mobile device was the Java ME and its Web Service extension, the JSR 172. The problem with this API is that it neither supports byte arrays nor SOAP messages with attachments based on the Multipurpose Internet Mail Extensions (MIME) multi-part message types [15]. In [43] a solution to this problem was outlined; it suggests that the binary data can be encoded as a base64 string. The drawback of this is that it introduces an increase in size of 1/3 of the original size. It was, however, decided that as long all formats were encoded in the base64 format, the comparison would still be fair, and hence this was chosen as the solution.

The next decision that had to be made was whether to have one message for each of the formats, or to have one message and to distinguish between the desired return formats based on a parameter in the request message. It was decided to have a request response pair of messages for each format. The reason for this choice was that it would be much clearer than to hide such information inside a parameter, and therefore less error prone and easier to debug. One possible drawback of such a solution is that it does not permit future addition of other formats. To deal with this, an extra request response pair of messages was included for later extensions.

So, in short, the service should have five operations and hence five request return pairs of messages corresponding to each format as shown in Table 4.1.

Table 4.1 Addressbook Web Service request and response messages

Encoding	Request message	Response message
PLAIN	getNumberPlainAsString(String name)	getNumberPlainAsStringResponse(String numberPlainAsString)
ZLIB	getNumberZlibAsString(String name)	getNumberZlibAsStringResponse(String numberZlibAsString)
EFX	getNumberEfxAsString(String name)	getNumberEfxAsStringResponse(String numberEfxAsString)
EFX_ZIP	getNumberEfxAndZipAsString(String name)	getNumberEfxAndZipAsStringResponse(String numberEfxAndZipAsString)
For future use	getNumberNotInUse(String name)	getNumberNotInUseResponse(String numberNotInUseAsString)

Each of the formats have an own request message to send requests to the Web Service for address information. The name of the person one wants address information about, is sent with these messages and the address information corresponding to that name is then returned in the requested format in a response message. There is one response message for each format, and this indicates the encoding used for the address information it relays. So if the same name is requested in different types of messages, the same result will be returned in different formats by different messages.

With most of the service functionality in place, it was time to have a look at the basis for the address format that should be returned.

4.5 The return format

The return format is the format in which the address information is returned by the *AddressBook* Web Service when it receives a request from a client as described above. This format is initially encoded as classic XML content, and is presented in this sub-chapter. The four different formats that are returned by the Web Service are all derived from this original content.

The idea behind the original format was that it should be fairly simple, but at the same time form a realistic set of information for an address book service. With this in mind, it was decided that the format should consist of first name, last name, phone number, email, address, postal code and town. It is possible to argue that this is too little information; however it was felt that this forms a good basis and at the same time keeps it as simple as possible. It is important to keep in mind that this is a prototype service and is not meant to provide a complete functionality. The XML schema derived based on these constraints is shown in Figure 4.3. The schema consists of a *ContactBook* root element. This element can have many *contacts* children of type *Contact*. The *complexType Contact* constraints the information that can and has to be stored with each contact. An example of a XML file based on this schema is shown in Figure 4.4. As indicated, the file can have any number of contacts stored in it. The other representations of the information that are needed for the measurements are either based on this format or derived from this. The way this was done will be presented in more detail later in this report.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://johnsrud.no/AddressBookSh"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://johnsrud.no/AddressBookSh">

    <xs:element name="ContactBook">
      <xs:complexType>
            <xs:sequence>
                  <xs:element name="contacts" type="tns:Contact"
                  minOccurs="0" maxOccurs="unbounded"></xs:element>
            </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:complexType name="Contact">
      <xs:sequence>
            <xs:element name="firstName" type="xs:string"> </xs:element>
            <xs:element name="lastName" type="xs:string"></xs:element>
            <xs:element name="phoneNumber" type="xs:int"></xs:element>
            <xs:element name="eMail" type="xs:string"></xs:element>
            <xs:element name="Address" type="xs:string"></xs:element>
            <xs:element name="PostalCode" type="xs:int"></xs:element>
            <xs:element name="Town" type="xs:string"></xs:element>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Figure 4.3 XML Schema, which defines the information returned by the *AddressBook* Web Service.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:ContactBook xmlns:tns="http://johnsrud.no/AddressBookSh"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://johnsrud.no/AddressBookSh xsd0.xsd" >
      <contacts>
            <firstName>Anne</firstName>
            <lastName>Borgundvaag</lastName>
            <phoneNumber>98032836</phoneNumber>
            <eMail>borgundvaag67@stud.ntnu.no</eMail>
            <Address>Uranienborgv. 11 E</Address>
            <PostalCode>0351</PostalCode>
            <Town>Oslo</Town>
      </contacts>
      <contacts>
            <firstName>Anne</firstName>
            <lastName>Clausen</lastName>
            <phoneNumber>94332503</phoneNumber>
            .
            .
            <eMail>wikborg75@stud.ntnu.no</eMail>
            <Address>Haukelibakken 8</Address>
            <PostalCode>0775</PostalCode>
            <Town>Oslo</Town>
      </contacts>
</tns:ContactBook>
```

Figure 4.4 XML address information for the *AddressBook* Web Service.

4.6 Main design

Now that most of the fundamental choices have been described, it is time to have a look at the overall design of the service. The service has two main parts, namely the server side and the client side. Both of these can in the next round be divided in two. Each side will have one part related to Web Service communication, in addition the server side will have business functionality and the client will have user interaction functionality.

The Web Service part on both the server and client side is almost completely depended on the targeted platform and the Web Services Description Language (WSDL) file. Platforms that support Web Services will in most cases provide a utility to generate the most essential code related to the Web Service based on the WSDL file. Since this was the case for the service described here, the design will not describe the internal structure of the generated code, but instead treat it as one class. The description of the generated code will be given in more detail in the chapter related to the implementation later in this report. The design of the server and the client side will now be described separately.

4.6.1 Server Design

Since the focus of this study was to investigate the performance on the mobile device it was decided to keep the server side as simple as possible. The straightforward way to do this, was to include the data repository as a class on the server. In Figure 4.5 one of the five message exchanges already described are presented.

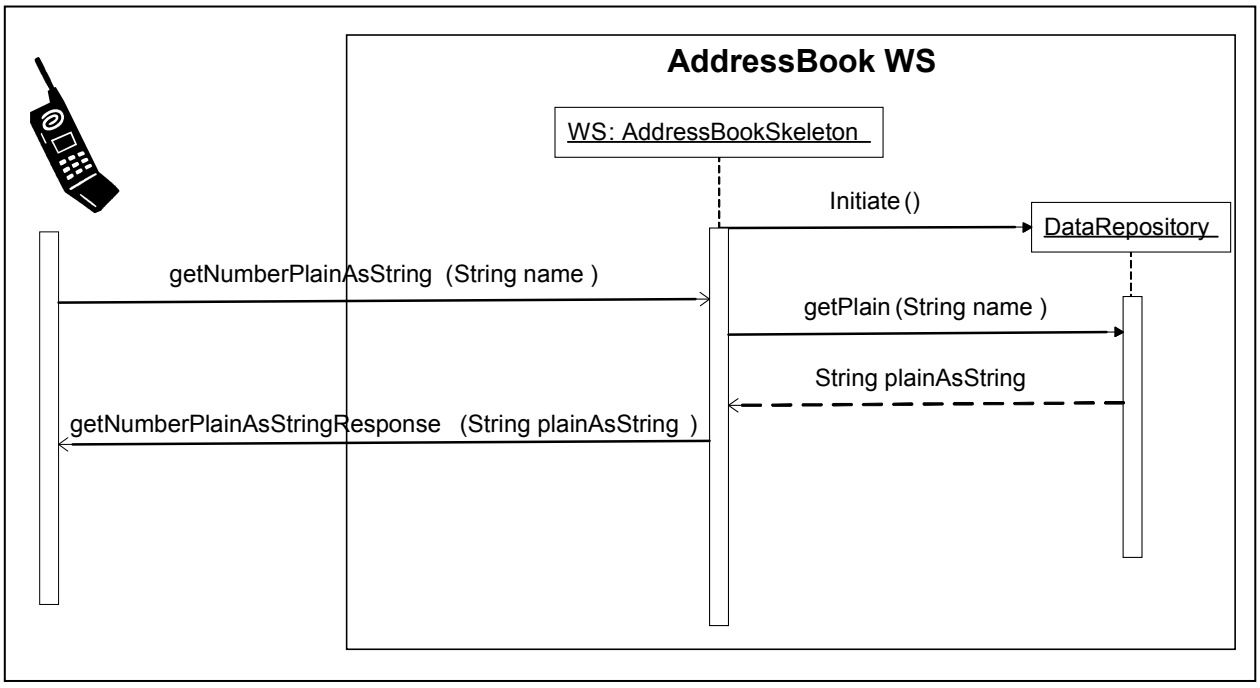

Figure 4.5 Usage pattern of the *AddressBook* Web Service.

The part that is restricted by the box labeled *AddressBook Web Service*, is the part that makes up the server side. The class diagram for the server side is shown in Figure 4.6. The Web Service part is represented by the *AddressBookSkeleton*. This is the only automatically generated class on the server side which has to be modified. In addition one more class was needed, the *DataRepository*. The *initData()* method reads all the address information into the data repository and stores it in an array for each format. The information was divided by names, meaning that the data repository has a finite set of test data that was separated based on first name of the persons stored. For each of these sets the necessary processing was performed to store the information in the four formats under study. This means that when a request is received for some information in one of the formats, the only job that had to be done was to look for this information in the array which stores this format. To retrieve some information from the data repository, four methods were supplied; one for each format. In this way, all the information was preprocessed and should take as little time as possible to return. By designing the data repository in such a manner, it should not have any significance on the time to retrieve information from the address book Web Service.

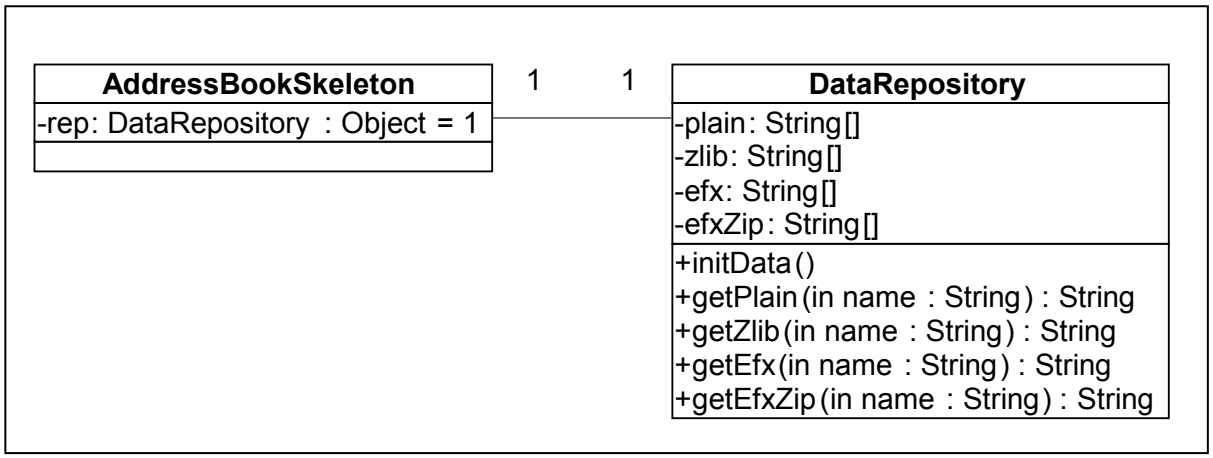

Figure 4.6 Class diagram for the *AddressBook* Web Service.

4.6.2 Client Design

Now that the design of the server side is in place, it is time to turn our attention to the client side of the service that was to be deployed on the mobile device.

As described earlier, the Web Service related communication that had to be performed on the client side would be performed by classes that had been generated from the WSDL file. On the client side, the class that enables the communication with the server is denoted the stub, and for this specific service, *AddressBookStub*. The application can make calls on the stub and when the communication with the Web Service is finished the stub will return the result [44]. Since the stub will block until the call is finished, all calls on the stub should be performed in an own thread. In addition, it was identified early that a good solution would be to have an own class to keep control of the application, handle the user input and process the results retrieved from the service. It was also concluded that it would be a good solution to have a

class that presented the results on the screen. The above requirements lead to the class diagram shown in Figure 4.7.

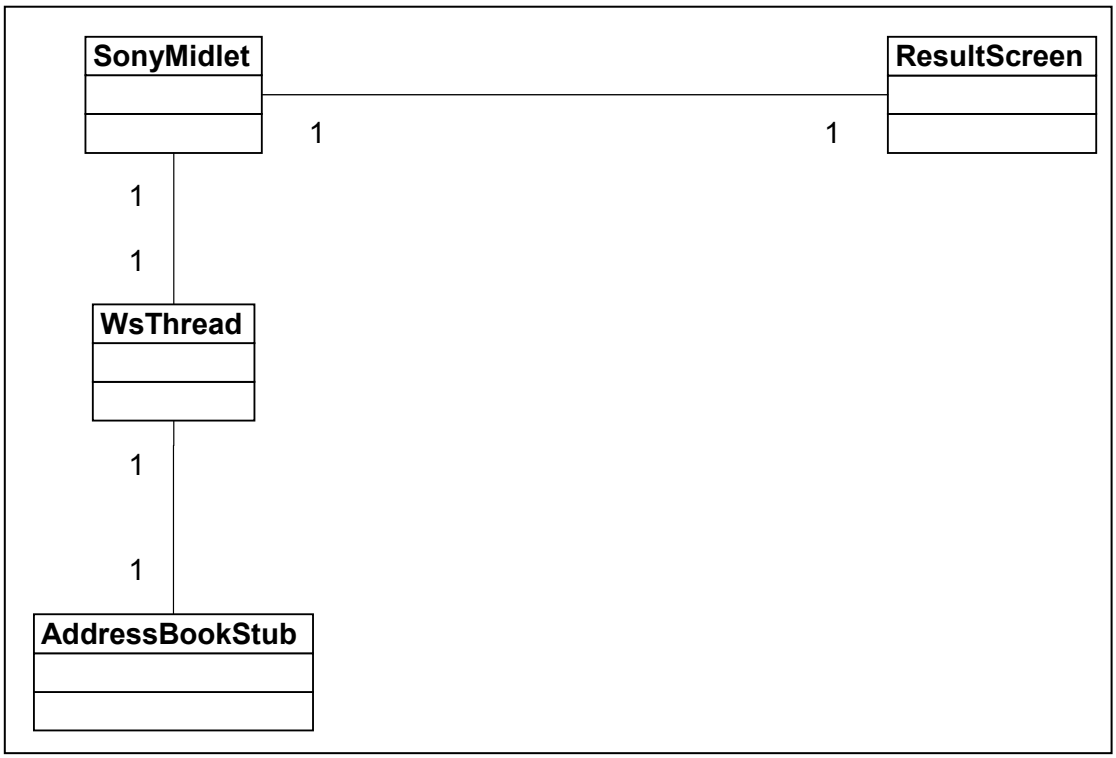

Figure 4.7 Basic class diagram for the *AddressBook* Web Service client.

A typical usage scenario for the application will lead to the following behavior on the client side:

1. The user types a name to find information about on his/her device.
2. The user selects and activates the button corresponding to the method of interest.
3. The *SonyMidlet* class starts a *WsThread* and supplies it with information about type of method to use and the name that should be looked up.
4. The *WsThread* performs a call according to the supplied information on the *AddressBookStub*.
5. The *AddressBookStub* performs the necessary action to retrieve the information from the service.
6. The call on the *AddressBookStub* returns in the *WsThread*, and the *WsThread* relays this information to the *SonyMidlet*.
7. The *SonyMidlet* performs the necessary processing of the result and presents it on the *ResultScreen.*

One important aspect of this service was to measure the time it took to retrieve information from the service. This time is measured from the call on the stub is performed in number 3 until it returns in number 6. A class diagram that describes the main parts of the client is presented in Figure 4.8.

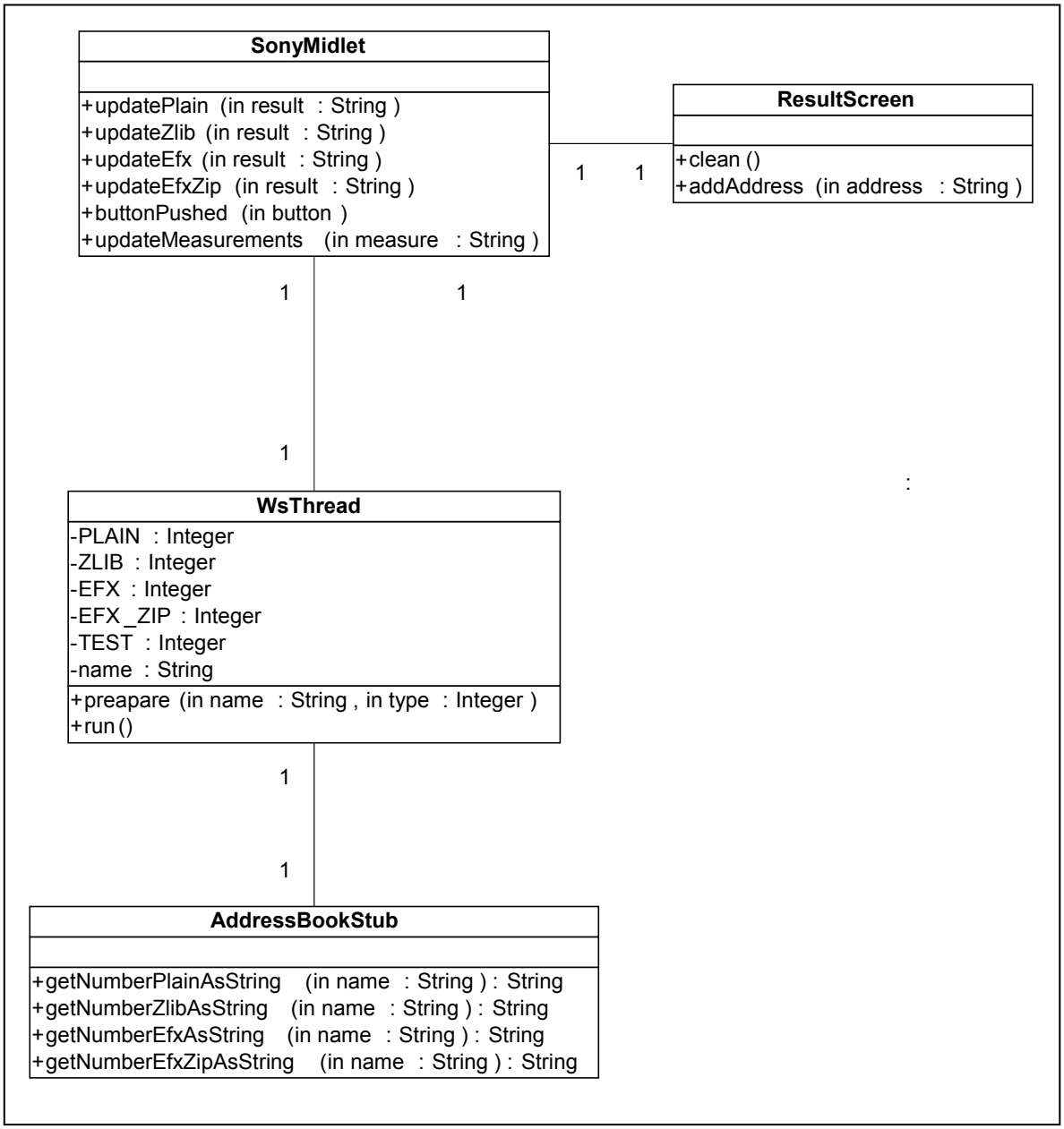

Figure 4.8 Class diagram for the *AddressBook* Web Service client

The design presented in this sub-chapter forms the basis for the implementation of the address book client presented in the next chapter.

4.6.3 WSDL file for the Address Book Web Service

As the name Web Service Definition Language, WSDL, indicates, this is a file that describes the Web Service. WSDL is a XML format that is used in order to define how to access Web Services. In short it describes where a service is and how to access it [6]. Below, a brief description of the design of the WSDL file for the Address Book Web Service is given.

The WSDL file was designed using Eclipse SDK [45], and its Web Tools Platform (WTP) [46]. This tool provides a good graphical user interface for developing Web Services. A screenshot of the design of the WSDL file is given in Figure 4.9.

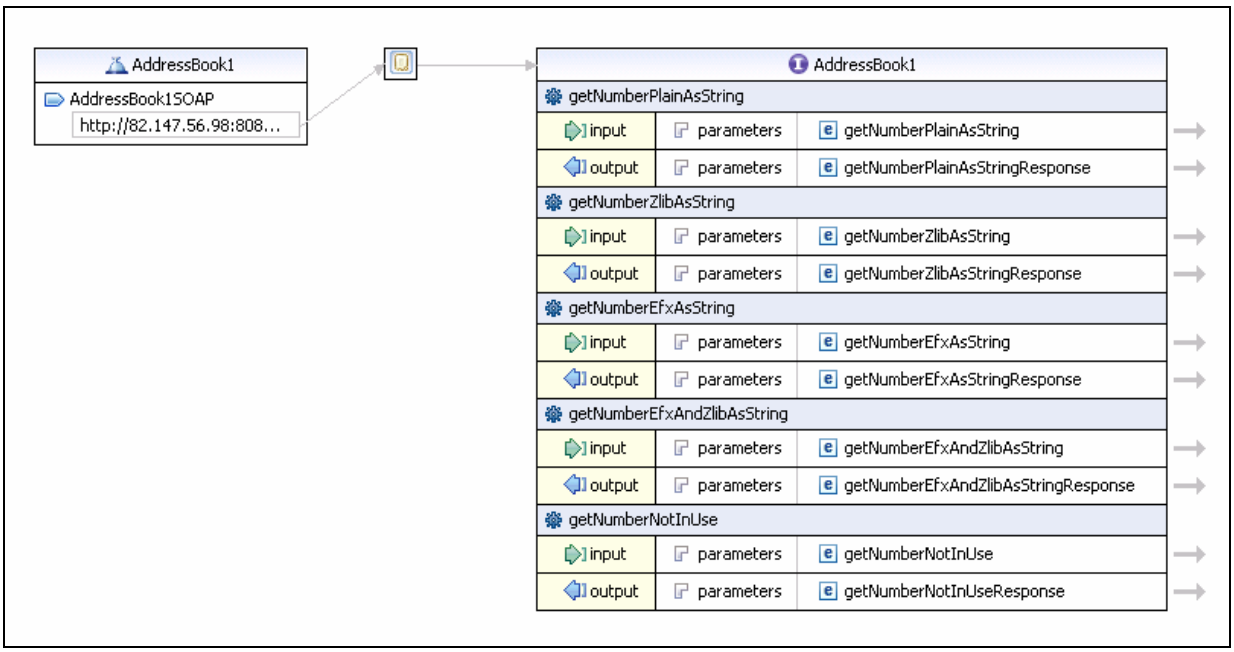

Figure 4.9 The essential parts of the *AddressBook* WSDL file.

As can be seen in Figure 4.9, the tool provides a straightforward way to design a WSDL file. An approach like this is also less error prone compared to writing the WSDL file without any tool assistance. To the left in the figure the SOAP port is specified, in the middle the binding is specified and to the right the operations are added. Each operation has one input and one output message corresponding to those given earlier in Table 4.1. By following the arrows to the right in the figure the content of the messages can be specified. The textual representation of the WSDL file can be found in appendix A.

With all the parts of the design in place it is time to have a look at how it was realized in the implementation. Throughout the next chapter the implementation will be presented.

5 Implementation

The implementation phase included the implementation of the client and the server, as well as generating the test data. In this chapter the implementation phase will be presented in detail. The Eclipse tool [45] was used for the development, mainly because it is an open development platform

5.1 Hardware and Software

Both the hardware and software that formed the target of the service is available off-the-shelf. The hardware and software are not the main focus, but are however required, so a brief review will be provided.

5.1.1 The server platform

The computer that served as the server had the following hardware configuration;

- AMD Athlon 64 3200+ CPU
- 512 MB RAM
- Network interface card of 100 Mbps.
- Internet connection with download speed approximately 3400 kbit/s and upload speed approximately 600 kbit/s.

The operating system on the server was Microsoft Windows XP professional version 2002. This formed the environment for the rest of the software, and was a natural choice both because of its widely use and the writer's previous experience with it.

In conjunction with the Web Service development, the choices were more open. Because of the writers' previous experience with it, and that the client should be written in it, java was chosen as the development language. However it should be clear that the fact that the client should be written in java itself does not dictate that the server should. The fact that the communication is totally platform independent is one of the main strengths of Web Services. So with the target language in place the search for the necessary software could start. The first additional software that was installed was the java SDK [47]. This consists of the java run time environment that forms the platform to run java applications and tools that enables the development of java applications. During the work the Java Development Kit (JDK) 1.6.0 was used.

The writer has had good experience with software released by the Apache Software Foundation [48] from earlier projects. This foundation develops open-source software for many different purposes. Their latest Web Service engine at the time of performing this work was the Apache Axis 2 [49], which also has a code generation plug-in for the Eclipse which was the development tool of choice. The version of Axis 2 used, was the 1.1.1. The Apache Tomcat servlet container served as the container for Axis 2. Apache Tomcat version 5.5.20 was used during this work.

After all the installations were done the setup was tested by pointing a web browser to the address of the server, and the response showed that all were functioning as intended.

5.1.2 The client platform

With the server up and running it is time to have a look at what formed the platform on the client side. The client should be targeted at a mobile device, and at time of performing the implementation the writer was in possession of a Sony Ericsson K610i. After some investigation it was clear that this device would do the job. The K610i has the ability to access the Internet both by GPRS and UMTS connections [50], and thereby it can access the server. It also supported both the Java ME profile MIDP 2.0 and the JSR 172 Web Service API [50].

In addition three additional libraries were needed to implement the functionality described in the design. First of all, the information retrieved from the server should be presented as a base64 encoded string. There was no initial support for this in the platform, so a decoder that would run on the MIDP 2 profile was downloaded from the Internet [51]. In addition, both ZLIB and EFX needed some external libraries to be processed on the mobile device. A java ZLIB implementation was therefore downloaded from the Internet [52]. As this implementation depended on the class *FilterInputStream* which was not part of MIDP 2 profile, another search on the Internet was needed and the file was found [53]. This file was added to the same package as the ZLIB to ensure that the compression algorithm functioned as intended. The last library that was needed was the efficient XML libraries. These were needed to parse the binary XML format, efficient XML, on the mobile device. A 30 days full functional trial version of these libraries was downloaded from their publisher AgileDelta [31]. All the tools and libraries mentioned above formed the basis for the development of the client side of the Web Service.

5.2 The test data

To perform the measurements described in the following chapter 6, there was a need for quite a lot of test data. This data should be in the format described in chapter 4.5. In order to get realistic sets of data, vCards from the Norwegian Internet page www.telefonkatalogen.no [54] were downloaded. vCard is a standardized digital format for storing and exchanging information like name, phone number and address about individuals [30].

For each test set a search was made on a name, for example Anne. Based on the results from this search, the vCard for as many people as needed to get the desired size of the test set was downloaded to an own directory. This procedure was then repeated until the desired number and size distribution of the test sets were reached.

A small utility java class was written to read the vCards and convert them into the desired XML format. This utility class also generated the email addresses since almost none of the vCards had this information. The email addresses were generated based on the last name and a number. Each one of the test sets was represented as its own XML file.

In addition to original XML, the information should also be represented in three more formats; ZLIB, EFX and the combination of EFX and ZIP. It was decided that the XML files should be compressed to ZLIB as they were read into the data repository of the Web Service. The efficient XML SDK retrieved from AgileDelta [31], included a command line tool to convert XML files to booth EFX and EFX_ZIP. Since this would do the job and was the straightforward way, it was the method of choice.

When the efficient XML SDK was set up properly, it provided the necessary command line tools to do the job. Efficient XML provides two methods to encode the information. It can either encode the XML files without any more information, or it can encode them by using an efficient XML schema. The difference between the two methods, is that when a schema is supplied the resulting efficient XML file becomes smaller. The solution that required a schema was chosen here to get as small files as possible. The *GettingStarted* manual for the Efficient XML SDK described how the encoding should be done [55]. The first thing that had to be done was to compile the Efficient XML schema. This schema was based on the original XML schema that is described in chapter 4.5. The original schema was named *AddressBookSh.xsd*. The *GettingStarted* manual for the efficient XML described how to compile the schema. The following command had to be executed in the directory that stored the *AddressBookSh.xsd*:

schemac AddressBookSh.xsd

The result of this command was the *AddressBookSh.cxs*, which is the efficient XML schema for the format. This schema again can be used to encode the XML files to the smallest possible efficient XML files. The way this was done, was to stand in the directory that hold the XML files and execute the following command:

efx -schema AddressBookSh.cxs Anne.xml –o Anne.xml.efx

The result of this command is that the file *Anne.xml* is encoded to the efficient XML file *Anne.xml.efx* which resides in the same directory as the original file. The last format that was needed was the combination of efficient XML and ZIP. This could be done using the same command by adding the *zip* option. The command to encode to EFX with ZIP is as follow:

efx -zip -schema AddressBookSh.cxs Anne.xml –o Anne.xml.efx

The command to encode to EFX and EFX with ZIP was executed for all the files in the test set. The result of this was that all the test data were encoded in XML, EFX and EFX_ZIP. The server only had to read them in.

5.3 The Address Book Web Service

Most of the code of the address book Web Service was generated from the WSDL file. The axis 2 had a code generation plug-in for Eclipse, and this was used to generate the Web Service skeleton. It also generated a *build.xml* file that could be used to build the final service. The class that interfaced the functionality of the service got the name *AddressBook1Skeleton.* Some code had to be added in this to make the service act as intended. The first thing that was

done was to add an instance of the data repository. The code added is shown in Figure 5.1. The result of this addition is that the Web Service has a way to contact the data repository. Since the skeleton holds an instance of the data repository it can make method calls right on it when number requests arrive at the Web Service. The skeleton was also generated with empty methods for each of the operations described in the previous chapter.

```
private DataRepository rep;

public AddressBook1Skeleton(){
      rep = new DataRepository();
}
```

Figure 5.1 Code in *AddressBook1Skeleton* considering the *DataRepository*.

In Figure 5.2 it is shown how the skeleton handles the requests. When a request arrives at the Web Service a method in the skeleton will be called.

```
public no.johnsrud.addressbook1.GetNumberEfxAsStringResponse
      getNumberEfxAsString
      (no.johnsrud.addressbook1.GetNumberEfxAsString param20)
        {
            //Todo fill this with the necessary business logic
            throw new java.lang.UnsupportedOperationException("Please
            implement " + this.getClass().getName() +
            "#getNumberEfxAsString");
        }
```

Figure 5.2 The method skeleton for the Web Service operations in *AddressBook1Skeleton*.

The first three lines in Figure 5.2 give the method signature for the Web Service operation *getNumberEfxAsString* in the class *AddressBook1Skeleton*. The method should return an object of type *GetNumberEfxAsStringResponse*. This object holds all the information that was needed in the return message of the Web Service. The class that forms this object is also automatically generated based on the WSDL file. Also the parameter object, in this case *GetNumberEfxAsString param20*, was generated from the WSDL file. This object holds all the information received in the request message to the Web Service. The code in the *getNumberEfxAsString* that was needed to perform the intended functionality is shown in Figure 5.3.

```
public no.johnsrud.addressbook1.GetNumberEfxAsStringResponse
      getNumberEfxAsString
      (no.johnsrud.addressbook1.GetNumberEfxAsString param20)
         {
1           String name = param20.getName();
2           no.johnsrud.addressbook1.GetNumberEfxAsStringResponse res;
3           res =
4           new no.johnsrud.addressbook1.GetNumberEfxAsStringResponse();
5           String result = rep.getEfx(name);
6           res.setNumberEfxAsString(result);
7           return res;
        }
```

Figure 5.3 The method getNumberEfxAsString in *AddressBook1Skeleton*.

The code in Figure 5.3 is pretty straightforward. In line one the name is extracted from the parameter supplied in the method call. Line two through four instantiate the return object. The main part of the functionality is performed in line five where a call to the data repository is performed. The name is provided as a parameter in the *getEfx* call. The call returns the result as a base64 encoded string. This result is then appended in the return object in line six and returned in line seven. The actual functionality was delegated to the data repository, and is performed when the *getEfx* call is performed in line five. The three other operations that represent the other formats are all implemented in the same way.

With the skeleton in place it was time to implement the data repository. The data repository has one method for each format, which takes a name as a parameter, and returns the addresses related to that name as in the corresponding format encoded as a base64 string. The information in the data repository is prepared at start-up of the service. For the PLAIN, EFX and EFX_ZIP format, the only thing that had to be done was to read them in from a directory and encode them as base64 strings. When the PLAIN format was read in, it was cloned and ZLIB compression was applied, and this way the ZLIB format was obtained. Base64 encoding was applied and then all the formats were ready. The prepared information is then stored in an array representing that format, and each test set is stored as an own entry in each array.

The main part of the initializing of the test data will now be described. Figure 5.4 presents some pieces of the code. The code shown was for the data represented as EFX, but the code for the other formats was pretty much the same. In line one the array that stores the EFX test sets is initialized and in line two a counter that indicates which set is the current, is prepared. Line three gives the directory where the test data was stored. The method that starts on line five should be called one time for each test set, and the parameter name indicates the test set that should be read in. This method read the equivalent data in the different formats and stored them in the appropriate array. For clarity, only the code for the EFX format is shown. In line six the EFX encoded file is read in from the directory and in line seven it is encoded as a base64 string and stored in the appropriate location in the EFX array.

```
1     private String[] efx = new String[14];
2     private int set = 0;
3     String baseUrl = "z:/master/dev/data/";
4
5      private void preapere(String name){
6           byte[] inBytesEfx = readFile(baseUrl+name+".xml.efx");
7           efx[set]= encodeBase64(inBytesEfx);
8           set++;
9     }
```

Figure 5.4 Code for reading in the test data to the *DataRepository*.

When the data was ready, it was time to implement the methods that were needed in order to retrieve the information from the data repository. Since the data were ready, it was only a matter of finding the right test set and return it. A part of the code to do this is shown in Figure 5.5.

```
public String getPlain(String name){
      String res="feil";

      if(name.equalsIgnoreCase("lars")){
            res = plain[2];
      }
      else if(name.equalsIgnoreCase("per")){
            res = plain[12];
      }
      else if(name.equalsIgnoreCase("ole")){
            res = plain[13];
      }
      .
      .
      .
      else if(name.equalsIgnoreCase("ida")){
            res = plain[11];
      }

      return res;
}
```

Figure 5.5 Method to retrieve information from the *DataRepository*.

The code extracts the information from the appropriate place in the array based on the name in question, and returns this information. An equivalent method was implemented for each format, namely; *getZlib*, *getEfx* and *getEfxZip*. When this was done the service could be built

with the *build.xml* file, and the service could be deployed on the server. With this in place it was time to turn the attention to the client side of the service.

5.4 The Address Book Web Service Client

5.4.1 The Web Service Communication on the Client

As for the server side, a lot of the code could be generated from the WSDL file. The wireless toolkit described earlier had a utility called Stub Generator. This utility takes a WSDL file as input and outputs the required classes to communicate with the Web Service. On the client side, the class that interfaces with the rest of the application is the stub, in this particular case it had the name *AddressBook1PortType_Stub*. Classes representing the messages that are exchanged are also generated. However in this case it only was one string that was exchanged in the messages both ways, so there was no need for additional classes to represent complex message structures. The stub class provides methods for each operation the Web Service provides. In this case the name of the "person" one wants to retrieve information about, is provided in the message call on the stub, and the person corresponding to the name is returned by the method.

Since these methods perform communication over wireless networks, the potential of messages being lost or the occurrences of large delays are present. For this reason all the calls on the stub is performed in an own thread. This thread is created in the main class in the application named *SonyMidlet*. When a request to the Web Service is required, the main class creates a *WsThread* and calls its *prepare* method. The prepare method sets the type and name variable in the *WsThread*, indicating what call should be performed on its stub. When this was done the thread was started. Based on the information supplied in the prepare method the required task was performed. The code for this is shown in Figure 5.6.

```
public void run() {
      long startT = 0;
      long stopT= 0;
      String result="";

      try {
            if (type==PLAIN_STRING){
                  startT = System.currentTimeMillis();
                  result= wsStub.getNumberPlainAsString(name);
                  stopT = System.currentTimeMillis();
                  this.parent.updatePlainAsString(result, stopT-startT,
                  name);
            }
            else if(type==ZLIB){
                  startT = System.currentTimeMillis();
                  result=wsStub.getNumberZlibAsString(name);
                  //System.out.println();
                  stopT = System.currentTimeMillis();
                  this.parent.updateZlib(result, stopT-startT, name);
            .
            .
            .
            else if(type==TEST){
                  test(50);
            }
```

Figure 5.6 The run method in *WsThread*, used to access the *AddressBook* Web Service from the mobile device

Most of the code in Figure 5.6 is self-describing, but there are a few lines that need a bit of explanation. A call on the stub is for example the line:

result=wsStub.getNumberPlainAsString(name);

The variables *startT* and *stopT* in Figure 5.6 are used for measurements. The *startT* is set right before a method call on the stub is performed, and the *stopT* is set when the call returns. By subtracting the *startT* from the *stopT,* as shown in the line below, the time to retrieve the information is obtained;

this.parent.updatePlainAsString(result, stopT-startT, name);

This line updates the main class, *SonyMidlet*, with the retrived information and the measurement. The *SonyMidlet* provides methods to process the results retrieved from the service. Here the method to update the PLAIN format is shown, but there are equivalent methods for the three other formats also. The result encoded as a base64 string, the time to retrieve the information and the name that was searched for, are provided with the method. When these methods are called, the main class processes the result.

The type *TEST* also needs some explanation. When this type is set the *WsThread* method *test* is called. This is a method that goes thoroughly through the test sets, and retrieves the information corresponding to that test set in all the four formats the given number of times. For each time, the time to retrieve the information is measured and logged. When the test is

finished, the measurements are written to a file. This type was added to make the measurements more effective.

5.4.2 The Functionality of the Client

The above section describes the most important parts of the Web Service communication on the client side. The attention will now be turned towards the *SonyMidlet* class which can be said to control the application. When the application starts up, the *SonyMidlet* class initializes the first screen. This screen has a *TextField* where the name of the person one wants to find address information about, can be typed in. In addition this screen has one button for each format, labelled PLAIN, ZLIB, EFX and EFX_ZIP. By pressing one of these buttons, the information would be retrieved in the format corresponding to the button that was pushed. In Figure 5.7 an emulated phone with the first screen open is shown. One more button was applied to the opening screen, named *TEST*. The test button starts the test described above.

When a user wants to find address information about another person, he or she types in the name of that person and pushes one of the buttons to retrieve the information in the format corresponding to the button pushed. The *SonyMidlet* class listens for action events and when they arrive, investigates them and performs the appropriate action. In the case of an address lookup, the *SonyMidlet* calls the prepare method of the *WsThread* and starts it. The code for this is standard Midlet code, and for that reason it will not be looked into its details here.

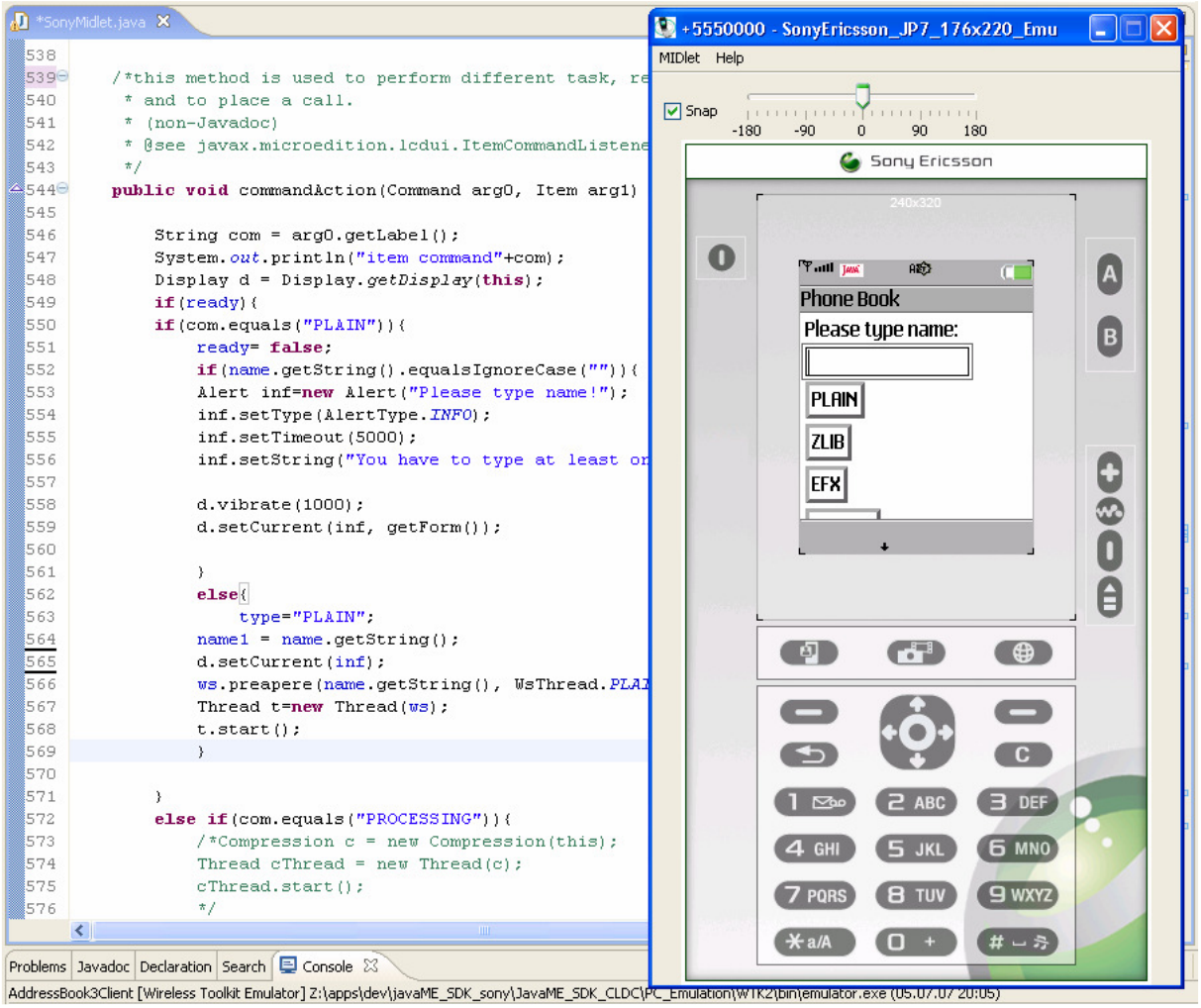

Figure 5.7 Emulated phone displaying the opening screen of the *AddressBook* Web Service client.

The *WsThread* will then wait until the message with the result from the search is received. The reception of this message leads to a method call in the *SonyMidlet* according to the format of the result. The methods with which *SonyMidlet* processes the result, are shown in Table 5.1.

Table 5.1 Update methods in *SonyMidlet*.

Format	Method
PLAIN	updatePlainAsString
ZLIB	updateZlib
EFX	updateEfx
EFX_ZIP	updateEfxAndZip

These methods should process the information and present them on an own result screen. The result screen was implemented by a class named *Result*. The *Result* class is initiated at start-up of the application and the *SonyMidlet* class keeps a reference to it. The Result class has

two methods that are worth mentioning, *clean* and *addEntry*. When any of the four methods in Table 5.1 are called, the first thing that is done is that they call the clean method. This removes any previous information from the result screen, and makes it ready to display new results. The *addEntry* method adds address information about one person, and is typically called numerous times when a result is processed. When the processing of a result is finished, the result screen is set as the one that should be shown on the mobile device.

Figure 5.8 shows the result of the search for the person Anne on the screen of an emulated phone. The arrows in the middle of the bottom of the screen indicate that there is more information both above and below what is currently presented on the screen. The call option in the lower, right corner of the screen can be used to place a call to the marked number. The back option in the low left can be used to return to the search screen.

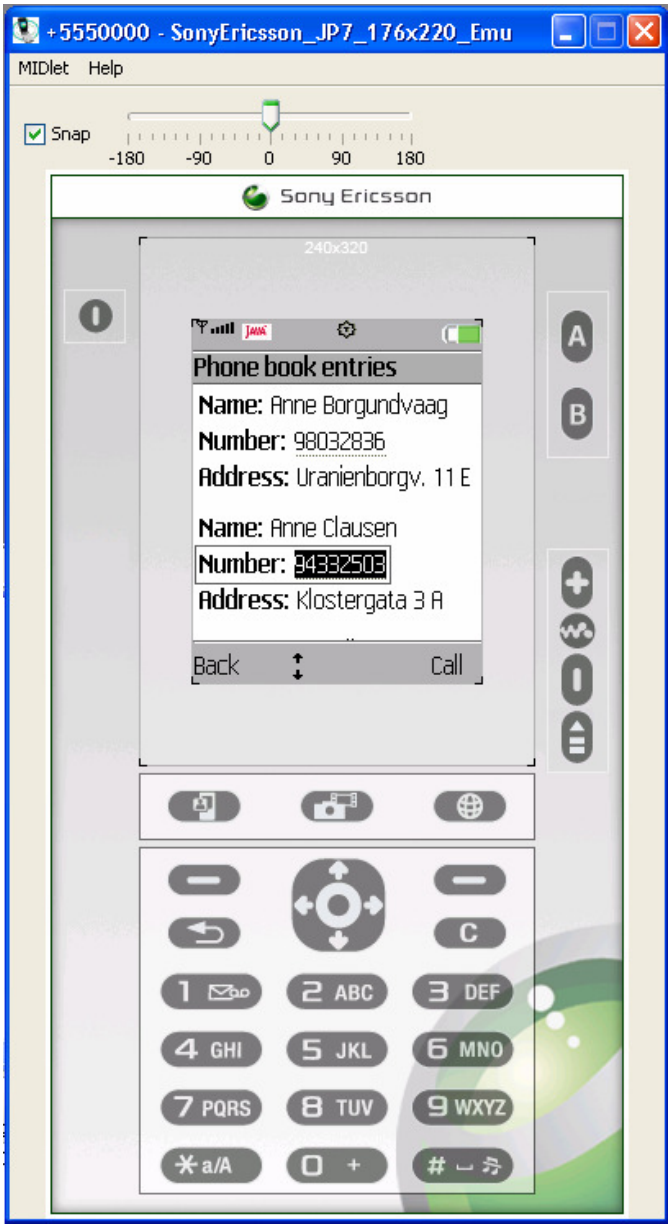

Figure 5.8 Emulated *AddressBook* client showing the result screen.

5.4.3 Processing of the Information

Now that the way the results are presented is in place it is time to have a look on the way the results retrieved from the Web Service is processed in the *SonyMidlet*. The first thing that had to be done for all the four formats was to decode the base64 string. The result from this decoding was a raw byte array representation of the format in question. This application should be able to parse two different representations of the formats, namely the PLAIN XML format and the EFX XML format. The ZLIB and EFX_ZIP formats are both based on the other format with applied compression. In conjunction with the ZLIB format, this meant that the data had to be uncompressed with the ZLIB algorithm, and after that, the exact same parsing as with the PLAIN format had to be done. The EFX_ZIP format, however, resulted in a problem that is described in chapter 5.5.

Since the parsing of the PLAIN and EFX format are the most interesting, we will have a closer look at their implementation. The two parsers used here works in different ways. The parser used for the PLAIN format was the sax parser, which is a push parser, while the one used for EFX was a pull parser. Let us first take a look at the sax parser which was used for the PLAIN format. The most essential code for this parser is presented in Figure 5.9.

```
ByteArrayInputStream bais = new ByteArrayInputStream(plainBytes);

SAXParserFactory fac = SAXParserFactory.newInstance();
SAXParser saxP =fac.newSAXParser();

saxP.parse(bais, new ContactHandler(this));

d.setCurrent(resScreen);
```

Figure 5.9 Code for the XML parser in *SonyMidlet.*

The first line in Figure 5.9 makes an *InputStream* of the byte array that represents the PLAIN XML format. In the two next lines the *SAXParser* is obtained and in the fourth line the parsing is performed. The method parse is called on the *SAXParser*. This method takes two parameters; the first is the *InputStream* that holds the information that should be parsed and the second, *ContactHandler*, is the class that implements methods the parser calls when it has parsed some information. For example, the parser calls the *ContactHandlers* method *startElement* every time it finds a start element in the XML document. The *ContactHandler* class is responsible for performing the necessary action every time this happens. The parser pushes information on the *ContactHandler*, and every time the *ContactHandler* concludes that it has information about a whole new contact it updates the result screen. The last line in Figure 5.9 displays the information on the screen.

In contrast to the sax parser, the parser for the EFX fomat is a pull parser, which means that the information has to be pulled out of the parser. Now let us have a look at some of the necessary codes to do this. In Figure 5.10 the first line creates the *EFXFactory*, and the second creates an *InputStream* of the EFX byte array. The third line creates an *EFXStreamReader* to parse the EFX document. This document is supplied as an *InputStream*

when the EFXStreamReader is created. The fourth line reads something from the document, and this line shows the main difference compared to the SAX parser used for the PLAIN format. Here the next method is called on the *EFXStreamReader* to pull information out of the document. The SAX parser would have pushed this on the application through method calls in the *ContactHandler*.

```
efx = EFXFactory.newInstance();
bais = new ByteArrayInputStream(efxBytes);
readEfx = efx.createEFXStreamReader(bais);

readEfx.next(EFXStreamReader.START_ELEMENT);
.
.
d.setCurrent(resScreen);
```

Figure 5.10 EFX parser code in *SonyMidlet.*

To read all the necessary information, the *next* method with different parameters has to be called a series of times inside a loop. For each walk through the loop, one contact is extracted, and added to the result screen. When all the contacts are extracted the last line in the figure shows the screen with the result.

This concludes the most essential parts of the implementation, however there were some problems during the implementation as described in chapter 5.5.

5.5 Problems during Implementation

The first problem encountered during the implementation phase, was related to the message sent from the server to the client. When there was no information in the header tag in the message, axis2 returned the tag;

<soapenv:Header />.

This tag is opened and closed in one line. After some search on the Internet, a forum explained the problem [56]. The parser supplied with the Java ME implementation did not support this kind of a tag, and reported a missing end tag when the SOAP message containing this tag was received. The way this was solved was to remove this tag. In the class *AddressBook1MessageReceiverInOut* that was automatically generated by axis2, the following line was added to remove the tag before the message was sent to the client:

envelope.getHeader().detach();

This solution solved the problem and the client could now exchange messages with the Web Service.

The next problem was related to the message size the Java ME application could receive. Initially it was the intention to have some large test sets, but during the testing of the application, it became evident that the Java ME application did not support SOAP messages larger than about 32 KB. Though the writer was not able to find any good explanation for this, it gave a restriction on the size of the test data, and actually led to a good result. Since the Java ME platform does not support bigger SOAP messages, it would be a good idea to use the EFX format since this can provide the same information in a much smaller format. Hence this format gives the ability to transfer much more information in one message than plain XML does.

The next problem was related to special characters and the EFX parser. In the Norwegian language the special letters æ, ø and å are used. During the conversion of the XML documents to the EFX representation, it was observed that the EFX parser did not support these characters and halted every time it reached one of them. There was also some other special characters that seemed to bring the parser to a halt. This was solved by removing the information that contained these letters, and then it functioned as it should. This solution is adequate for the purpose of this paper, but for a commercial service other solutions would be necessary.

Another obstacle was to get the EFX format to work on the mobile device. Since this was an external library with no initial support on the device, it had to be packed with the application. The problem was that if it was not packed with the application, the application would still work in the emulator, but on the actual device a *ClassNotFoundException* was thrown. This was solved by telling Eclipse to pack this library with the application.

Finally, there was a problem related to parsing the EFX_ZIP format. This format was created by setting an option that told the EFX parser to apply ZIP on the file. When the implementation on the client side was performed, the corresponding ZIP property was set to tell the parser that the document was compressed. This, however, did not work and a mail was sent to AgileDelta support with a question about this problem [57]. In the response mail, it was revealed that this property was not yet supported in the Java ME version of the EFX library. There was nothing in the documentation that indicated that this should not work. The documentation regarding this issue could therefore have been better.

With the implementation in place it was time to turn the attention to the measurements that are presented in the next chapter. More information about the source code can be found in appendix B.

6 Results and Discussion

The main focus of this thesis was to identify and evaluate techniques that improve the performance of Web Services on mobile devices. Performance in this context is a reduction in the time needed to retrieve the information from the Web Service. This time is highly related to the limited bandwidth and high latency of the wireless links mobile devices use to access Web Services. A reduction in the cost of data transfer would also improve the user-experience of Web Services.

In order to evaluate different techniques, a prototype service was developed. The prototype is thoroughly described in chapter four and five. In this chapter the measurements produced by that prototype will be presented and evaluated.

6.1 Measurements

The measurements presented in this chapter and implemented in the prototype service, are based on the approach described in chapter 3.2 and 3.3. This means that all results presented in this chapter have to be evaluated in conjunction with XML compression and Binary XML. In other words, none of the results presented here are related to stack optimization.

Four different types of data representation were implemented in the prototype service and form the basis of the measurements presented in this chapter. Plain XML format, by this it is meant pure XML as we all know it, is denoted PLAIN in all the results. The second format that is looked into is ZLIB, which is a pure compression of the PLAIN files and the data itself is not manipulated in any way. This format is labeled ZLIB throughout the results. The two last data formats are both based on efficient XML. One of these formats, labeled EFX, is a pure efficient XML representation, whereas the other format in addition to efficient XML representation also has ZIP compression applied, and is labeled EFX_ZIP.

In the following sub-chapters, 6.1.1. to 6.1.3, a short description of the three different sets of measurements is given. Each of these measurements presents some aspects of Web Services and performance related to the data being transferred. The results from these measurements are presented in chapter 6.3, 6.4 and 6.5.

6.1.1 File Sizes

The results related to file sizes presents the file sizes for the four different representation techniques which were studied. In other words, this measure shows how much smaller the test data becomes when applying different techniques for reducing the size of the data represented in the PLAIN format.

6.1.2 Data Transfer Prices

Many people would not think that the service price for data transfer to and from a mobile device is a factor with significant importance. Such an assumption is only partly true. At the moment, the prices of transferring data in UMTS, EDGE and GPRS networks are depending hugely on whether you are inland or abroad.

The prices used to compare the cost of different compressions are based on the prices of the Norwegian company Telenor [1]. These prices are presented in Table 6.1 below, and are from spring 2007.

Table 6.1 Prices for transmitting data in mobile networks in Norway [1].

Place	NOK/MB	NOK/KB
Norway*	20	0,01953125
Scandinavia	25	0,02441406
Europe	40	0,0390625
Rest of the world	112,50	0,10986328

* In Norway the users will not be charged more than 50 NOK per day. 1 EURO is about 8 NOK, June 2007.

6.1.3 Round Trip Times

The measurement set, named the Round Trip Time (RTT), measures the time from the Web Service call is performed from the mobile device until all the information is received back at the mobile device's Web Service client. This can also be understood as the response time for the service. The results from these measurements show the benefit in time, of compressing the data that will be transferred. There were performed both simulated and real mobile network measurements in relation to the RTT.

The simulated measurements were obtained by use of the National Institute of Standards and Technology (NIST) Net emulator. NIST Net makes it possible to emulate almost any kind of network [58]. To achieve the desired behavior of a network, the parameters corresponding to that network have to be supplied. The parameters used for the simulation of GPRS, EDGE and UMTS presented here are obtained from Teknisk Ukeblad [16], and are given in Table 6.2.

Table 6.2 Network parameters used for simulation of RTT [16].

Network	Upload bandwidth	Download bandwidth	One-way delay
UMTS	128 Kbps	300 Kbps	120 ms
EDGE	50 Kbps	200 Kbps	200 ms
GPRS	30 Kbps	50 Kbps	350 ms

The measurements for both the simulated and real mobile network were obtained by running the phonebook application on an emulated phone on a laptop. For the simulated measurements the laptop was connected through a Local Area Network (LAN) connection of 100 Mbps to the NIST Net network emulator. This setup is illustrated in Figure 6.1.

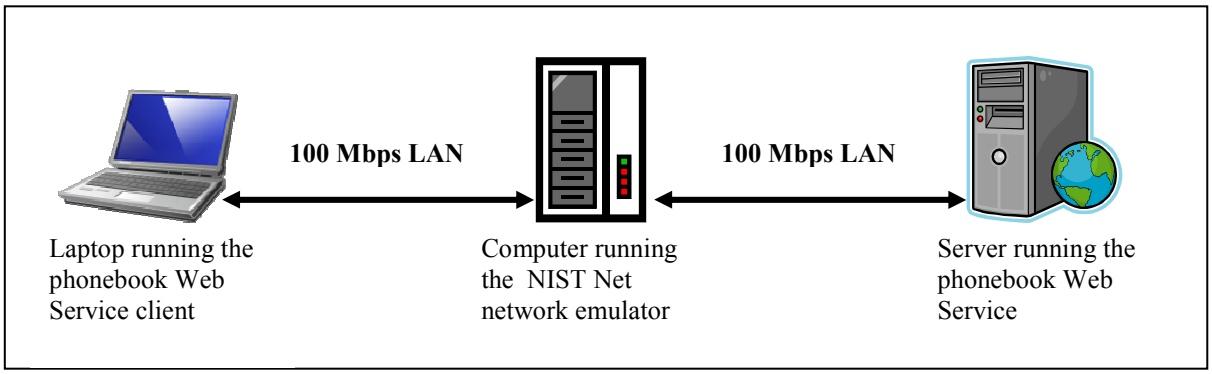

Figure 6.1 Setup for measurements of RTT in simulated environment.

When measuring the effect from a real mobile network, a mobile network card was plugged into the laptop. This gives the same network connection as a real phone utilizes, as illustrated in Figure 6.2.

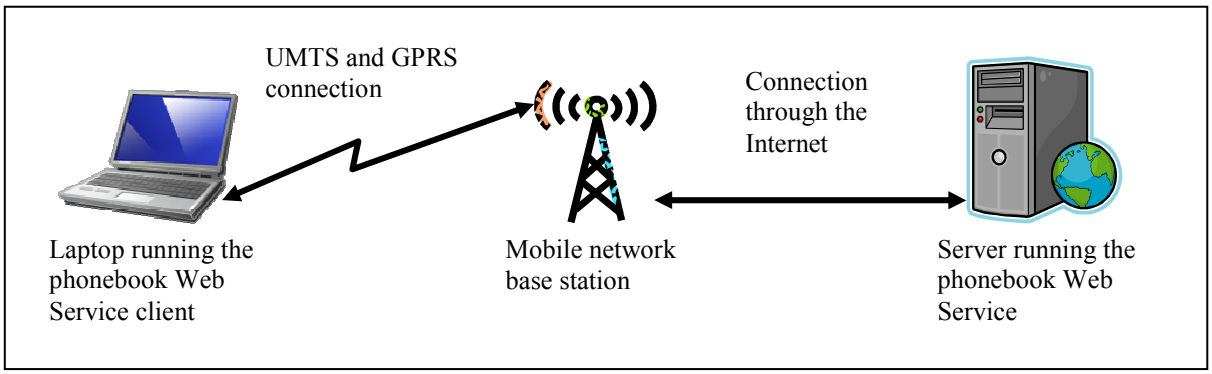

Figure 6.2 Setup for measurements of RTT in real environment.

The results from the RTT measurements are presented in chapter 6.5

6.2 The test data

To be able to perform the different measurements there was a need for some amount of test data. This data was based on the address format described in chapter 4.5, and is shown in Figure 6.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:ContactBook xmlns:tns="http://johnsrud.no/AddressBookSh"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://johnsrud.no/AddressBookSh xsd0.xsd" >
      <contacts>
            <firstName>Anne</firstName>
            <lastName>Borgundvaag</lastName>
            <phoneNumber>98032836</phoneNumber>
            <eMail>borgundvaag67@stud.ntnu.no</eMail>
            <Address>Uranienborgv. 11 E</Address>
            <PostalCode>0351</PostalCode>
            <Town>Oslo</Town>
      </contacts>
      <contacts>
            <firstName>Anne</firstName>
            <lastName>Clausen</lastName>
            <phoneNumber>94332503</phoneNumber>
            .
            .
            .
            .
            <eMail>wikborg75@stud.ntnu.no</eMail>
            <Address>Haukelibakken 8</Address>
            <PostalCode>0775</PostalCode>
            <Town>Oslo</Town>
      </contacts>
</tns:ContactBook>
```

Figure 6.3 The Address format that forms the basis for the measurements

The data used for measurements was XML files with different size on the PLAIN format as presented in Figure 6.3. A description of how these test sets were obtained is given in chapter 5.2. Initially, the files were in the PLAIN format only, and the ZLIB, EFX and EFX_ZIP formats were generated from this. The names and sizes of the test sets in the different formats are presented in Table 6.3. The sizes of the test sets in the different formats are important results in themselves, and are presented and discussed more thoroughly in sub-chapter 6.3. They do, however, also form the basis for the results presented in sub-chapter 6.4 and 6.5, and for that reason they are presented here. For a complete presentation of these files the reader can take a look in appendix C.

Table 6.3 The test data used in the measurements.

Test set	PLAIN Size in bytes	ZLIB Size in bytes	EFX Size in bytes	EFX_ZIP Size in bytes
File 1	507	303	69	69
File 2	1 556	485	295	246
File 3	2 081	595	427	346
File 4	4 471	972	979	691
File 5	6 347	1 183	1 329	908
File 6	8 769	1 567	1 902	1 252
File 7	11 883	1 892	2 485	1 580
File 8	12 913	2 063	2 690	1 719
File 9	15 291	2 425	3 303	2 046
File 10	16 280	2 482	3 558	2 060
File 11	18 134	2 591	3 805	2 264
File 12	19 125	3 033	4 209	2 596
File 13	23 608	3 301	4 903	2 874
File 14	24 309	3 584	5 267	3 127

6.3 File Sizes

The most important factor regarding improved Web Service on mobile devices is probably related to file size. The reason for this is that all the other results are somehow related to how big the data is. One of the big problems in adopting Web Services to mobile devices, is the huge overhead of traditional XML. XML files contain a considerable quantity of overhead data, so a relatively large decrease of the file size should be expected independently of which technique is being applied. In this chapter we will have a look at the effect on the file sizes of the different methods that have been studied.

The first thing that can be looked into is the size of the test data represented in different formats and with applied compression. In Figure 6.4, it is clear that all other representation than the original XML, PLAIN, produce smaller files.

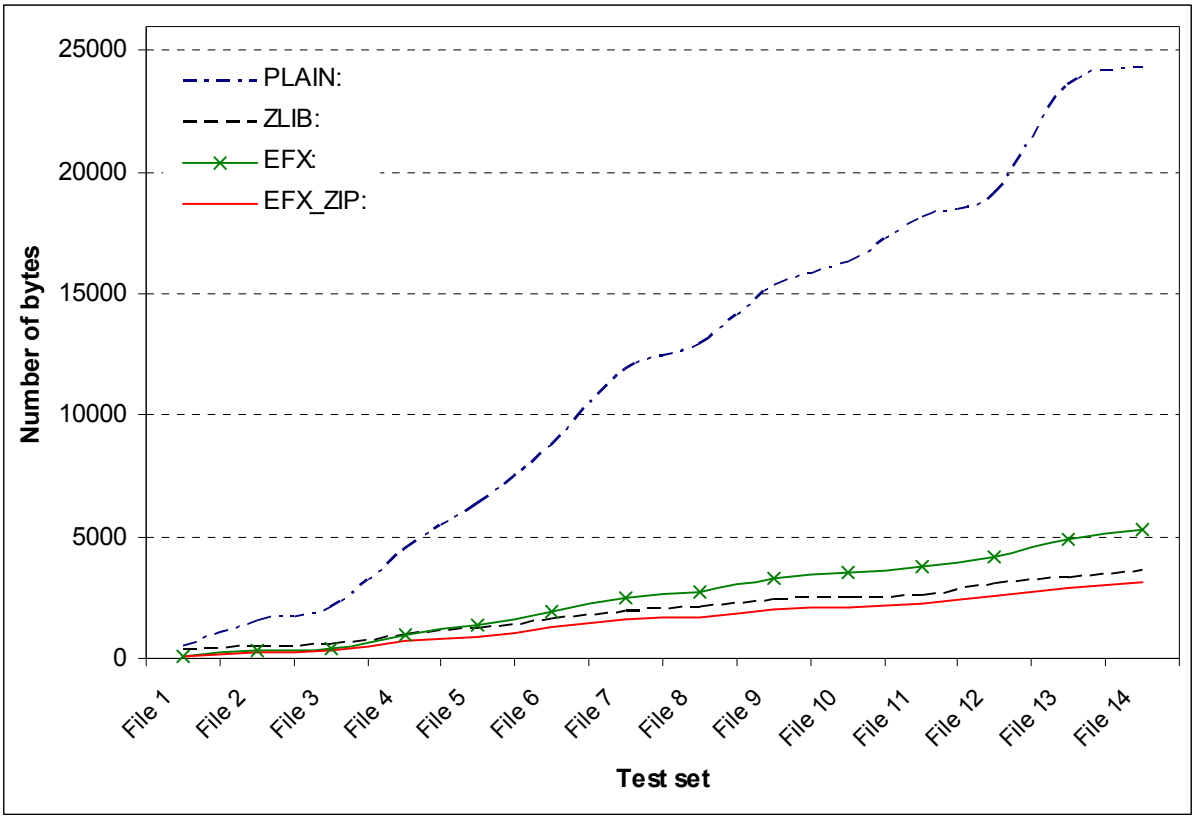

Figure 6.4 File sizes of the test data represented in different formats.

At first sight, it might seem that the ratio between the PLAIN files and the other representations increases with the file size. However, by calculating the percentage of the original file sizes, as shown in Figure 6.5, the picture alters. As could be expected, the ZLIB format performs poor when the file is small, but performs better and better the larger the original file is. Compression algorithms need a certain amount of data before their algorithms reduce the data sizes. The next approach shown in the same figure as EFX, takes a fundamentally different approach by representing the information in the efficient XML format. The principal of this format is that a lot of the overhead introduced by traditional XML is removed, which has been thoroughly described in chapter 3.3. As can be seen, this alone leads to a representation that is more than 70 % smaller than the original size.

The last method to reduce the file size is to apply compression on the EFX representation and hence both utilize the advantage of smaller original file and compression. This format is shown as EFX_ZIP in Figure 6.5, and is the one that performs best. This was as expected since it is a combination of the two other methods described.

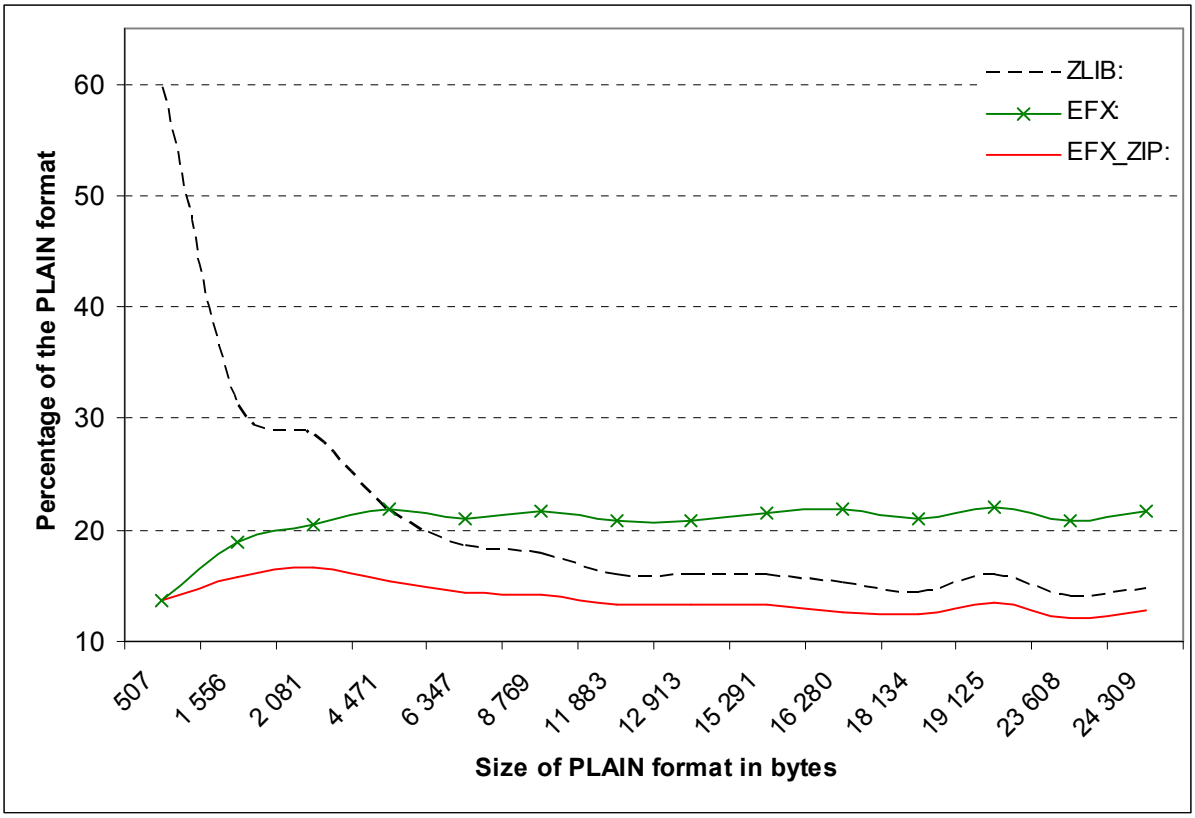

Figure 6.5 The percentage of the original XML file size when different techniques for reducing the file size are applied.

All in all there is nothing surprising with the obtained results. EFX removes a lot of the overhead and therefore initially produces much smaller data. However, no compression is applied so it will be possible to perform better. In this case the ZLIB compression performed better than EFX when the original PLAIN format exceeded about 5000 bytes. The EFX_ZIP format performs best at all file sizes since it utilizes the benefits of both of Binary XML and compression.

One other important discovery that was made during the implementation and testing of the prototype service, was that mobile devices might not support big files. The target device for the prototype service developed in this thesis, the Sony Ericsson K610i, did not support http documents larger than approximately 32 KB. This alone was an important result, because it tells us that by reducing the size of the information transferred a lot more usable information can be transferred.

6.4 Prices

The price of the data transfers alone constitutes a good reason for reducing the size of the data transferred between Web Services and mobile devices. In Norway, the use of services that transfer data are currently charged based on the amount of data being transferred. This principal prevails regardless of whether the users are in their own country or abroad. The only difference is that each MB transferred in an abroad country has a higher price, as previously shown in Table 6.1. Even though a lot of people would not consider this a big problem at the first glance, they might be shocked when the bill arrives after spending some time abroad, either on a business travel or on a vacation. To reduce this impact, it might be a good idea to apply different compression techniques on the data that will be transferred to a Web Service client on a mobile device. Figure 6.6 and 6.7 illustrates the effect in NOK of the different formats studied in this thesis. The prices in Figure 6.6 are based on the file, *File 1,* of size 507 bytes in the PLAIN format.

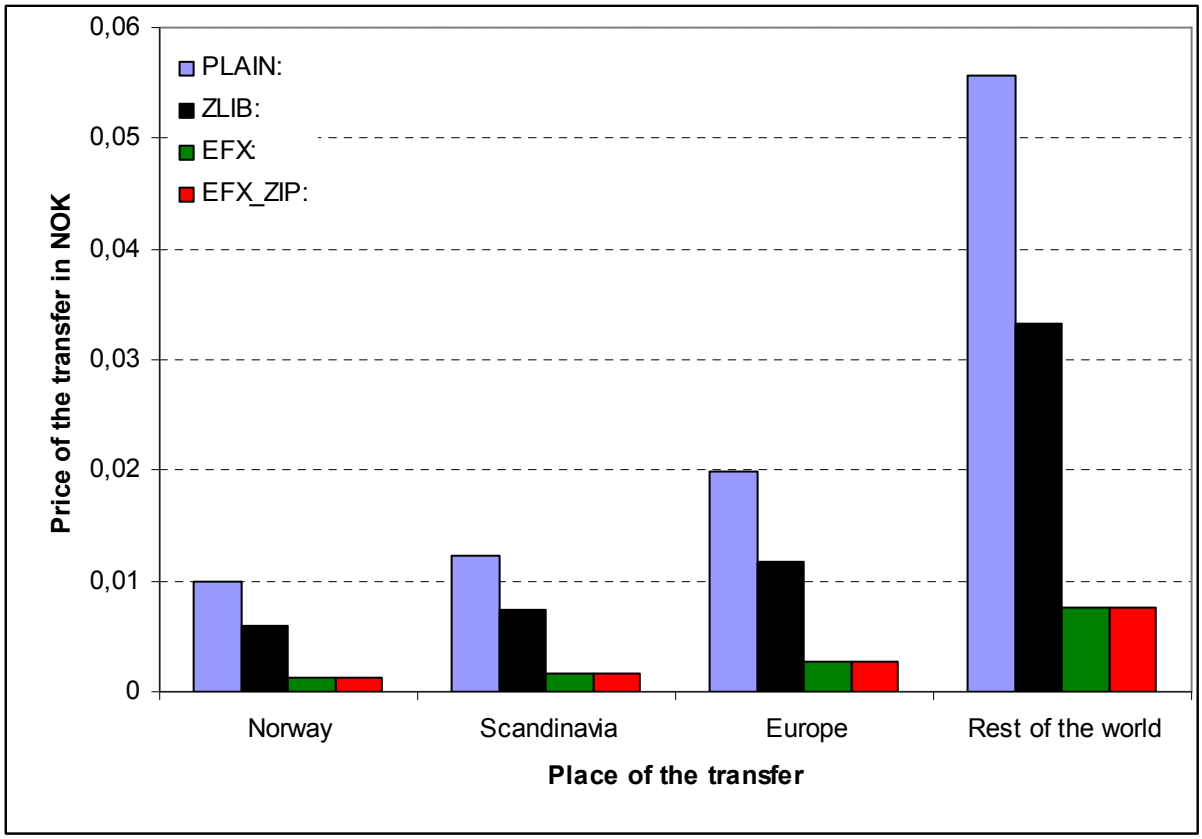

Figure 6.6 Price to transfer *File 1*, the smallest file in the test set, in different places in the world.

In Figure 6.7 the equivalent prices are presented, but this time it is the biggest file in the test set, *File 14*, of size 24 309 bytes in the PLAIN format that is presented.

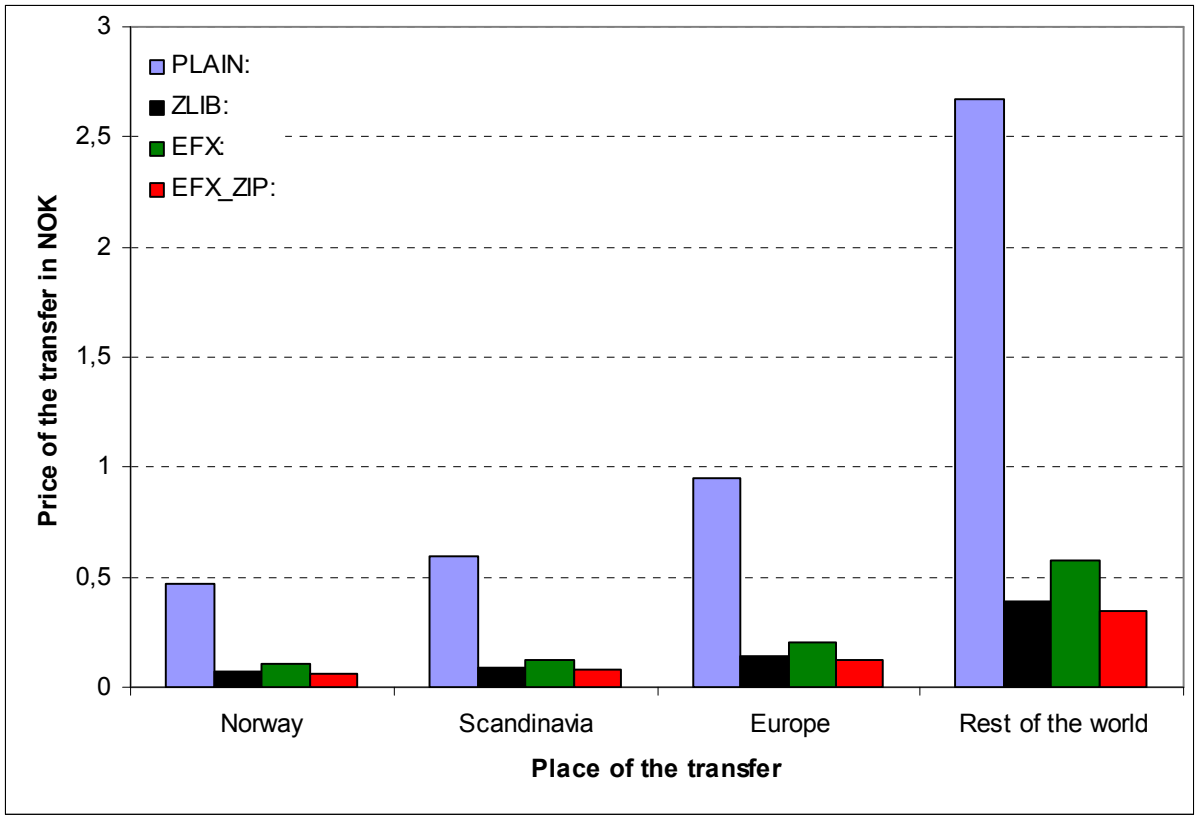

Figure 6.7 Prices of different formats based on the test set named *File 14.*

As can be seen above, there is a huge advantage in applying any of the techniques to reduce the size of the data being transferred. For both the smallest and the largest file, illustrated in Figure 6.6 and 6.7, the difference between the EFX with ZIP, EFX_ZIP, and plain XML, PLAIN, is at least a factor of 7. This means that you have to pay more than seven times more if you use a Web Service that is based on plain XML compared to one that uses a combination of efficient XML and ZIP. As can be seen in the figure, the difference between the three techniques applied to reduce the size of the data is less significant. This indicates that if one has to choose one of the techniques, other factors should be taken into account. More generally it should be understood that the price is related to the file size, and for a more general comparison of the formats, the reader is advised to read the chapter 6.3 about file size.

The single most important thing that should be noticed is that for all the test data used in this thesis, there is always one technique that reduces the size of the data, and thereby the price of the transfer by at least a factor of seven. If the amount of data being consumed by the Web Service client is huge, this would lead to a significant amount of money being saved.

6.5 Round Trip Times results

In the following sub-chapters, we will have a look at the time it takes to retrieve the information from the service in the different formats that are implemented in the prototype service. We will have a look at the three most widely deployed technologies for data transfer to mobile devices today, namely UMTS, EDGE and GPRS. Many people will argue that GPRS is an obsolete technology, this is however not the case. An example might be a user contacting a weather service from his or her cabin high up on the mountain. In situations like this, GPRS might be the only possible connection for data transfer. RTT measurements from both real connections as well as simulated connection will be presented. The parameters used for simulation in this sub-chapter are given in Table 6.2 in sub-chapter 6.1.3, but are presented here as well for readability.

6.5.1 UMTS results

For UMTS both measurements in the real network and in a simulated environment were performed. The parameters employed for the simulation were:

- upload speed: 128 Kbps
- download speed: 300 Kbps
- one-way delay: 120 ms

These parameters led to results that were very close to the ones obtained in the real network. The diagram for the simulated result can be seen in Figure 6.8, and is based on the average of 6 measurements.

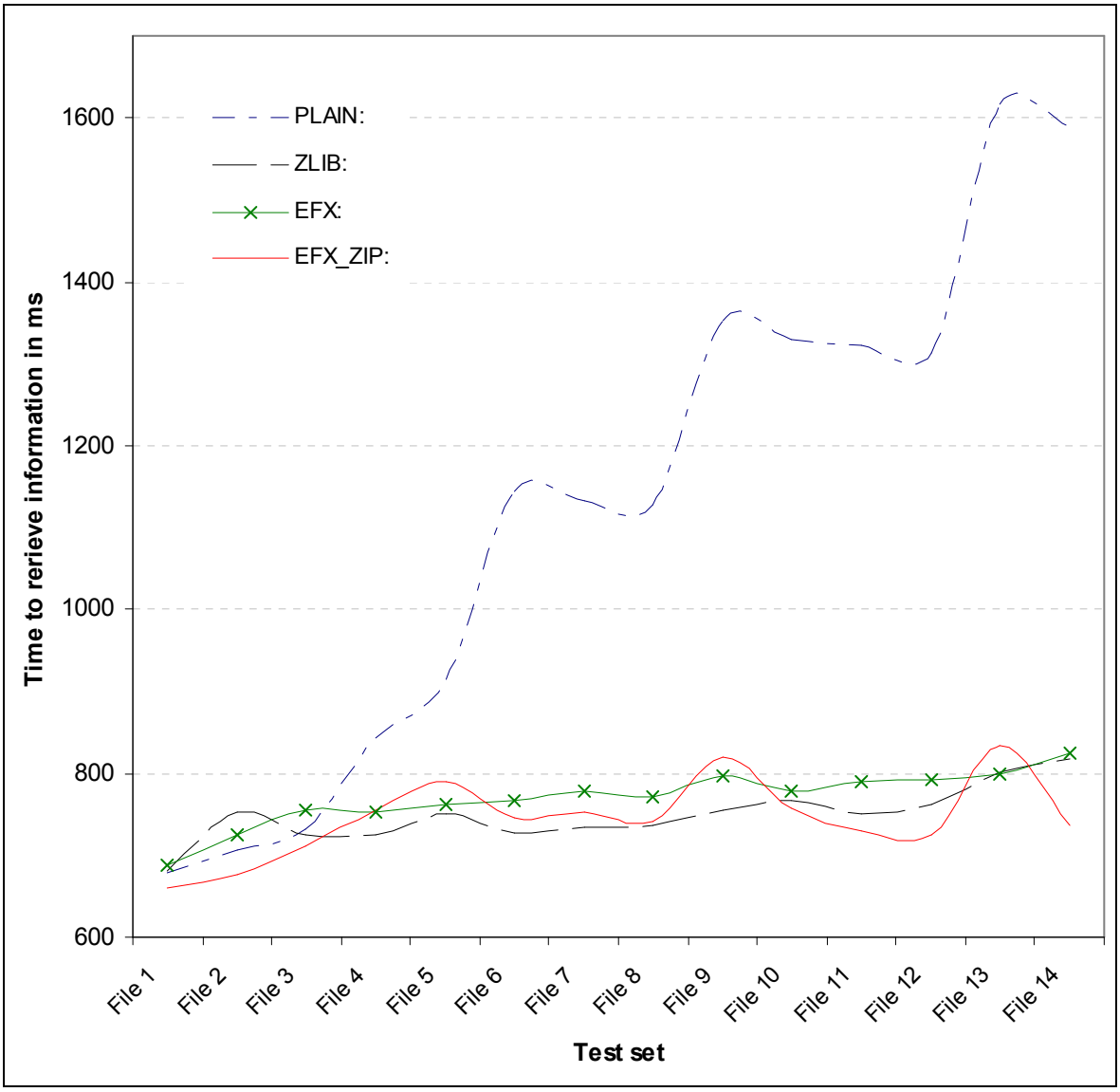

Figure 6.8 Results of UTMS simulation.

The results for the real UMTS network were obtained by using an UMTS card in a laptop. For each format and data set, 200 measurements were performed. The averages of these measurements are presented in Figure 6.9.

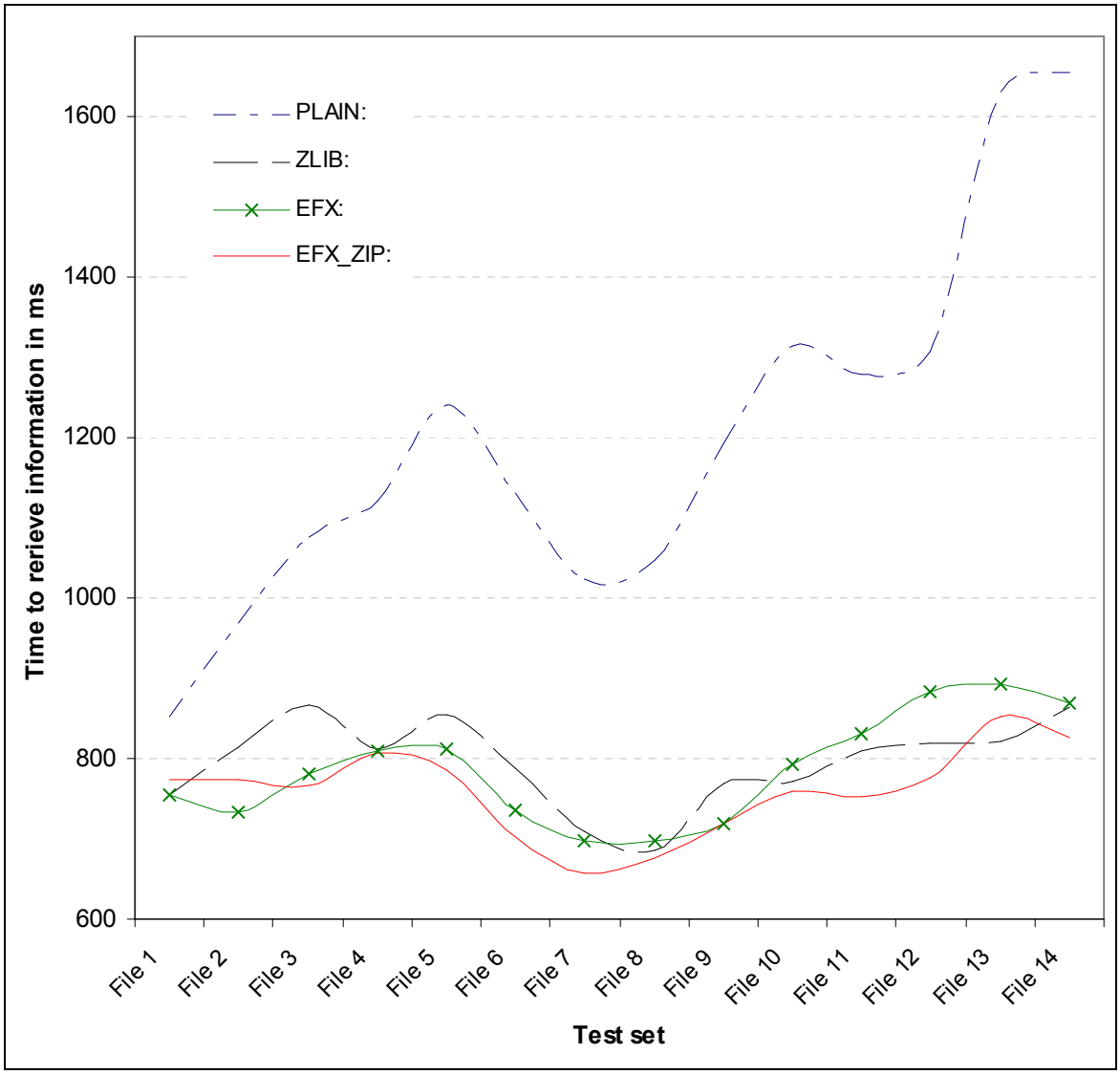

Figure 6.9 UMTS times from measurements in real networks.

As can be seen in both Figure 6.8 and 6.9 above, the PLAIN XML files are the only ones that separate themselves. With the largest test data, *File 14*, the time to transfer the data in the PLAIN format was twice the time of all the three other formats studied. For the three other formats the main part of the transfer time seems to be related to the latency.

6.5.2 EDGE results

Regarding the EDGE measurements, there did not exist any way to lock the data card in the laptop to the EDGE service, so the results presented here are based on the simulated measurements. The parameters used for the simulation were:

- upload: 50 Kbps
- download: 200 Kbps

- one-way delay: 200 ms

The results of this simulation, as an average of 6 measurements of each format, are shown in Figure 6.10.

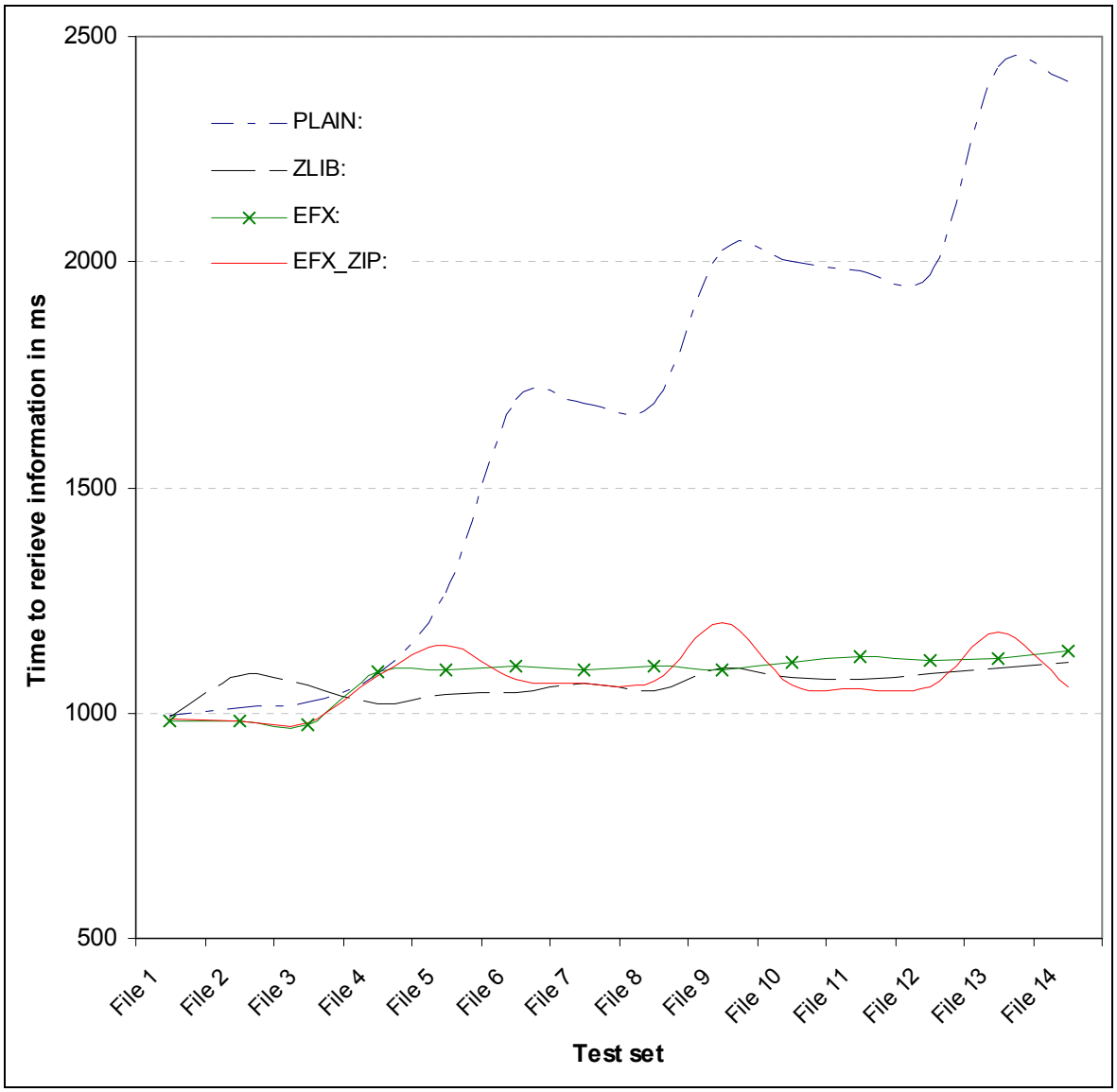

Figure 6.10 Results for the EDGE simulation.

The first thing to take notice of in Figure 6.10, is that the time for the smallest test data, *File 1*, has increased with about 200 ms compared to the UMTS results. The main part of this can be expected to be a result of the added time related to the one-way delay, which increased from 120 ms to 200 ms for EDGE. This alone led to an 80 ms increase in the time for each message transferred. For the biggest file in the test data, *File 14*, the picture is a bit different. For the PLAIN format, the time has increased with 750 ms compared to UMTS, almost all of this is expected to be a result of the reduced bandwidth. For the ZLIB, EFX and EFX_ZIP formats, the file sizes are kept small for all the data in the test sets, and hence the major difference between transfer in UMTS and EDGE networks is related to the delay.

6.5.3 GPRS results

The last RTT that was looked into was GPRS. As for UMTS, both simulated and real network measurements were performed. The parameters used for the simulation were:

- upload: 30 Kbps,
- download: 50 Kbps
- one-way delay: 350 ms.

The measurements in the real network were performed on a laptop with a mobile data card locked to the GPRS network. The results of the simulation are based on the average of 6 measurements for each file in each format, and are presented in Figure 6.11.

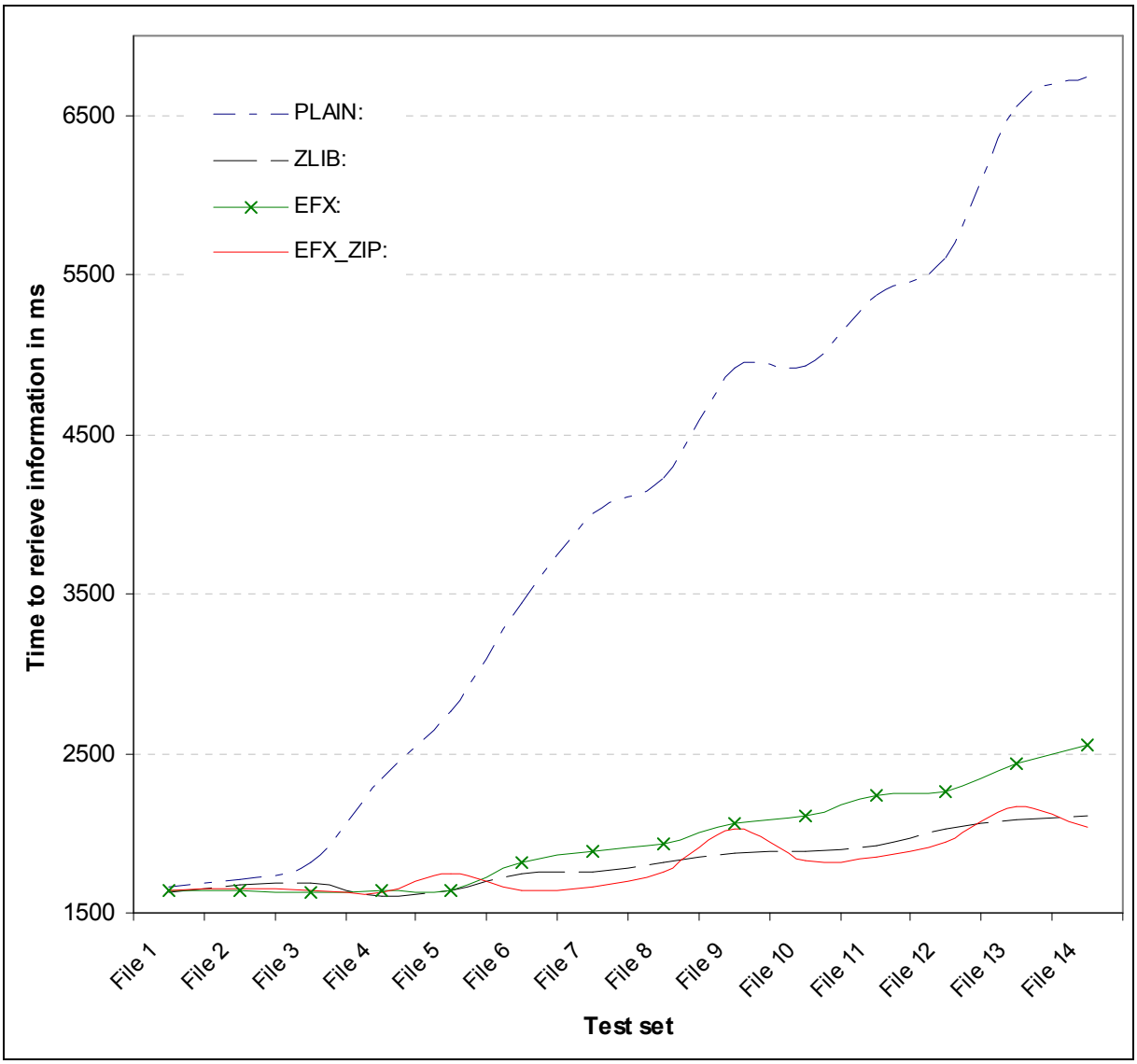

Figure 6.11 Results from the GPRS simulation.

In the real GPRS network 30 measurements were performed for each file in each format. The average of these measurements is presented in Figure 6.12, and as can be seen, the results from the simulation and the real network did not give the same results.

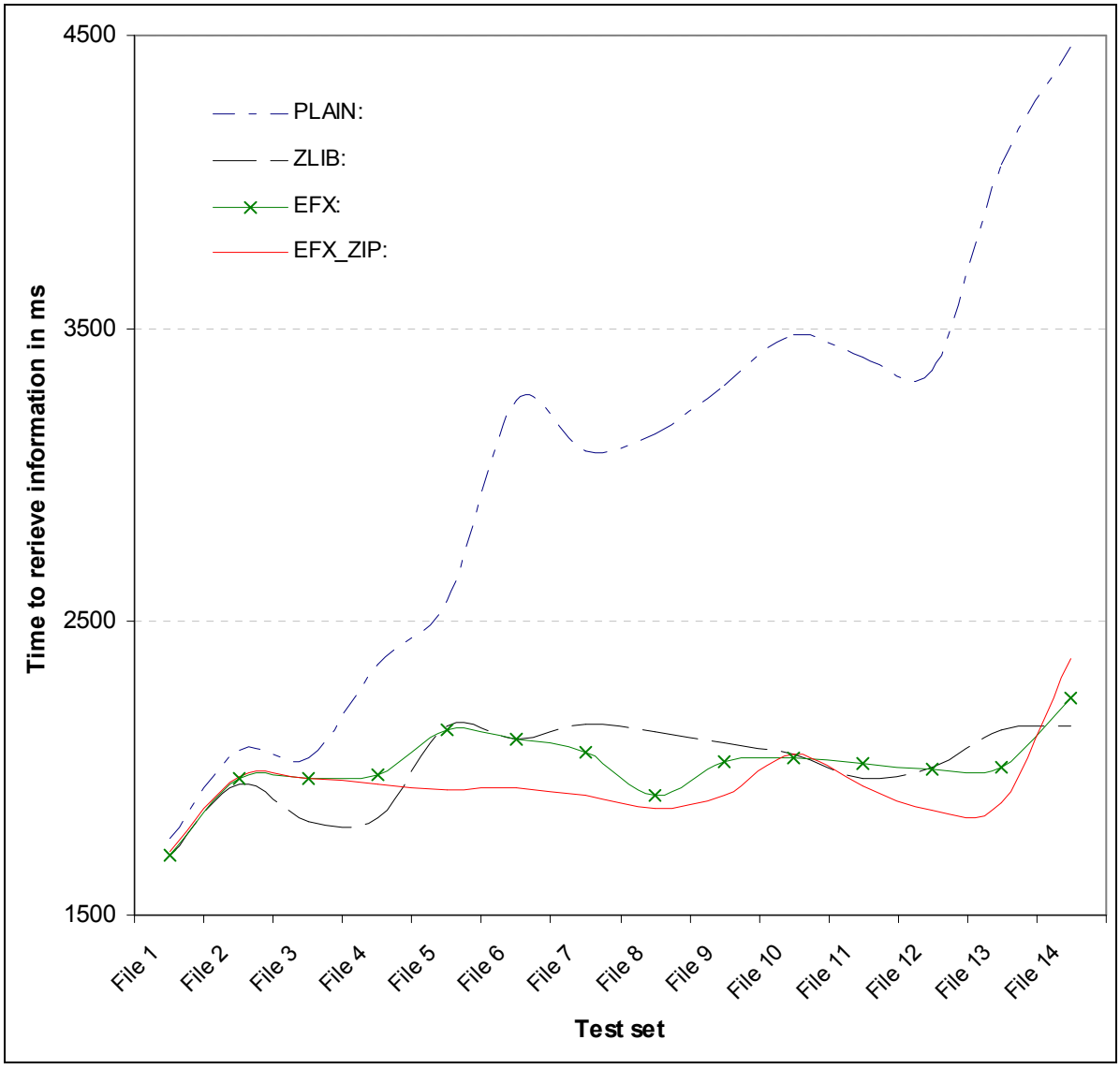

Figure 6.12 Average times of the real GPRS network.

When comparing the results in Figure 6.11 and Figure 6.12 one can notice only is a slight difference in the elapsed time for the smallest test data, which indicates that the delay is correct. However, for the largest transfer, the simulated result takes more than 2000 ms more to finish. This tells us that the download bandwidth for these measurements is higher than those used in the simulation. So a user can expect a download bandwidth that is higher than 50 Kbps.

A closer look at the results obtained from the real GPRS network paints much of the same picture as for UMTS and EDGE. Initially, the largest part of the time is related to the delay, which in this case adds up to more than 1500 ms. For all the three formats that are alternative

representation of the original xml files, the delay is the major part of the RTT. The RTT increases slightly as the size of the test data increases, but not nearly as much as for the original xml files. For the largest file in the test data, *File 14*, the PLAIN format takes more than 2000 ms more, or about twice the time to transfer than the three other formats.

6.6 Final Words on the Results

Throughout chapter 6.3 to 6.5 the executed measurements related to representing XML in other formats have been presented. First, the benefits in term of reducing file sizes were shown. It was demonstrated that both compression, ZLIB, and Binary XML, EFX, give considerably smaller files. However, ZLIB needs a certain amount of data to perform as well as EFX. If compression is applied to the EFX format, a format that produces the smallest files for all the data in the test set is achieved.

Two facts that indicate that EFX should be the preferred format over ZLIB are the limited processing power and memory on mobile devices. ZLIB is an intermediate format that needs decompression before it can be further processed, thereby leading to processing overhead that consumes time and battery lifetime. In contrast, EFX can be processed immediately. The other problem with ZLIB is that when it is decompressed, the resulting data has the same memory footprint as the original XML, whereas EFX produces a much smaller memory footprint. The latter is the preferred situation on mobile devices with limited memory.

In relation to the cost of Web Services use, ZLIB, EFX and EFX_ZIP all give a significant saving over the PLAIN format. EFX_ZIP will reduce the cost with at least a factor of 7 for all the data in the test set used here. It should be noted that the ZLIB format does not perform well when it is applied on small data. For this reason it should be avoided in conjunction to cost when the Web Service client and server mainly exchange small amount of data.

The impact the different formats had on the RTT, or response times, were also investigated. The first thing that was noted, was that for the small test files all the formats took almost the same time to retrieve the information. This was confirmed both by measurements in real UMTS and GPRS networks and in simulated UMTS, EDGE and GPRS networks. This tells us that when the data that are transferred by a Web Service mainly consists of small messages, the possible benefit, in term of reduced response time, is limited in conjunction with representing the data in other formats to reduce the size of the messages. To get a better response time in these situations, stack optimization may be the solution. However, when the data in the test files got larger, the response time was significantly reduced when representing the data in ZLIB, EFX or EFX_ZIP. For the largest file in the test set, the response time for all these formats were about half that of the PLAIN format for all the networks. Both the simulated and real networks confirmed this. So the possible benefits of more compact representation of the information increases as the amount of data transferred increases.

To achieve better response time, both for small and large files, the possible optimization to the stack proposed in chapter 3.5 should be investigated further. If for example the number of messages exchanged between the server and client could be reduced from 4 to 2, it would probably lead to a reduction to almost half the response time. However, further study and measurements are needed to say anything certain about this. The processing times needed for

the different formats should also be studied in detail to give a good foundation for choice of the most appropriate format for different situations.

7 Conclusion

Mobile devices which were initially meant to be peripheral voice call devices have now become small computers that are expected to do almost anything anywhere. In connection with this, the desire to employ Web Services from mobile devices has increased considerably. However, there are issues that need to be addressed in order to obtain satisfactory services. Although there has been an introduction of high-speed mobile packet-switched networks like EDGE and UMTS, the large information overhead of XML and the protocol stack of Web Services may cause an unacceptable response time. The reason for this is the limited bandwidth and high latency of wireless links. In addition, the usage of these networks is usually charged based on the amount of data transferred, which may lead to high cost of use for Web Services. This thesis identifies two main approaches to overcome these challenges; reducing the size of XML information and optimizing the communication stack.

Several techniques to reduce the size of the XML exchanged between the Web Service and the mobile device have also been examined. Three methods were applied on the original XML information to reduce its size; ZLIB compression, EFX binary XML, and EFX_ZIP, the latter being a combination of binary XML and compression. The effect this had was studied in a phonebook prototype Web Service developed for measurements use. ZLIB, EFX and EFX_ZIP all produced files that were smaller than 25 % of the original files, with an exception for the combination of small files and ZLIB. The immediate effect of a reduction of data size is reduced costs of transfer. Data transfer on mobile devices is usually charged per byte, and therefore less data sent and received leads to reduced costs. Furthermore, it was identified that the mobile device used in this study, had limitations regarding how large files it could handle. By utilizing one of the smaller formats proposed, much more information can be transferred before this limit is reached.

The measurements of the response time showed that ZLIB, EFX and EFX_ZIP had almost the same response time. However, these formats led to about half the response time of the PLAIN format for the largest files in the test. This shows that when the files are large, the limited bandwidth contributes to a significant part of the response time and implies that one of the smaller formats should be used. This difference between the formats decreased as the file size decreased, and for the smallest files, the PLAIN and the other three formats had almost the same response time. Both the simulated and the real network measurements confirmed these results for UMTS, EDGE and GPRS networks. Since the response time is about equal for all the formats for small files, this indicates that the main part of the response time is caused by the latency in this situation. To reduce the response time more, protocol stack optimization seems to be the solution. Since SOAP is independent of underlying protocols it should be possible to remove unnecessary protocols from the stack; HTTP might be one protocol that can be removed. In addition, TCP does not perform well over wireless links, so it should either be optimized for such links or be replaced by a protocol that better meets the challenges of such wireless links.

Concerning the cost of using Web Services from mobile devices, all the formats, ZLIB, EFX and EFX_ZIP reduce the cost as well as the response time compared to the PLAIN format. However, this thesis cannot conclude that one of the formats is better than the others. To draw such a conclusion more investigation of the processing times, memory footprints etc. have to be studied. The EFX format does, however, seem to have great potential since it is not an

intermediate format such as ZLIB. Thereby it can be processed directly, it is kept small and does not increase in size like the ZLIB format does after decompression.

Reducing the size of XML is concluded to be a good solution both for reducing the cost and the response time when accessing Web Services from mobile devices, however, stack optimization should be investigated to clarify its potential to further increase the performance.

7.1 Future Work

This report illustrates the possibly huge benefits of reducing the size of XML information both in terms of response time and costs. There is, however, a need to study other characteristics of the formats dealt with in this thesis. Processing times for all the formats are one area that is very important in conjunction with their usability on mobile devices. The proposed optimization of the stack should also be studied. Especially optimizing TCP for wireless links, or even change it with another protocol, seems to have great potential for reducing the response time of Web Services. This should be implemented, and measurements should be performed to identify possible benefits. It was also identified that the parser used for Efficient XML did not handle the Norwegian letters æ, æ and å, and this should also be investigated.

Bibliography

[1] Telenor. “Priser for bruk av GPRS, EDGE og UMTS i utlandet”. Norway: Telenor. http://telenormobil.no/priser/iutlandet/gprs/

[2] Kangasharju, J., Tarkoma, S., Raatikainen, K. “Comparing SOAP Performance for Various Encodings, Protocols, and Connections“. Finland: Helsinki Institute for Information Technology. (File included)

[3] Lai, K. Y., Phan, A. K. T., Tari, Z. “Efficient SOAP binding for mobile Web services”. Australia: IEEE, 2005.

[4] White, G., Kangasharju, J., Brutzman, D., Williams, S. “Efficient XML Interchange Measurements Note”. W3C: July 2007. http://www.w3.org/TR/exi-measurements/

[5] IBM. “Extend the value of your core business systems”. USA: IBM, September 2006. (File included)

[6] Nagappan, R., Skoczylas, R., Sriganesh, R. P. “Developing Java Web Services”. USA: Wiley Publishing Inc., 2003.

[7] Thanh, D. V. “Guest editorial - Future mobile phones”. Norway: Telektronikk, vol. 101 No. 3/4, page 1 - 2, 2005.

[8] Jerz, M. “Nokia N95 review”. My-Symbian.com., February 2006. http://my-symbian.com/s60v3/review_n95.php, visited 10.08.07.

[9] Wikipedia. “Smartphone”. Wikipedia., 2007. http://en.wikipedia.org/wiki/Smartphone, visited 05.06.07.

[10] Sun Microsystems. “Connected Limited Device Configuration”. USA: Sun Microsystems., March 2003. (File included)

[11] Open Mobile Terminal Platform. “OMTP Java with focus on CDC: Definition and Requirements”. OMYP Ltd. January 2006. (File included)

[12] Howorth, R. “Sun details open-source plan for J2ME”. IT Week, August 2006. http://www.itweek.co.uk/itweek/news/2162369/sun-details-open-source-plan, visited 01.06.07

[13] Sun Microsystems. “CLDC Hot Spot Implementation Virtual Machine”. USA: Sun Microsystems, February 2005. (File included)

[14] Sun Microsystems. “Mobile Information Device Profile”. USA: Sun Microsystems, April 2006. (File included)

[15] Oritz, C. E. “Understanding the Web Services Subset API for Java ME”. Sun Microsystems, March 2006.

[16] Valmot, O. R. “Mobilt internet med fart”. Norway: Teknisk Ukeblad, vol. 16, page 26 - 27, 2007.

[17] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F. “Extensible Markup Language (XML) 1.0 (Fourth Edition)”. W3C, September 2006.

[18] Wikipedia. “PKZIP”. Wikipedia, 2007. http://en.wikipedia.org/wiki/PKZIP, visited 18.04.07.

[19] Deutsch, P. “RFC 1951: DEFLATE Compressed Data Format Spesification version 1.3”. IETF, May 1996.

[20] Ziv, J., Lempel, A. “ A Universal Algorithm for Sequential Data Compression”. IEEE transactions on information theory, vol. IT-23, No. 3, May 1977.

[21] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. “ Introduction to Algorithms”. USA: The MIT Press, 2001.

[22] Adler, M. “ZLIB Home Site”. ZLIB, 2007. http://www.zlib.net/, visited 19.04.07.

[23] Deutsch, P. “RFC 1952: GZIP file format specification version 4.3”. IETF, May 1996. http://tools.ietf.org/html/rfc1952.

[24] Deutsch, P., Gailly, J-L. “RFC 1950: ZLIB Compressed Data Format Spesification version 3.3”. IETF, May 1996. http://tools.ietf.org/html/rfc1950.

[25] Suciu, D., Liefke, H. “XMill: An Efficient Compressor for XML”. XMill, 2004. http://www.liefke.com/hartmut/xmill/xmill.html, visited 23.04.07.

[26] Cokus, M., Pericas-Geertsen, S. “XML Binary Characterization Use Cases”. W3C, March 2005. http://www.w3.org/TR/xbc-use-cases/.

[27] W3C. “XML Binary Characterization Working Group Public Page”. W3C, 2005. http://www.w3.org/XML/Binary/.

[28] Goldman, O., Lenkov, D. “XML Binary Characterization”. W3C, March 2005. http://www.w3.org/TR/xbc-characterization/.

[29] W3C. “Efficient XML Interchange Working Group”. W3C, 2007. http://www.w3.org/XML/EXI/.

[30] Dawson, F., Howes, T. “RFC 2426: vCard MIME Directory Profile”. IETF, September 1998. http://www.ietf.org/rfc/rfc2426.txt.

[31] AgileDelta. “AgileDelta Home Site”. AgileDelta, 2007. http://www.agiledelta.com/, visited 08.06.07.

[32] ITU-T Study Group 17. “ITU-T rec. X.891 Information technology – Generic applications of ASN.1: Fast infoset”. ITU-T, May 2005. (File included)

[33] Schneider, J., Kamiya, T. “Efficient XML Interchange (EXI) Format 1.0”. W3C, July 2007. http://www.w3.org/TR/exi/.

[34] AgileDelta. “Performance and Features Site”. AgileDelta, 2007. http://www.agiledelta.com/efx_perffeatures.html, visited 03.07.07.

[35] Audestad, J. A. “Protocol Design: Principles and Methods”. Norway: Tapir Akademiske Forlag, 2004.

[36] ITU-T Study Group 17. “ITU-T rec. X.693 Information technology – ASN.1 enccoding rules: XML Encoding Rules (XER)”. ITU-T, December 2001. (File included)

[37] ITU-T Study Group 17. “ITU-T rec. X.694 Information technology – ASN1 encoding rules: Mapping W3C XML schema definitions into ASN.1”. ITU-T, January 2004. (File included)

[38] Sandoz, P., Pericas-Geertsen, S., Kawaguchi, K., Hadley, M., Pelegeri-Llopart, E. “Fast Web Services”. Sun Developer Network, August 2003. http://java.sun.com/developer/technicalArticles/WebServices/fastWS/index.html, visited 09.06.07.

[39] Sandoz, P. “Fast Infoset and the Pragmatic SOA Approach”. Sun Developer Network, October 2005. http://java.sun.com/developer/technicalArticles/WebServices/soa2/fastinfoset-soa.htm, visited 09.06.07.

[40] Mitra, N., Lafon, Y. “SOAP Version 1.2 Part 0: Primer (Second Edition)”. W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[41] Hirsch, F., Kemp, J., Ilka, J. “Mobile Web Services, Architecture and Implementation”. England: John Wiley & Sons, Ltd., 2006.

[42] Tanenbaum, A. S. “Computer Networks”. USA: Prentice Hall PTR, 1996.

[43] Thanh, D. V., Jørstad, I., Thuan, D. V. “Fetching home music - Sending photos home” Norway: Telektronikk, vol. 101 No. 3/4, page 123 - 130, 2005.

[44] Sun Microsystems. “Java 2 Platform, Micro Edition (J2ME) Web Services”. USA: Sun Microsystems, July 2004.

[45] Eclipse. “Eclipse – an open development platform Home Site”. Eclipse, 2007. http://www.eclipse.org/, visited 13.03.07.

[46] Eclipse. “Web Tools Platform (WTP) Project Site”. Eclipse, 2007. http://www.eclipse.org/webtools/main.php, visited 15.03.07.

[47] Sun Developer Network. “Java SE at a Glance”. Sun Developer Network, 2007. http://java.sun.com/javase/, visited 04.03.07.

[48] The Apache Software Foundation. “The Apache Software Foundation Home Site”. The Apache Software Foundation, 2007. http://www.apache.org/, visited 28.03.07.

[49] The Apache Software Foundation. “ Apache Axis2/Java Site”. The Apache Software Foundation, 2007. http://ws.apache.org/axis2/index.html, visited 01.04.07.

[50] Sony Ericsson Mobile Communications AB. “K610 White Paper”. Sony Ericsson Mobile Communications, July 2006. (File included)

[51] Source-Code.biz. “Base64 encoder/decoder in Java”. Source-Code.biz, 2007. http://www.source-code.biz/snippets/java/2.htm, visited 02.05.07.

[52] JCraft. “JZlib – zlib in pure Java”. JCraft, 2005. http://www.jcraft.com/jzlib/, visited 11.05.07.

[53] Docjar. “java.io Class FilterInputStream”. Docjar, 2007 http://www.docjar.com/docs/api/java/io/FilterInputStream.html, visited 12.05.07.

[54] Telefonkatalogen. “Finn telefonnummer i Norges største telefonkatalog”. Gulesider.no, 2007. http://www.gulesider.no/tk/index.c, visited 17.06.07.

[55] AgileDelta. “Getting Started with Efficient XML 2.0”. AgileDelta, 2006. (File included)

[56] Sun Developer Network forum. “Error when invoking WebService from WTK 2.5 client”. Sun Developer Network, December 2006. http://forum.java.sun.com/thread.jspa?threadID=793461&tstart=135, visited 16.04.07.

[57] Efficient XML Support. “Email from: efx@agiledelta.com regarding EFXProperty, ZIP, in J2ME”. AgileDelta, May 2007.

[58] Carson, M., Darrin, S. “NIST Net – A Linux-based Network Emulation Tool”. National Institute of Standards and Technology. (File included)

The files marked *File included* can be found in the directory *literature/* in the ZIP file *Lars_Johnsrud_master_files.zip*, which follows this report.

Appendix A WSDL File for the Phonebook Web Service

This is the WSDL file that forms the basis for the Phonebook Web Service described in chapter 4 and 5.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns2="http://johnsrud.no/AddressBook1/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="AddressBook1"
targetNamespace="http://johnsrud.no/AddressBook1/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://johnsrud.no/AddressBook1/">


      <xsd:element name="getNumberPlainAsString">
        <xsd:complexType>
                <xsd:sequence>
                                <xsd:element name="name" type="xsd:string"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getNumberPlainAsStringResponse">
        <xsd:complexType>
                <xsd:sequence>
                                <xsd:element name="numberPlainAsString"
                                type="xsd:string"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getNumberZlibAsString">
        <xsd:complexType>
                <xsd:sequence>
                                <xsd:element name="name" type="xsd:string"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getNumberZlibAsStringResponse">
        <xsd:complexType>
                <xsd:sequence>
                                <xsd:element name="numberZlibAsString"
                                type="xsd:string"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getNumberEfxAsString">
        <xsd:complexType>
                <xsd:sequence>
                                <xsd:element name="name" type="xsd:string"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getNumberEfxAsStringResponse">
        <xsd:complexType>
                <xsd:sequence>
                                <xsd:element name="numberEfxAsString"
                                type="xsd:string"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getNumberEfxAndZlibAsString">
        <xsd:complexType>
                <xsd:sequence>
                                <xsd:element name="in" type="xsd:string"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getNumberEfxAndZlibAsStringResponse">
        <xsd:complexType>
                <xsd:sequence>
                                <xsd:element name="out" type="xsd:string"></xsd:element>
```

```
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="getNumberNotInUse">
            <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="in" type="xsd:string"></xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="getNumberNotInUseResponse">
            <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="out" type="xsd:string"></xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
</xsd:schema>
</wsdl:types>

  <wsdl:message name="getNumberPlainAsStringRequest">
        <wsdl:part name="parameters" element="tns2:getNumberPlainAsString"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberPlainAsStringResponse">
        <wsdl:part name="parameters" element="tns2:getNumberPlainAsStringResponse"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberZlibAsStringRequest">
        <wsdl:part name="parameters" element="tns2:getNumberZlibAsString"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberZlibAsStringResponse">
        <wsdl:part name="parameters" element="tns2:getNumberZlibAsStringResponse"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberEfxAsStringRequest">
        <wsdl:part name="parameters" element="tns2:getNumberEfxAsString"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberEfxAsStringResponse">
        <wsdl:part name="parameters" element="tns2:getNumberEfxAsStringResponse"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberEfxAndZlibAsStringRequest">
        <wsdl:part name="parameters" element="tns2:getNumberEfxAndZlibAsString"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberEfxAndZlibAsStringResponse">
        <wsdl:part name="parameters"
        element="tns2:getNumberEfxAndZlibAsStringResponse"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberNotInUseRequest">
        <wsdl:part name="parameters" element="tns2:getNumberNotInUse"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="getNumberNotInUseResponse">
        <wsdl:part name="parameters" element="tns2:getNumberNotInUseResponse"></wsdl:part>
  </wsdl:message>

  <wsdl:portType name="AddressBook1">
    <wsdl:operation name="getNumberPlainAsString">
        <wsdl:input message="tns2:getNumberPlainAsStringRequest"></wsdl:input>
        <wsdl:output message="tns2:getNumberPlainAsStringResponse"></wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getNumberZlibAsString">
        <wsdl:input message="tns2:getNumberZlibAsStringRequest"></wsdl:input>
                <wsdl:output message="tns2:getNumberZlibAsStringResponse"></wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getNumberEfxAsString">
        <wsdl:input message="tns2:getNumberEfxAsStringRequest"></wsdl:input>
                <wsdl:output
                message="tns2:getNumberEfxAsStringResponse"></wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getNumberEfxAndZlibAsString">
        <wsdl:input message="tns2:getNumberEfxAndZlibAsStringRequest"></wsdl:input>
        <wsdl:output message="tns2:getNumberEfxAndZlibAsStringResponse"></wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getNumberNotInUse">
        <wsdl:input message="tns2:getNumberNotInUseRequest"></wsdl:input>
        <wsdl:output message="tns2:getNumberNotInUseResponse"></wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
```

```
  <wsdl:binding name="AddressBook1SOAP" type="tns2:AddressBook1">
        <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="getNumberPlainAsString">
                <soap:operation
                soapAction="http://johnsrud.no/AddressBook1/getNumberPlainAsString" />
                <wsdl:input>
                        <soap:body use="literal" />
                </wsdl:input>
                <wsdl:output>
                        <soap:body use="literal" />
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="getNumberZlibAsString">
                <soap:operation
                        soapAction="http://johnsrud.no/AddressBook1/getNumberZlibAsString" />
                <wsdl:input>
                        <soap:body use="literal" />
                </wsdl:input>
                <wsdl:output>
                        <soap:body use="literal" />
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="getNumberEfxAsString">
                <soap:operation
                        soapAction="http://johnsrud.no/AddressBook1/getNumberEfxAsString" />
                <wsdl:input>
                        <soap:body use="literal" />
                </wsdl:input>
                <wsdl:output>
                        <soap:body use="literal" />
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="getNumberEfxAndZlibAsString">
                <soap:operation
                        soapAction="http://johnsrud.no/AddressBook1/getNumberEfxAndZlibAsString"
/>
                <wsdl:input>
                        <soap:body use="literal" />
                </wsdl:input>
                <wsdl:output>
                        <soap:body use="literal" />
                </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="getNumberNotInUse">
                <soap:operation
                        soapAction="http://johnsrud.no/AddressBook1/getNumberNotInUse" />
                <wsdl:input>
                        <soap:body use="literal" />
                </wsdl:input>
                <wsdl:output>
                        <soap:body use="literal" />
                </wsdl:output>
        </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="AddressBook1">
    <wsdl:port binding="tns2:AddressBook1SOAP" name="AddressBook1SOAP">
      <soap:address location="http://82.147.56.98:8080/axis2/services/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Appendix B Source Code for the Phonebook Web Service

The source code related to this thesis can be found in the ZIP file *Lars_Johnsrud_master_files.zip*, which follows this report. All the development has been performed with the Eclipse development tool.

Server: The source code for the server can be found in the directory: *development/AddressBook3*.

Client: The source code for the client can be found in the directory: *development/AddressBook3Client*.

Appendix C Test Data Used for Measurements

The test data used in the measurements performed in this thesis can be found in the ZIP file *Lars_Johnsrud_master_files.zip*, which follows this report.

The files shown in Table C.1 can be found in the directory *data/*. Each file is in the PLAIN, EFX and EFX_ZIP format in this directory. The files have file extensions corresponding to their format.

Table C.1 Test data used in the measurements

Test set	Name	PLAIN Size in bytes	ZLIB Size in bytes	EFX Size in bytes	EFX_ZIP Size in bytes
File 1	**Fredrik.xml**	507	303	69	69
File 2	**Anne.xml**	1 556	485	295	246
File 3	**Lars.xml**	2 081	595	427	346
File 4	**Elin.xml**	4 471	972	979	691
File 5	**Jostein.xml**	6 347	1 183	1 329	908
File 6	**Susanne.xml**	8 769	1 567	1 902	1 252
File 7	**Harald.xml**	11 883	1 892	2 485	1 580
File 8	**Kristian.xml**	12 913	2 063	2 690	1 719
File 9	**Trine.xml**	15 291	2 425	3 303	2 046
File 10	**Marianne.xml**	16 280	2 482	3 558	2 060
File 11	**Mathias.xml**	18 134	2 591	3 805	2 264
File 12	**Ida.xml**	19 125	3 033	4 209	2 596
File 13	**Per.xml**	23 608	3 301	4 903	2 874
File 14	**Ole.xml**	24 309	3 584	5 267	3 127