NTNU
Norwegian University of
Science and Technology

Web Service Clients on Mobile Android Devices

A Study on Architectural Alternatives and Client Performance

Johannes Knutsen

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Dag Svanæs, IDI
Co-supervisor: Erlend Stav, Sintef

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

In the ongoing EU IST project MPOWER, an open platform to simplify and speed up the task of developing and deploying services for persons with cognitive disabilities and elderly have been developed. This master assignment will investigate architectural choices and realization mechanisms for developing web service based applications for the Android platform, with the MPOWER services as the main case. By developing one or more proof-of-concept application using MPOWER services from the handheld device, it will be evaluated how different architectures affects the properties of the client and whether direct invocation of SOAP based Web services from Android is a viable approach.

MPOWER - Middleware platform for empowering cognitive disabled and elderly - is an ongoing IST research project. MPOWER has defined and implemented an open platform to simplify and speed up the task of developing and deploying services for persons with cognitive disabilities and elderly. The services of the platform cover two focus areas:
- A collaborative environment for distributed and shared care, providing requirements for information security, information models, context awareness, usability and interoperability.
- A SMART HOUSE environment, providing requirements for information security, information models and usability.

The MPOWER middleware services been released for free and are available in open source from: http://sourceforge.net/projects/free-mpower/.

Assignment given: 19. January 2009
Supervisor: Dag Svanæs, IDI

Abstract

This paper studies Android, a new open source software stack initiated by Google, and the possibilities of developing a mobile client for MPower, a service oriented architecture platform based upon SOAP messaging.

The study focuses on the architectural alternatives, their impacts on the mobile client application, Android's performance on SOAP messaging, and how Web services' design can be optimized to give well performing Android clients.

The results from this study shows how different architectures directly impacts properties, like off-line usage support, of a SOAP client application on Android. Additionally, the performance measurements shows that building Android client applications which directly invokes Web services with SOAP messaging is possible to make effective enough for typical usage situations. Further, the results indicates how Web services should be designed with care to minimize the required data transfer and processing on the device. Such careful design can be achieved by using coordinating Web services which hides complexity and provides an interface designed for the specific client applications.

Preface

This report is a documentation of the work carried out throughout the spring of 2009 by Johannes Knutsen in TDT4900, Computer and Information Science, Master Thesis. The work is performed in the tenth, and last, semester of the Master of Technology education in computer science in The Norwegian University of Technology and Science, NTNU.

The project was defined by co-supervisor Erlend Stav at Sintef, and is supervised by Dag Svanæs at the Department of Computer and Information Science. I would like to thank both supervisors for their feedback, guidance, encouragements, and an interesting topic to work on during my master.

Trondheim, June 2009

Johannes Knutsen

Glossary

Dalvik The *Android* virtual machine. One *Dalvik* instance runs for each running *Android* application.

EDGE (Enhanced Data rates for GSM Evolution) Data transmission rate extension to GSM networks.

GPRS (General Packet Radio Service) Packet switch mobile service of the 2G GSM service.

HSPA (High Speed Packet Access) Higher speed transmission protocol, typically deployed on UMTS mobile services.

IDE Integrated Development Environment.

Mobile client In the context of this thesis, a mobile client is a application running on a mobile platform like Android, which accesses SOAP based Web services.

SOA Service oriented architecture. An architecture where functionality is divided into loosely coupled communicating services.

SOAP A protocol specification which describes the representation of messages in an XML format and how they can be transported over application protocols like HTTP or SMTP.

Third party code library Ready to use code which solves a specific task developed by a party external to the official platform API. For example a third party XML library, helps developers to parse XML data.

UMTS (Universal Mobile Telecommunications System) Third generation mobile services (3G).

XML Extensible Markup Language, is a structured, hierarchical, text-based data format widely used as a data exchange format[17].

Contents

List of Figures

List of Tables

[1]Percentile is here defined as the request duration below which a certain percent of measures fall. For example do 60% of the measurements last 98ms or less.

Chapter 1

Introduction

Smart mobile devices have become increasingly popular in the recent years. Together with the popularity, a range of system platforms and application programming environments have been created for the phones. This includes Symbian, PalmOs, J2ME, Blackberry, Windows Mobile and iPhone.

In this context, *Google* together with the *Open Handset Alliance*[1] recently released an open source mobile software stack named *Android*. Android supports a subset of the Java API, uses Java as it's programming language, has broad customization support, has built-in graphical user interface components and comes with a set of core applications accessible by third-party developers. Google's heavy effort in *Android* has resulted in a range of mobile device manufacturers to announce their support and commitment to the *Open Handset Alliance*. This includes Motorola, HTC, Samsung, LG, Sony Ericsson and many more.

A range of service platforms have recently been built based on a Service Oriented Architecture (SOA) with the *SOAP* messaging protocol. By creating mobile clients for these application platforms, an increased support for in-field usage of the systems is possible.

MPOWER[12] is one such service platform which aims to support rapid development and deployment of services for cognitive disabled and elderly[11]. MPOWER has been seen as a success from the end-users perspective and Android is an upcoming mobile platform, which makes them particular interesting in light of the research questions. Hence they are both chosen as research platforms for this thesis.

This thesis evaluates which architectural choices must be made and how such choices impacts client applications on mobile devices running Android. Further, a study of how directly invoking Web services and the performance of such invocations, is performed on Android. It will also analyze to what degree existing Web services must be redesigned to fit such

[1] `http://www.openhandsetalliance.com`

mobile access.

1.1 Research questions

The main topic of this thesis is a study on how mobile devices running Android can be incorporated in service oriented architectures and the following research questions will be answered:

1. Which architectural alternatives exists for using SOAP based Web services on Android, and how do the architectural choice affect the client application?
2. Is it possible to directly invoke SOAP Web services on Android, and will such invocation be effective enough?
3. How can the design of SOAP Web services be optimized for use on mobile devices running Android?

A mobile client is in this thesis' context defined as a Web service client application which runs on a mobile handheld device, like a device running Android. Further, are these Web services implemented on a SOAP based service oriented architecture.

The meaning of architectural alternatives is in this context the composition of nodes, like servers and clients, between the service oriented architecture endpoint to the client, including the transmission protocols and communication paths. The internal composition alternatives of the client application and service platform is thus not evaluated.

1.2 Research method

The research method used, is based upon a design science[10] research methodology. Design science is performed by implementing an artifact, testing it and using the gathered knowledge to further develop the artifact.

The artifact development used the services available on the MPOWER service oriented platform platform, described in section 2.7.

1.3 Research design

In order to answer the research questions, the following research activities was performed.

A search for architectural alternatives In order to find alternative architectures, a search for architectures described and used on other platforms than Android is performed.

Analyse architectural alternatives The analysis should evaluate the architectural alternatives found for using SOAP Web services on Android and identify how the architectural choice affects the client application. The analysis is based upon quality attributes as defined by Bass, Clements and Kazman[3].

Implement a simple Android SOAP client Create a simple SOAP client which makes a simple SOAP request and parses the SOAP response.

The goal is to identify a suitable approach for performing SOAP invocations directly from Android and through basic benchmarks identify potential bottlenecks and problems. Additionally, this will be the first use of the Android SDK and the research action should provide a better understanding of the new platform.

Implement Android SOAP client against a MPOWER service Access the *MessageBoard* service on MPOWER and create a mobile client which is able to send a message through the message board to another user. The client should call the required Web Services directly.

This will be a proof of concept application on the Android, showing how the direct invocation architecture works on Android. The goal is to use it for further benchmarking of direct SOAP invocation against the MPower platform. Additionally, it should prove that direct SOAP invocation is possible on Android.

Measure performance on expected MPower MessageBoard user operations A set of defined user operations which are likely to be performed on an Android client application for the MPower MessageBoard, should be performed on the proof of concept application.

The measurements should help identify Android's effectiveness and performance on parsing SOAP responses. Additionally, by performing the operations an impression of the application responsiveness is experienced.

Measure performance on large SOAP responses A defined set of SOAP responses with increasing complexity should be processed by the proof of concept application and the processing duration of each response should be measured.

The action should identify Android's performance on parsing larger responses and by this help to identify how Web services can be designed to better fit client applications running on Android.

Measure performance on native Java The same measurements performed on Android, are to be run on a desktop computer with native Java.

The results creates a reference for the measurements on Android. The comparison of the Android and native Java results enables a comparison on how Web services should be design for use on Android in difference to a desktop computer.

1.4 Outline

Chapter 1 Introduction Gives an introduction to the project, research questions, research method and this outline.

Chapter 2 Prestudy An initial study of the Android platform, its development tools, introduction to service oriented architectures, SOAP, the MPower platform, and a basic introduction to existing research.

Chapter 3 Architectural alternatives and their impacts A presentation of architecture alternatives and an analysis of quality attributes for each presented architecture.

Chapter 4 Basic Web service invocation on Android An initial study and implementation of a Web service client on Android. Identify minimal requirements and an initial response time measure.

Chapter 5 MPower proof of concept Android client Description of a MPower proof of concept application on Android, and a deeper study of the performance of such an application. Identifies best practice guidelines for clients and Web services.

Chapter 6 Conclusion A final conclusion which answers the research questions defined in the introduction.

Chapter 2

Prestudy

This chapter gives an introduction to the Android platform and the development tools. It gives a brief introduction to service oriented architectures (SOA) and describes MPower, a service oriented architecture research platform used throughout the study.

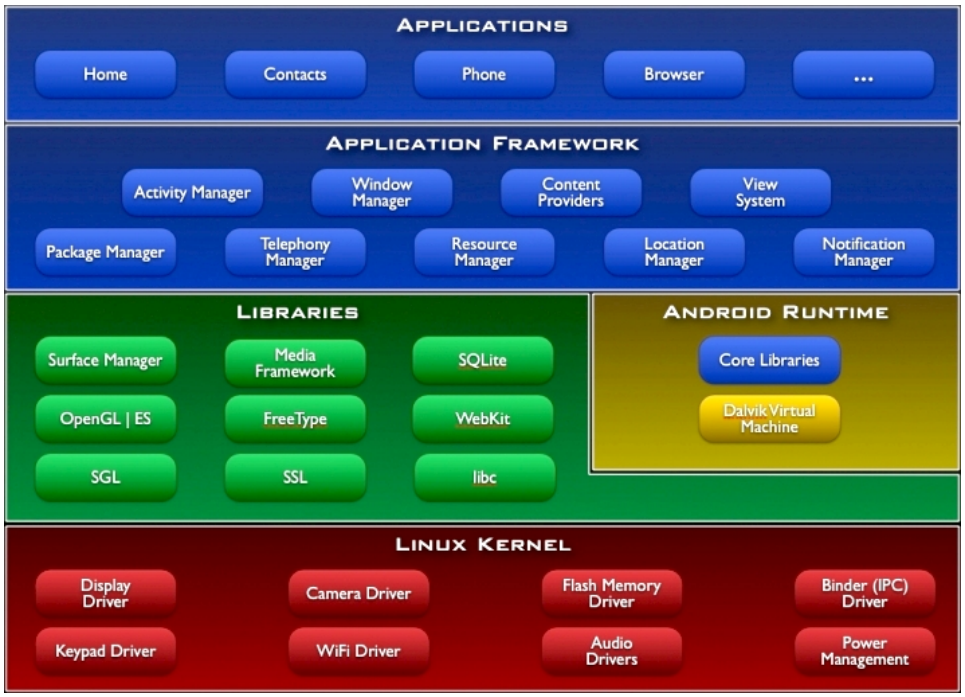

Figure 2.1: Android system architecture as found in [13]

2.1 Android platform and development environment

Android is a software stack for mobile devices that includes an operating system, middleware and key applications[13]. It relies on a *Linux 2.6* kernel for core system functionality[13] and runs code written in the *Java* programming language on a specially designed virtual machine named *Dalvik*. *Dalvik* executes *Dalvik Executable* files which are *Java* compiled classes optimized to minimize memory footprint[13]. The overall system architecture is found in figure 2.1.

2.1.1 Android Software development kit (SDK)

The *Android SDK* consist of several tools to help *Android* application development. This includes both an *Eclipse IDE* plugin, emulator, debugging tools, visual layout builder, log monitor and more. The tools included in the SDK are described in appendix B.

2.1.2 Why Android?

In contrast to most other mobile platforms, Android is available as open source software. This enables devices to be customized without

restrictions and enables any device manufacturer to ship devices with Android. Likewise are developers able to distribute applications to any Android device trough the *Android market*. Unlike *Apple's iPhone platform*, application distribution does not require any external review or acceptance and multiple application markets exists.

Within the context of this thesis, Android is especially interesting due to it's state of the art status. It is a newly released platform which has already gained massive support from device manufacturers like HTC, LG, Motorola, Samsung, and Sony Ericsson. This is expected to make it an attractive platform, since the development of a single application can reach a broad range of devices, all with a rich user interface experience.

2.1.3 Android availability

By the beginning of this project *HTC Dream*, marketed as *T-Mobile G1*, was the only available Android device. This device was only available at T-Mobile markets, thus not available in Norway. The first phone to be released on the Norwegian market is the *HTC Magic*, which is available for pre-orders as of 4th of June 2009. Because of this lack of available Android devices, applications and analysis performed during this thesis, will be performed on a Nokia N800 internet tablet which runs Android thanks to the NITdroid project[1]. NITdroid is *"a kernel and userspace port from scratch of the Android operating system to the Nokia internet tablet's hardware"*[1]. The device is shown in figure 2.2.

Nokia N800 compared to the HTC Dream

Running NITdroid on the Nokia N800 implies a rather undocumented hardware support. Hardware driver compatibility issues and bugs, are likely to appear or give performance penalties to the Android system running on the device. After booting Android on the device, this is immediately shown by the touch screen being a bit hard to use and the software keyboard constantly crashing. Nevertheless, the Nokia N800 device is believed to give a good indication of Android's performance, although the HTC Dream is rated with a processor running at 528 MHz versus Nokia N800's 330 MHz. A summary of the device specifications are given in table 2.1.

2.1.4 Third party code library support

The Android platform is not a complete implementation of the *Java API*. Instead it is a defined subset of the API. This imposes challenges in using external libraries which depends on parts of the API which is not included.

[1] Available at `http://code.google.com/p/nitdroid`

Figure 2.2: Nokia N800 running NITdroid (Android port)

	Nokia N800	**HTC Dream**
Processor (CPU)	ARM1136, 330 MHz	ARM11 Qualcomm MSM7201A 528 MHz
Memory	128MB	192MB
Network	802.11b/g WLAN	HSPA/WCDMA, GSM/GPRS/EDGE, 802.11b/g WLAN

Table 2.1: Nokia N800 compared with HTC Dream

A complete list of the Android implemented and missing packages is found in appendix A.

Missing *Java API* dependencies are unfortunately not possible to add, since application developers are not allowed to add classes within the java.* or javax.* packages. The restriction is not found documented, but is confirmed on the official *Android IRC* chat room and an attempt to create a class within these package names, gives a compilation error.

This limits the choice of usable third party libraries to the ones which only depends on *Android* implemented *Java API* classes or where the source code is available. In such cases, Android specific implementations of the dependencies can be added to the libraries. Unfortunately, this often would require major rewrites of the libraries and requires a deep knowledge the library itself and the missing dependencies.

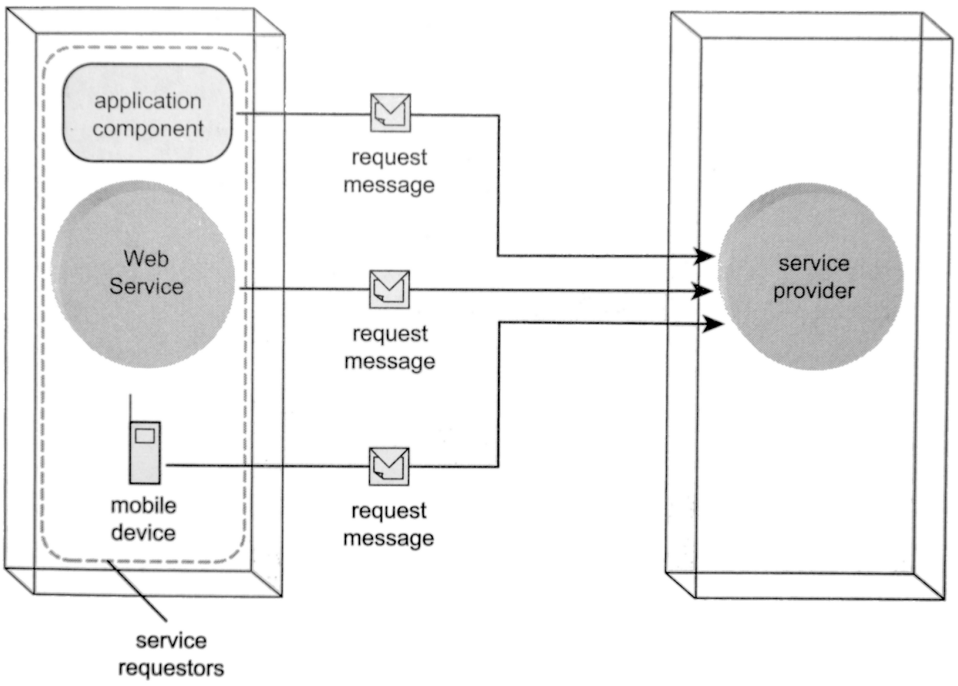

Figure 2.3: Service Oriented Architecture request overview as found in [7, p. 117]

2.2 Service Oriented Architecture introduction

A service oriented architecture (SOA) is based on the principle of separation of concerns[7, p. 32]. These concerns are separated as modules, or services, which communicates with each other through messages.

Although SOA can be built using a range of technologies, the scope of this thesis limits and focuses SOA to consist of an architecture which

a) uses SOAP as messaging protocol;

b) uses HTTP to transport messages; and

c) uses Web Services Description Language (WSDL) to describe available services.

Figure 2.3 shows multiple clients requesting a service's response. Response messages are not shown, but are returned to the initial requester. Figure 2.3 also shows how services can be both service providers and requesters. This enables a range of services to be *assembled* so that they together solve a larger task[7, p. 125].

2.3 Web service messaging

A SOA architecture is not dependent upon a specific technology, since different protocols for messaging can support the principle of separation of concerns.

Different protocols are mainly interesting for this thesis due to the different processing and memory requirements they may have. It is to be noticed that most SOA platforms supprts only a single or a few messaging protocols, thus making use of new protocols a non-trivial task for large scale systems.

2.3.1 Plain HTTP Post

Plain HTTP Post requests are typically simple messages encoded as *application/x-www-form-urlencoded* as defined by the HTTP protocol [22]. The response is typically in an application specific XML encoded format. This makes processing and memory requirements of message creation and parsing low, but it also restricts the messages by making complex requests hard to create and responses are very specific to a expected result.

2.3.2 JSON and REST

JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format[6]. Android have built-in support for JSON, thus making it a compelling alternative to XML within the context of this thesis.

Representational State Transfer (REST) is an architectural style for distributed hypermedia systems[8, p. 76]. It describes an architecture where each resource, such as a web service, is represented with an unique URL. The principle of REST is to use the HTTP protocol as it is modelled[20], thus accessing and modifying the resources through the standardized HTTP functions GET, POST, PUT, and DELETE.

REST is not in itself a standardized protocol, but defines principals on how to use existing standards. In combination with for instance JSON, they define a Web service architecture with an increasing support. Notice that REST does not require JSON as data interchange format.

REST based Web services' success is exemplified by major websites like Flickr, del.icio.us, eBay, Google, and Amazon which now offers access to Web services based on REST and JSON. Additionally, the WSDL 2.0 specifications now supports all HTTP functions[5], which enables REST services to be described.

JSON/REST based Web services are designed to be lightweight and easy to access, but lacks more complex functionality like type checking and adhesion to a contract.

2.3.3 SOAP messaging protocol

SOAP is a message transport protocol which has been accepted as the default message protocol in *SOA*[7, p. 142]. SOAP messages are created by wrapping application specific XML messages within a standard XML-based envelope structure[15, p. 55]. The result is an extendable message structure which can be transported through most underlying network transports like SMTP and HTTP.

SOAP Criticism

One of the main criticisms of SOAP relates to the way the SOAP messages are wrapped within an envelope. With HTTP as transport layer, most requests are typically sent as POST requests to a single URL, thus ignoring the HTTP resource-oriented design[20]. For example modelling requests as POST when they in fact could be modelled as GET, PUT, POST or DELETE requests results in HTTP traffic which is hard to monitor by for example firewalls[2].

2.3.4 Conclusion

Within a larger context, SOAP is the de-facto standard for Web service message exchange. A range of tools and application platforms are available with SOAP support, which enables techniques like model-driven development to be used in service design.

Within a mobile context, the *REST* architecture is considered as more light-weight[20] than a SOAP based Web service architecture. Additionally the Android platform have built-in JSON support which enables that objects can be directly converted from Java to JSON objects and back again.

A move towards a *JSON/REST* based Web service architecture and it's impact on mobile clients is outside the scope of this thesis, although an interesting subject for further investigation.

2.4 Mobile computing characteristics

Mobile computing is in [21] characterized by four constraints:

- *"Mobile elements are resource-poor relative to static elements.*
- *Mobility is inherently hazardous.*
- *Mobile connectivity is highly variable in performance and reliability.*
- *Mobile elements rely on a finite energy source."*

All these constraints puts restrictions and consequences for any architectural choices made. The constraints also are self contradicting. On one side, because of the resource poorness, lower trust and robustness, the constraints argues for a static server design. On the other, the varying network and finite energy resources argues for self reliance.[21].

An architecture for mobile SOAP clients must at least be able to support a static server design. The degree of self reliance on the other hand depends on the end users' requirements. A mobile client can be as simple as a web application with a HTML frontend designed for small screens. Clients will in these cases have no self reliance, since they are dependent upon an available static server. On the other hand, by having the clients cache it's data and do synchronization against the server, almost complete self relied clients is possible.

A description of the identified architectures for mobile SOAP clients and their differences is found in chapter 3. This analysis uses the constraints given above as a basis for the evaluation of each architecture. Most of this analysis is believed to be applicable not only to Android, but also most other modern mobile platforms.

2.5 Time limits on user feedback

In order to evaluate how effective a Web service client on Android must be, basic usability guidelines from Jacob Nielsen have been studied and used as guidelines. Jacob Nielsen is regarded as one of the gurus on Web usability and describes three time limits on user feedback in Web driven applications. These limits are described as follows in [19].

> **"0.1 second** *is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.*
>
> **1.0 second** *is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.*
>
> **10 seconds** *is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect."*

It should be noticed that the limits are not to be treated as absolute, but rather be part of the evaluation on how quick a user expects some kind of feedback. For this thesis, the limits are used in the evaluation on how fast a Web service client actually must respond to user actions in order for the user to stay focused and perceive the application as responsive.

2.6 Android Web services support

The Android platform has no built in SOAP support for Web services. However, it does have built in libraries for both HTTP communication through *Apache HTTPClient* and XML construction and parsing through application programming interfaces (APIs) like *SAX*, *DOM* and *XmlPull v1*.

2.6.1 Manually create SOAP messages

Android's built in XML and HTTP support enables SOAP request messages to be constructed manually and dispatched through the *HTTPClient API*. The SOAP response is then manually parsed and converted into Java objects.

Since mobile devices have limited resources, this manual way of creating and parsing SOAP messages could help minimizing memory and processing requirements. Additionally, mobile applications are built more task oriented than desktop and web applications which normally makes such manual SOAP handling far less complex than other SOAP clients.

Although manual SOAP handling could be justified in many cases, this thesis will focus on the use of third party libraries.

2.6.2 Third party libraries for SOAP support

Through *Apache Axis2* found at `http://ws.apache.org/axis2`, Java is given full support for SOAP based Web services. This includes code stub generators from WSDL definitions.

KSoap2 is a SOAP library made specifically for J2ME. This makes it much more light-weight than Axis2, thus believed to fit Android better. Several posts on the Android development forums confirms that KSoap2 works good not only on J2ME, but also on Android. However since KSoap2 is designed for J2ME, KSoap2 is not using Java features like reflection which is not available in J2ME.

KSoap2 does not have support for stub generation from WSDL definitions, but do have mechanisms to help serialization and deserialization of Java classes.

2.7 The MPower platform

MPOWER is a SOA platform targeting easier development and deployment of services helping persons with cognitive disabilities and elderly[12]. Currently, the platform has been tested with various proof of concept clients, including HTML based frontend and Windows Mobile client. The proof of concept applications currently released has been rewarded with positive response from the end users making it a interesting platform to extend with new clients.

Throughout this thesis, all architectural analysis and performance benchmarks have been performed with the MPower platform in mind. However, the MPower platform is built on a standard service oriented architecture. Therefore the results and evaluations should be applicable also on other similar platforms.

2.7.1 MPOWER benefits of mobile clients

So far the MPOWER platform primarily have been tested with desktop or web based clients. Such clients are restricted to be used in the physical location where they are deployed.

A mobile MPOWER client is identified to improve several of the features announced to be particularly supported in MPOWER, found at [12]. These features are foremost gained by the *a*) increased system availability to mobile users which often change context; and *b*) access to mobile devices' sensors like GPS location and accelerometer sensor.

Additionally, a mobile client would allow a tight integration with basic phone functionality like initiating phone calls, SMS or MMS messages. One could also benefit from bundled applications like the calendar with alarms and to-do lists by implementing synchronization services. Such use of built in functionality on the mobile device is believed to decrease required development time and increase the usability from a end user's perspective.

2.8 RPC versus document style Web services

RPC and document style Web services have a confusing dual meaning in a SOA architecture. Within a WSDL and SOAP formatting context, document style and RPC style describes syntactical differences of a SOAP message. Within a broader context, the difference between document style and RPC style Web services is based on the invocation and response type. A document style Web service is designed, as the name implies, to respond with a document type of response. This typically means a lager chunk of data, like a list of complex data. A RPC style Web service on the contrary, is designed to have a larger number of requests, each with a smaller response. In the latter years, document style Web services has become more and more common.

Although a clean SOA architecture should enforce to use only one of the invocation styles, there is no technical problems mixing them within an environment. It should also be noticed that document style invocations, generally are harder to make abstract and reusable since a larger response will require more information on what to return.

MPower is designed to use as much document style services as possible.

2.9 Existing research

Topics concerning the use of service oriented architectures on mobile devices is not a new research field. Mobile platforms like *Microsoft's .NET Compact Framework* have had SOAP messaging support since it's release[4]. Likewise has the J2ME platform had Web service support through the use of libraries like KSoap[16] and the *JSR-172 Web Services Specification*[14].

However, in contrast to the .NET Compact Framework, Android does not have any built in SOAP support neither in the development tools nor in the platform API. Like the J2ME platform, Android is Java based, but Android is supposed to be much closer to the desktop Java (J2SE) and devices running Android will typically run on faster hardware than devices with J2ME. These differences makes it hard to directly use the studies performed on other platforms.

Since Android is a new and community supported platform, the most current research and documentation is the resources made available by people who have tried the platform and shares their knowledge and experience on blogs, developer forums, and communication channels like IRC. The lack of articles and books, especially on topics like SOAP messages on Android, makes these shared experiences the only updated source of information.

Chapter 3

Architectural alternatives and their impacts

This chapter presents three architectural alternatives for accessing SOAP based Web services on an Android client. Further, an analysis of the three architectures and how they will affect the client application is performed and the results are presented.

3.1 Architectural alternatives for mobile Web service access

Three well suited client/server architecture patterns are identified for mobile devices and SOA architectures. HTML frontend, Web service gateway[16] and direct Web service invocation[16].

3.1.1 HTML frontend

A Web service client is placed on a server as a web application serving dynamically generated HTML responses to HTTP requests. A standard web browser is used on the mobile client to access and communicate with the web application. As seen in figure 3.1, requests are plain HTTP requests from the browser to a HTTP server. The HTTP server runs a web application which uses the available SOAP services, and returns a HTML formatted response. User input is handled through standard HTML form posts.

3.1.2 Direct Web service invocation

All requests to the Web services are performed from the mobile device. Implies that the mobile client must be able to wrap and unwrap SOAP messages.

3.1.3 Web service gateway

A Web service client is placed as a gateway server and converts incoming requests from the mobile client to Web service invocations. Web service responses are then converted to a more lightweight format than SOAP and returned to the client. Responses can typically be formatted as XML, WBXML, JSON or other formats with a simpler structure than complete SOAP messages. The architecture is seen in figure 3.3. Notice that one request to the gateway, could result in multiple SOAP request and responses.

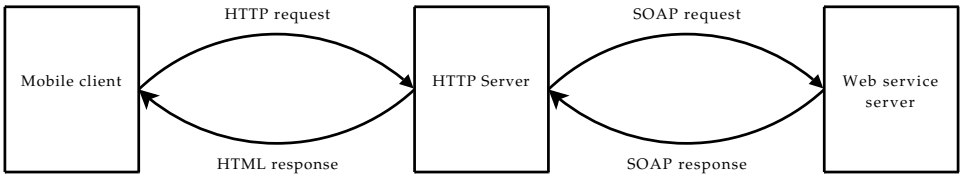

Figure 3.1: HTML frontend illustrated

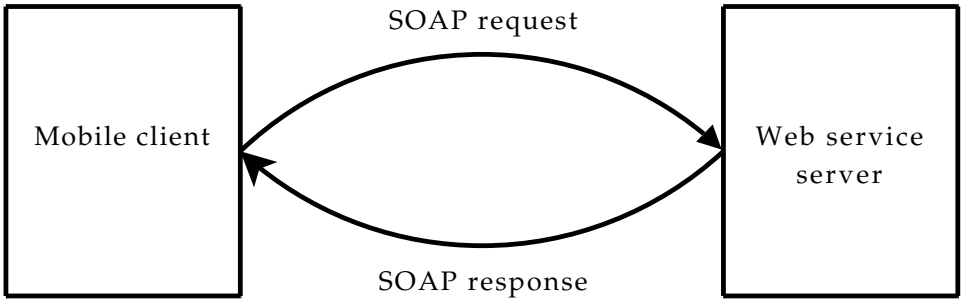

Figure 3.2: Direct Web service invocation illustrated

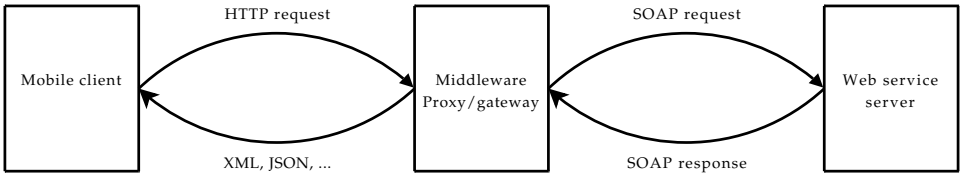

Figure 3.3: Web service gateway illustrated

3.2 Quality attributes and attribute tactics

Bass, Clements and Kazman defines the concept of *quality attributes* for architectural evaluation. These attributes are characterized by a set of quality attribute scenarios[3]. In order to compare the three architectural designs given in section 3.1, these quality attributes and their characterisation through general attribute scenarios are used.

As described in [3], each quality attribute have a set of tactics which should enforce the architecture to support the focused quality attributes.

During the following sections, each of the quality attributes are briefly presented and defined based on the general quality attributes given in [3]. Further are the three architectural designs evaluated on how they support the achievement of each quality attribute. It should be noticed that this evaluation is based on the tactics presented for each attribute as given in [3, chp. 5].

3.2.1 Availability

Availability is defined to concern any failure which makes the system unable to deliver the service it is design to deliver[3, chp 4]. This includes both service for the end user and other systems relying on the system. It also includes a systems possibility to recover from an unavailable state and resume it's normal operation.

With the increasingly available mobile networks, one could have believed that off-line support became an issue out of date. However, questions like how to handle connectivity interruptions are of more interest than ever. Applications are expected to be available at all times and always updated with the most recent data. A transparent handling of changes in network connectivity includes dealing with conflicting revisions, data changes, and avoidance of locked data. All with as little user intervention as possible.

In this analysis it is focused on loss of network connectivity, which should be expected to happen frequently in a mobile environment.

HTML

Pros A web application enables a centralized point of control which makes the system availability easy to monitor and redundancy tactics can be applied. Synchronization issues of data is typically easy to handle, since clients are directly requesting data from the server.

Cons Since the HTML frontend architecture is based on HTTP requests of basic HTML pages, such an architecture would not be possible to use in an environment without a network connection. The web application does

neither have any ways of detecting which clients are online and in use, except when the clients actually are sending a HTTP request.

Depending on the implementation and security measures applied, recovering after loosing network connectivity might require users to relogon to the system. This is especially true if the client's IP address is changed or a session has timed out. This makes rapid loss of network connectivity annoying to the users.

Direct invocation

Pros Direct Web service invocation supports availability tactics like ping/echo and heartbeat. This means that each mobile client can publish their availability to the server by sending frequent messages about it's presence and detect the server availability by issuing frequent response requests to the server. The client can by this be designed to use an internal storage when it is off-line and resynchronize with the server when it is online again. This enables clients both to be used without network connectivity and to quickly detect a required network connection and recover it's normal operation.

Cons Implementing a system which transparently handles online and off-line modes is hard, since resynchronization is hard in a multiuser environment.

Gateway

Same as direct invocation.

3.2.2 Modifiability

The purpose of a SOA architecture in itself is to localize changes and separate concerns. Within a mobile client context, the question of portability and the work of extending the MPower platform with new services is especially relevant.

HTML

Pros Web applications are relatively easy to make and a range of frameworks are available with direct SOAP support. One web application will support any mobile device with a internet browser and deployment is only necessary on one server. Thus, are new features and changes very quick and easy to set in production.

Cons Multiple views could still be necessary in order to support multiple screen sizes.

Direct invocation

Pros Client applications can use existing device functionality like calendar and contact list instead of implement such features.

Cons Updated application versions must be deployed to all devices.

Gateway

Pros Could make porting clients to new devices easier, since the gateway may hide Web services complexity and serve more client specific communication interfaces.

Cons Three points of implementation creates more complexity when a new service is created, changed or removed.

3.2.3 Performance

The overall goal of performance attribute is to lower the time from request to response is available.

HTML

Pros All SOAP and heavy processing is done server side.

Cons Each request requires a complete post and wait for a new page as response. Thus, the response time will be dependent on size of each page.

Direct invocation

Pros Supports to have local cache of both already retrieved data and, by using optimistic fetching, data expected to be requested. Thus, network usage can be optimized and the response times can be minimal. Such optimizations are especially important in high latency data networks like GPRS, EDGE and UMTS networks.

Cons Direct invocation with SOAP messages typically have a significant size and complexity overhead. This leads to more data transfer and more processing required to create and read the messages.

Gateway

Pros Supports the same tactics as direct invocation but additionally can reduce computational overhead by using more lightweight, system specific, transport formats than SOAP.

Cons None major cons identified.

3.2.4 Security

Small mobile devices are easy to loose and get hijacked by third-party which should not have access to sensitive data. With MPower, this is especially true due to it's medical sensitive application context.

HTML

Pros As long as a secure transport protocol like HTTPS or VPN systems is ensured, a minimal amount of data is stored on the actual client.

Cons Cache is still vulnerable and HTML suffers from security issues like session hijacking. The web server is vulnerable to attacks and adds another software system to secure.

Direct invocation

Pros The developer will have complete control of the data stored locally on the actual device. Enables encryption and cache deletion when appropriate.

Cons Cached and saved data on the mobile device is vulnerable since the data is stored on the actual device. If anyone gains access to the device, they do not need access to the actual system to get the data.

Gateway

Pros Same as direct invocation.

Cons Data is replicated at least three places. In addition to secure the Web service server, the gateway must also be secured.

3.2.5 Testability

Testability concerns the implications of testing the correctness of the system.

HTML

Pros A number of web application test frameworks exists, making HTML frontends easy to test.

Cons The HTML rendering on mobile devices might vary from device to device. How the pages are rendered on a device is hard to automatically test.

Direct invocation

Pros Tests can be made with unit testing frameworks and mocks[1] which mimics external functionality.

Cons Each mobile platform, must have platform specific tests written to them. User interfaces are in general hard to test.

Gateway

Pros Simpler interfaces between the remote platform and the mobile client makes testing on the mobile client easier since the mobile client can potentially be made simpler with less complexity.

Cons The testing of the gateway introduces another software platform to test. Thus, simple to test, it introduces more code and another environment to test.

3.2.6 Usability

MPower is targeting persons with cognitive disabilities and elderly. Thus, usability is essential for the platform to be used.

HTML

Pros HTML has become a very well-known user interface to most people and is easy recognizable. Since HTML allows identical user interface on a range of devices, the interface will also be recognizable independent on the actual device a user uses.

[1] A mock is a software component which mimics another software component during software testing. A Web service mock can for example be used to return test data responses to a Web service client in order to have a controlled test environment.

Cons An architecture based on a HTML frontend, disables device specific benefits. A short list of example follows.

- Data from a GPS or an accelerometer is not possible to send to a HTML page.
- Integration with other device specific application like for example phone, SMS, MMS, calendar, alarm and notification functionality.
- User interaction styles are restricted to the ones defined by form inputs of HTML.

A sporadic usage pattern of the application is difficult to achieve. Users wishing to check the system for any update must open the browser, navigate to the systems logon page, enter credentials and find the correct page.

Additionally HTML, without the use of JavaScript, only support complete page reloads. Thus, dynamic update of the user interface based upon state and user interaction is impossible.

Direct invocation

Pros The direct invocation architecture is dynamic by nature since it is based upon using the native user interface and programming environment. This enables better support for adaptive user interfaces, device specific functionality, and sporadic usage. Information can be regularly pulled from the server and integration with the device enables device application to be synchronized with the Web service server.

Cons Building a user interface easily recognizable by users on any device, could be difficult because of the use of native user interface components.

Gateway

The gateway have the same pros and cons as the direct invocation above.

3.3 Architecture development impacts

The complexity of application development depends on range of factors, including but not limited to

- Number of supported platforms
- Required code and functionality abstractions and reusability
- Off-line usage support
- Possibility to auto generate code
- Availability of third-party code libraries
- SDK development support

The number of supported platform is specifically an issue for mobile application development. The large number of platforms, have little or no possibility to run the same application code. Hence, a pure HTML based front-end design lowers the complexity of supporting all platforms dramatically, since most devices have a web browser.

3.4 Conclusion

Each of the architectures in section 3.1 have their advantages and drawbacks. When choosing an architecture, the quality attributes should be prioritized against each other. This allows the architecture to be chosen based on the most important quality attributes one wishes to achieve.

In the context of MPower and this thesis, a goal has been to build a dynamic platform which is easy to use. A mobile client should be usable without network connectivity and the possibility to synchronize MPower information directly with Android's device specific functionality should be available. Thus, a HTML frontend would not support these requirements and a gateway is preferred to be avoided due to the additional development complexity.

A summarization of the advantages and drawbacks found for each architecture follows.

3.4.1 HTML frontend

The HTML frontend is typically ideal when a device is expected to always have an available internet connection and the same application must be available on a range of different device platforms. Additionally, a HTML frontend enables efficient development thanks to the available development tools with SOAP support.

Advantages

HTML frontends are quick and easy to create. They can be deployed on any web server and a range of SOAP client frameworks are available to web applications. Additionally, HTML frontends are device independent. Any mobile device with an internet browser, can access the same web application.

Drawbacks

By using a web browser interface on the client, any device specific feature is not possible to utilize. Synchronization with device applications like a calendar, contact list, map, SMS and so on is not possible.

Unfortunately, the client is required to be online for the web application to be available. When the internet connection is lost, the application is unusable with the risk of loosing data which the user was working on submitting.

Although dynamic web interfaces are built for desktop browsers, mobile browsers still have very limited or no Javascript support. Thus,

HTML frontend applications are limited to the default full page reload on updates.

3.4.2 Direct Web service invocation

A direct invocation is typically ideal when a client application must be possible to use without an internet connection and a high degree of modifiability is necessary.

Advantages

Mobile clients can have internal caching and synchronization logic. Several clients can connect to the same service architecture and make any application supported by the given services possibly without any specific service design.

Drawbacks

SOAP messages are XML based and typically large and complex[18]. Such complexity results in both computational and data transfer overhead.

3.4.3 Web service gateway

The Web service gateway is first and foremost found beneficial when a direct Web service invocation style is not usable due to too much message overhead or when the Web services are not open for changes.

Advantages

By using a gateway between the mobile client and the Web services, one can leave the services unchanged while building more lightweight interfaces for mobile clients.

Possibility to use compressed and task specific data representations, reduces data traffic and processing. This improves application performance and lowers the application latency.

Drawbacks

A gateway creates yet another architectural element to maintain. Any new features requires implementation both on a Web service server, gateway and mobile client.

Additionally, the gateway approach is not standardized. Thus, it easily becomes application specific and ignores Web service's principle of separation of concerns.

Chapter 4

Basic Web service invocation on Android

This chapter presents an example approach for direct invocation of SOAP based Web services from an Android client application and an evaluation of the approach.

4.1 Basic Web service invocation

4.1.1 Research action, justification and goals

The complexity of most MPOWER services, justified an initial study of calling simpler Web services from the Android platform.

The activity will have the following goals.

- Try to use the Axis2 library for Web service invocation.
- Try to use the KSoap2 library for Web service invocation.

A Web service found at W3Schools' website, `http://w3schools.com/webservices/ws_example.asp`, which converts temperatures between the Celsius and Fahrenheit scales was used for the clients.

4.1.2 Results

Axis2 for Web service invocation

Axis2 is Apache's reference SOAP implementation and consists of several modules, both for Web service server capabilities, client support and code generation tools.

The Axis2 did unfortunately have dependencies within the Java API, which was not implemented in Android. Thus, it is not possible to use without major changes.

It is also to be noticed that Axis2 is a large and complex library possibly not well suited for mobile device with lower resources.

KSoap2 for Web service invocation

KSoap2 is a light-weight SOAP client library targeting *Java 2 Micro Edition* platforms. KSoap2 is also released with a *Java 2 Standard Edition* version which works without modifications on the Android platform.

```
public class CelsiusFahrenheitConverter {
  String soapAction = "http://tempuri.org/CelsiusToFahrenheit";
  String methodName = "CelsiusToFahrenheit";
  String namespace = "http://tempuri.org/";
  String url = "http://www.w3schools.com/webservices/tempconvert.
      asmx";

  Transport httpTransport = new HttpTransportSE(URL);

  public String celsiusToFahrenheit(int celsius) throws Exception
      {
    SoapObject request = new SoapObject(namespace, methodName);
    request.addProperty("Celsius", celsius);
```

```
    SoapSerializationEnvelope envelope = new
        SoapSerializationEnvelope(SoapEnvelope.VER11);
    envelope.setOutputSoapObject(request);

    httpTransport.call(soapAction, envelope);
    SoapObject soapResponse = (SoapObject) envelope.bodyIn;
    SoapPrimitive body = (SoapPrimitive) soapResponse.getProperty(
        "CelsiusToFahrenheitResult");
    return body.toString();
  }
}
```

Listing 4.1: asic KSoap2 Web service request/response example.

Listing 4.1 shows a very basic use of KSoap2. It wraps an integer within a SOAP envelope and send the envelope to a service's end point through a HTTP transport. The response is then extracted from the KSoap2 serialization envelope and returned as a string. It should be noticed that this only demonstrates the most basic example and KSoap2 have a much more refined support for wrapping and unwrapping requests and responses.

4.1.3 Evaluation

Although an attempt to use Axis2 on Android failed, KSoap2 seems to give a beneficial support to this thesis' implementation needs.

By using a third-party library like KSoap2 when building a Web service client, it is easier to keep focus on the actual content of each SOAP invocation instead of the detailed XML structure of the message. Additionally, are message envelopes easy to change according to new SOAP versions or Web service implementations if necessary.

Based on the initial study, the KSoap2 library will be used throughout the rest of this thesis' implementations.

4.2 Code generation support

Automatic generation of code from Web services' WSDL files, especially simplifies the serialization and deserialization of XML and Java classes. Such serializing code would otherwise be coded by hand, often with repeating code patterns.

Axis2 have built-in support for generating stub classes and provides automatic mapping between XML and Java classes. Axis2 generated stubs does not fit on Android, since Axis2 is unusable, but the generator is easily tweaked to output code in other formats[9]. Such modifications allows generation of custom made Java bean stubs which for example can be used in KSoap2 or manually created SOAP environments on Android.

KSoap2 has not officially provided any tool for generating stub classes, but a user contributed patch is available at *KSoap2's SourceForge* page. The patch uses parts of the Axis2 library in order to analyze the WSDL and generate stub classes. Unfortunately this patch is totally undocumented, thus making it hard to use.

Based on the available alternatives to automatically generate code stubs, the code generation solutions are found too demanding to be used during this project. Not only because of the work involved in customizing and adapt existing solutions, but also due to the limited number of WSDL files the client applications which are to be developed will use.

4.3 Invocation performance

4.3.1 Research action, justification and goals

One of the issues concerning Web service invocation on a mobile device, is performance and latency. As a preliminary study the Androids performance on the most basic Web service call was performed. The goal was to identify Android's basic potential to host a Web service client. This includes identification of bottlenecks and performance on multiple sequential invocations.

As described in section 4.1.3, KSoap2 is used to leverage the test client development.

4.3.2 Testing environment

The temperature converter service described in section 4.1.1 and the code from listing 4.1 was used as a basis.

In order to avoid server bottlenecks and have a controlled server environment, *soapUI*[1] was used to set up a fictive SOA environment serving SOAP responses to SOAP requests. The environment was set up with a local wireless network infrastructure for data transport.

The Web service invocations were performed on a *Nokia N800* device with *Nitdroid* installed. The N800 is an internet tablet device designed to run with a light-weight Linux operating system. *Nitdroid* is a open source port of the Android platform to the *Nokia N800*. It runs a patched kernel with the complete Android stack on top.

4.3.3 Test measurements

A total of seven measure points were defined. Each point measured the invocation duration of a single or small set of Android API or KSoap invocations.

A total of 500 Web service invocations were performed in sequence. Each invocation measuring the durations stated above by saving the debug output to a log file.

createRequestData KSoap2 method which takes a KSoap2 SOAP envelope object and writes serializes it as as a byte array with the SOAP request.

setRequestProperties Creates a new connection represented as a `HttpURLConnection` object, which is part of the Java API and

[1]The official *soapUI* website states that: *"soapUI is the most used testing tool in the world for inspecting Web Services, invoking Web Services, developing Web Services, Web Service Simulation and Web Service Mocking, Functional Testing of Web Services, and load Testing of Web Services"*.

implemented in the native Android API. Additionally, it sets some basic HTTP headers to this connection.

openOutputStream Native Android API call to `getOutputStream()` on the current connection.

outputStreamWrite Writes the request data to the current connection.

connect Native Android API call to the current connection.

openInputStream Native Android API call to `getInputStream()`.

parseResponse KSoap2 method which reads the available data from the input stream and deserializes the SOAP response.

An overview of the method invocations in the performance benchmark is illustrated as a sequence diagram in figure 4.3.

4.3.4 Results

The measurements shows some irregularities with random requests lasting much longer than anticipated. A total of forty requests (8%) lasted more than a second. To verify the result, the same test was repeated on the *Android Emulator*. When running the same code on the emulator, no requests lasts more than a second. It is believed that the long durations happens because of bugs in Nitroid or other Nokia N800 specific issues. Also notice that most of these long durations lasts close to 3 seconds as shown in figure 4.3 and could be a socket timeout on the device.

As seen in table 4.1, 85% of the requests are shorter than 300 ms and more than 60% of the requests are shorter than 100 ms. When garbage collecting, which lasts approximately 120 ms and is included in the request duration measurement, is taken in account one could anticipate that approximately 85% of the readings are representable for the application running on a native Android device.

The shortest total request duration was 75 ms and the average request duration of the 90 percentage shortest requests was 137 ms. As seen in figure 4.3, the opening of the output stream takes the most time in average. It should be noticed that parsing of the response only takes 15 ms in average.

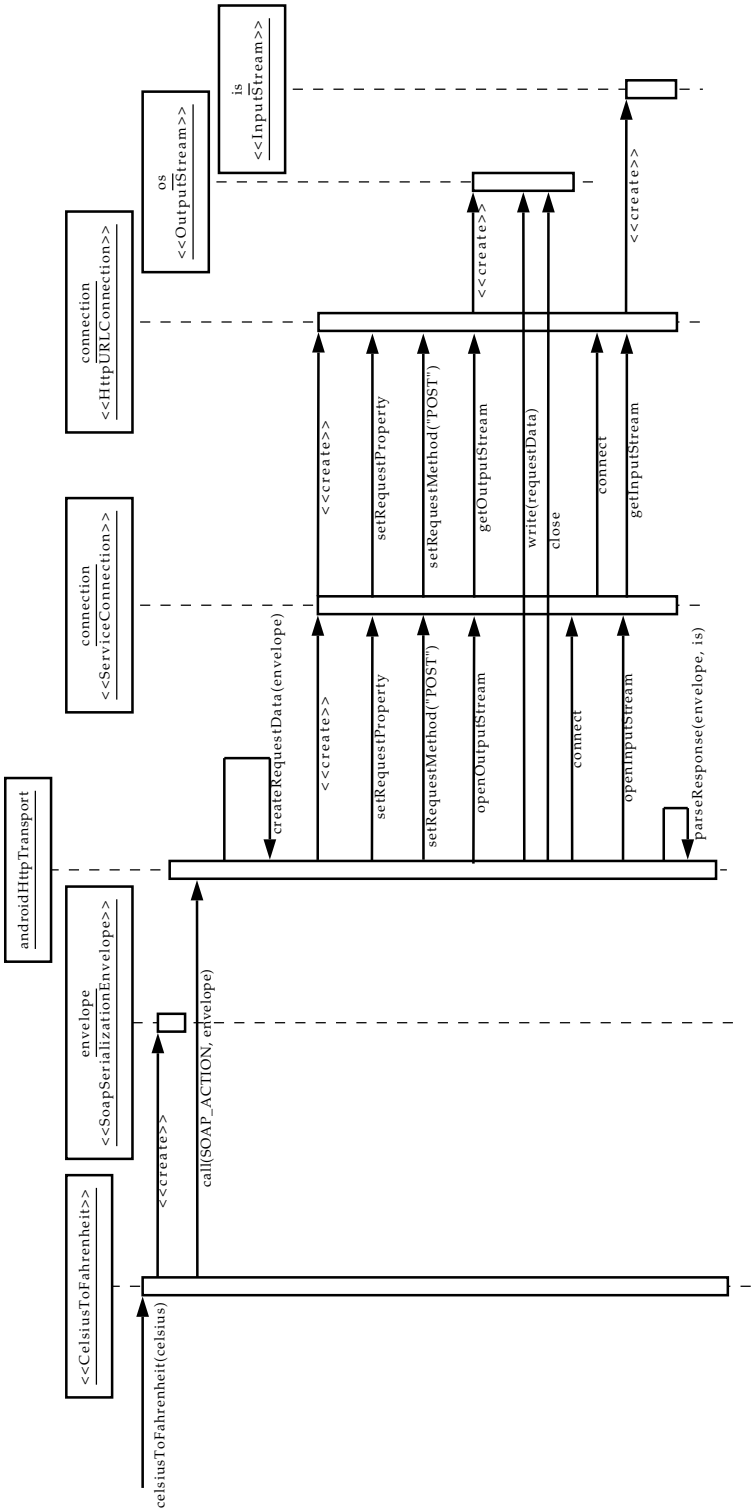

Figure 4.1: Sequence diagram which shows what happens in a KSoap2 Web service invocation.

Percentile	Request duration
100	21092 ms
95	3087 ms
90	675 ms
85	299 ms
80	290 ms
75	221 ms
70	216 ms
65	132 ms
60	98 ms
55	92 ms
50	90 ms

Table 4.1: Total request duration measurement percentiles.[a]

[a]Percentile is here defined as the request duration below which a certain percent of measures fall. For example do 60% of the measurements last 98ms or less.

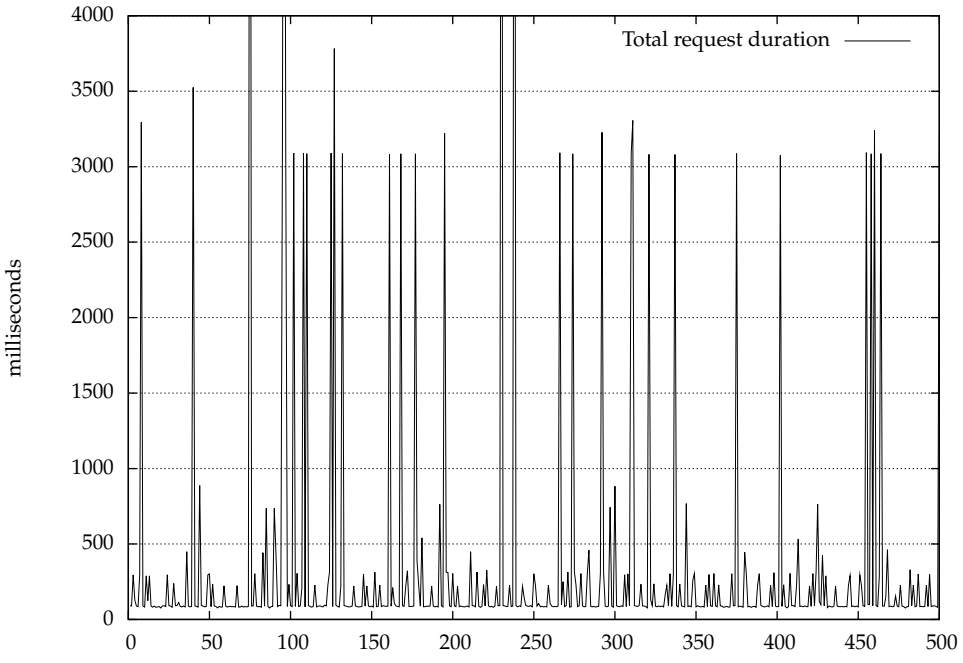

Figure 4.2: Total request duration for a simple SOAP request. Requests lasting longer than 4000 ms is not shown on the plot.

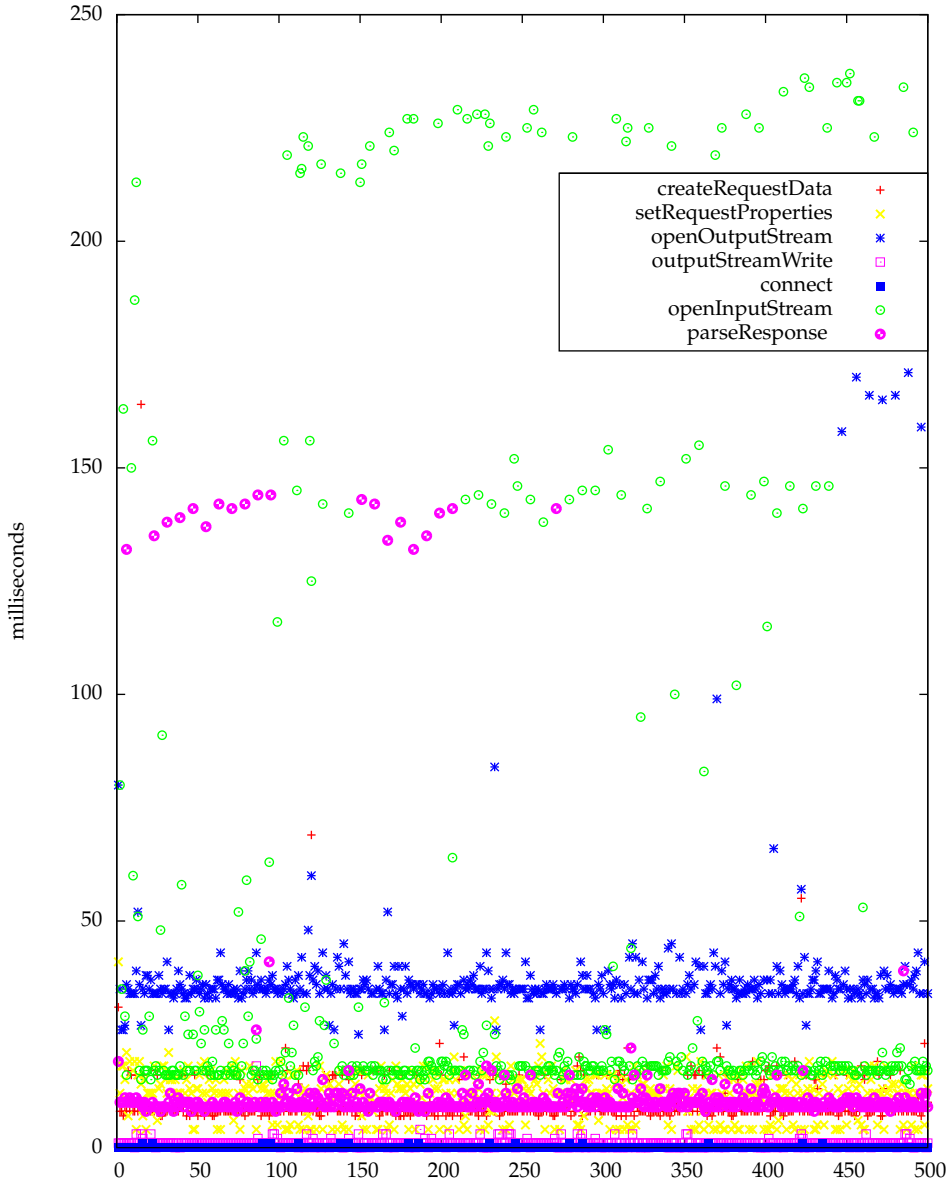

Figure 4.3: Measure points duration results. Measurements lasting longer than 250 ms is not shown on the plot.

4.3.5 Evaluation

Most requests lasted much less than a second, which is *the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay*[19]. Although the test is performed with an almost minimal SOAP request, the results indicates that the performance is well within the time limits required for building a responsive mobile client. Especially since the test device is not designed to run Android.

Chapter 5

MPower proof of concept Android client

This chapter presents a proof of concept Android application, which by direct invocation allows users to access the MPower MessageBoard Web service. Several benchmarks are performed on the application and the results are presented and evaluated.

5.1 MPower proof of concept client application

5.1.1 Research action, justification and goals

In order to verify the results from section 4.3.4 and to get a more actual experience with mobile SOAP client development on Android, a small proof of concept application was decided to be developed.

The goal was to create a small Android application which used some of the MPower Web services.

Such a proof of concept application enabled the client response to be experienced and measured.

5.1.2 Test environment

The application was built to support a *MPower MessageBoard* service. This is a *HL7*[1] which enables doctors and relatives to send text messages to a patient. The application should also provide an authentication screen. The goal is not to build a complete application, but rather a working example of how a mobile MPower client can be built on Android.

The client uses KSoap2 and performs direct SOAP invocations to a MPower application server located at Sintef, while the client is located at a remote site with an ADSL2 internet connection.

5.1.3 Benchmark description

The test client gives an actual feeling with the responsiveness of a SOAP client. Additionally, a set of application operations will be performed as described below. All benchmarks will be measured by the duration of each operation.

Benchmark operations

1. Authenticate a user.
2. Retrieve a list of 20 messages.
3. Retrieve a list of 20 messages without updating the user interface.
4. Remove a message.
5. Retrieve a single message.
6. Retrieve a single message without updating the user interface.
7. Post a new message to a user.

[1] HL7 (Health Level 7) is a standardization for health information data exchange.

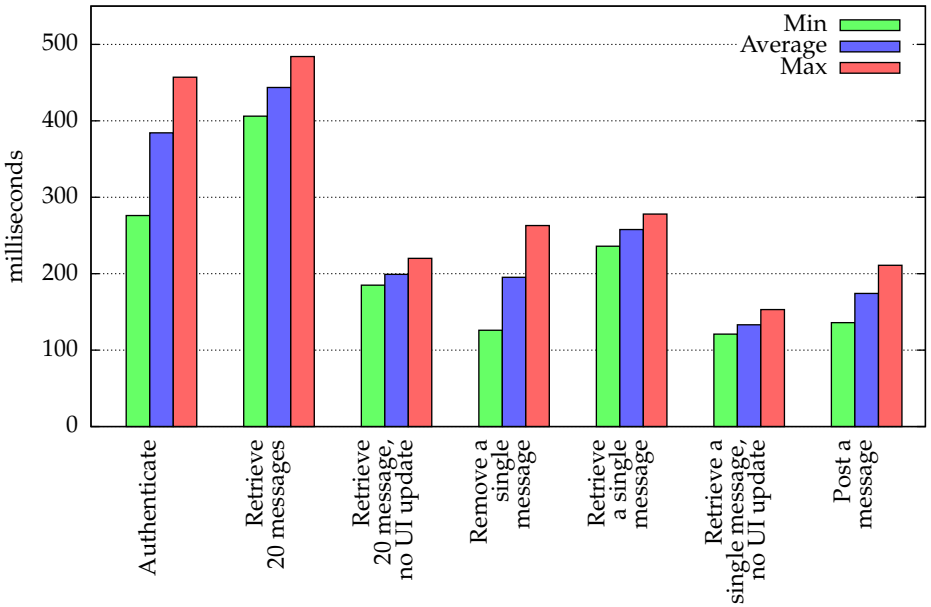

Figure 5.1: Benchmarks running on Nokia N800 with WLAN connection

All measurements includes SOAP request creation, network round-trip, response processing, and update of the user interface. Benchmarks will be performed both on the Nokia N800 device and Android emulator.

Each benchmark will be performed ten times and an average measure will be calculated and plotted together with the minimum and maximum duration. In order to simulate various network environments, the benchmarks are also to be run using the built-in GPRS, EDGE, and UMTS emulation options in the emulator.

In order to get an even and realistic performance measurement, measurements where the garbage collector runs or the server processes the request particularly slow, is discarded and the measurement is repeated.

5.1.4 Results

Screenshots from the proof of concept application running on Nokia N800, is shown in appendix C.

The results from the benchmarks are plotted in figure 5.1, 5.2, 5.3, 5.4, and 5.5. The plots shows the maximum, minimum, and average duration of each benchmark operation listed in section 5.1.3.

Throughout the benchmarks, the authentication invocation call took fairly long time to perform. Service invocations performed from native Java clients, indicates that this could be a server issue resulting in slow server

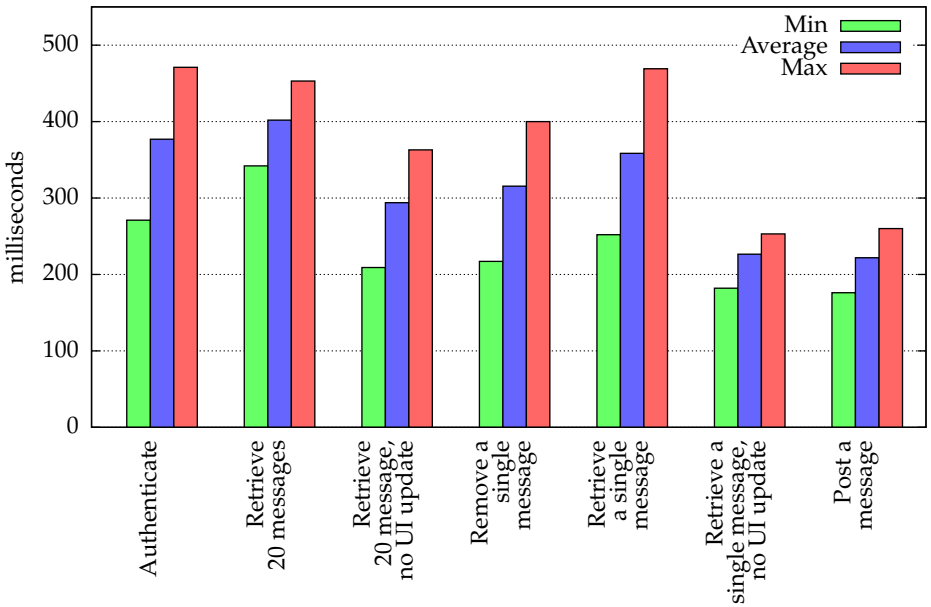

Figure 5.2: Benchmarks running on Android Emulator with LAN connection

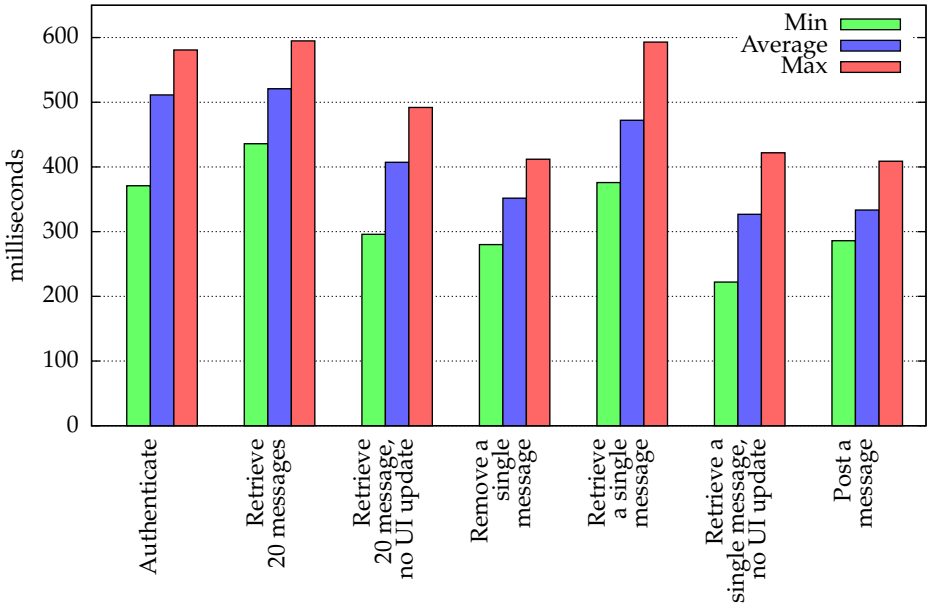

Figure 5.3: Benchmarks running on Android Emulator with UMTS simulation

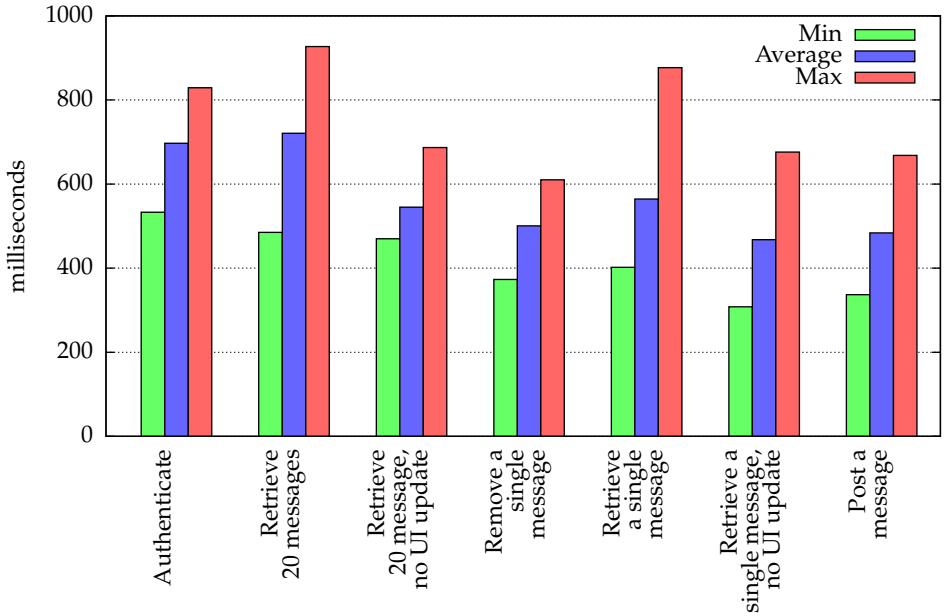

Figure 5.4: Benchmarks running on Android Emulator with EDGE simulation

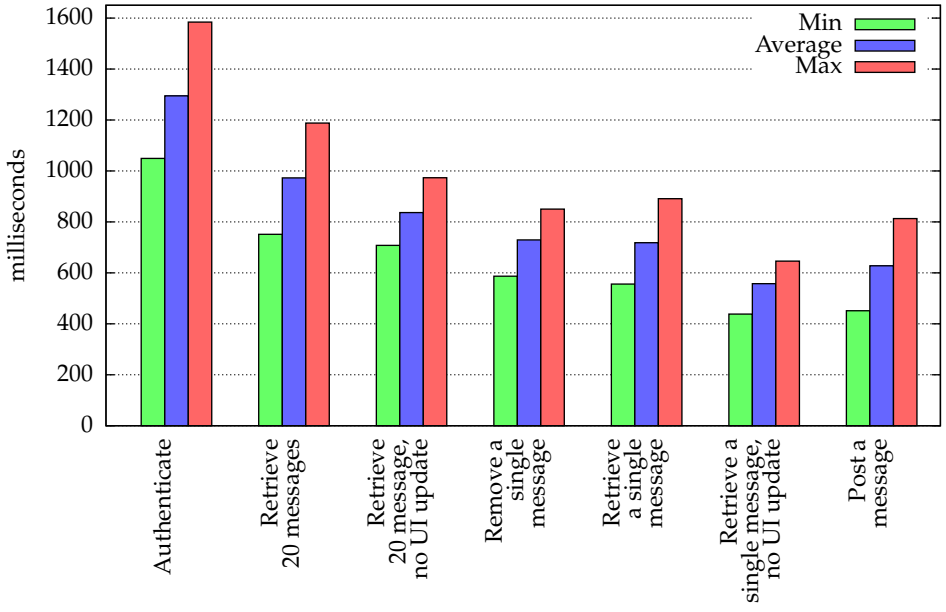

Figure 5.5: Benchmarks running on Android Emulator with GPRS simulation

responses. This is also indicated by the similar results on the authentication invocation when measuring native Java performance in section 5.3.4.

When latency and throughput simulation of GPRS, EDGE, and UMTS mobile networks are enabled on the emulator, the benchmark results varies more than using a LAN connection. Thus, the low number of measurements for each operation is believed to be too low to ensure valid results. The correctness of the network simulation is also uncertain and not found documented by Google. Nevertheless are the results believed to indicate the application performance when using devices within areas of good network signal strength.

Both the device and emulator clearly shows how updating the user interface impacts the performance. The performance penalty of updating the user interface is also strengthened in the retrieve message operations, because of these operations running in their own thread. Thus, thread switching is added to the complexity when the user interface is to be updated.

5.1.5 Evaluation

With a high speed network connection, like a wireless network, all measures are timed below 500 ms which is well within Jacob Nielsen's one second limit described in section 2.5.

Even though the performance decreases on benchmarks with mobile network simulation, most of the measures are still within Nielsen's one second limit. Thus, simple service invocation might be performed without having any progress bar shown, but might feel a bit sluggish to the user[19].

From the measured results, it should also be noticed that SOAP client applications which requires more than one SOAP invocation to show a response to a user's interaction, quickly will reach the one second limit. Such applications should either have Web services designed specifically to reduce the number of invocations or show a progress bar to indicate remaining time.

RPC versus document style requests

In order to evaluate RPC versus document style SOAP requests on mobile clients, a scenario where a user have 20 messages which he wants to retrieve on his mobile device is defined.

By considering the retrieval of a single message as a RPC style invocation and the retrieval of 20 messages as a document style invocation, the performed benchmark clearly show how mobile SOAP clients benefits from avoiding pure RPC style services. In the given scenario, retrieving the messages with only RPC style invocations would require 20 invocations. Thus, retrieving the messages would last approximately $20 \times 130ms =$

$2600ms$ on average. In contrast did the document style retrieval of 20 messages last $199ms$ on average.

However, the given scenario can be extended by having the mobile client to regularly check for new messages. In the implemented proof of concept application, this is performed by retrieving all 20 messages for each check, since retrieving all of the user's messages is the only way MPower allows messages to be retrieved. The list of messages is then compared to the list currently shown. A RPC style service would in this case allow the client to check if the list of messages have been changed and only download the new messages, thus dramatically reduce data transfer in the long run.

The given scenario calls for service platforms to support both RPC style and document style Web services. This would enable clients to use document style services typically to retrieve an initial data set and the RPC style services to update it's state according to changes. This combination would help to minimize both processing requirements, thus maximizing battery life, and data transfer.

5.2 Proof of concept invocation performance

5.2.1 Research action, justification and goals

The results from the proof of concept application, clearly demonstrated how a direct invocation architecture can be used on an Android device. The relatively low response times on the previously defined benchmark operations, justified a study on how the performance scales on larger sets of response data.

For this research action, the same proof of concept application will be used to test retrieval of large sets of user messages.

The goal is to identify how the processing duration scales according to the SOAP response size. Additionally, are the measurements comparable with the results from the invocation performance of the same code running on native Java found in section 5.3.4.

5.2.2 Test environment

In order to create controlled responses for the benchmark operations described in the following section, a service mock was set up in SoapUI as in section 4.3.2.

5.2.3 Benchmark description

Benchmark operations

1. Retrieve 100 messages and update the user interface.
2. Retrieve 100 messages without updating the user interface.
3. Retrieve 500 messages and update the user interface.
4. Retrieve 500 messages without updating the user interface.
5. Retrieve 1000 messages and update the user interface.
6. Retrieve 1000 messages without updating the user interface.

Each of the operations defined above will be run on the Nokia N800 device and emulator. The operations will also be run on the emulator with the different network simulation modes, UMTS, EDGE, and GPRS.

Each operation will be performed ten times and minimum, maximum, and average measurements are calculated for each operation and plotted.

Response complexity

In order to relate the results to other systems and SOAP responses, the response complexity from the operations is presented as the size of each response. However, if these results are to be compared with other SOAP responses, an evaluation of how the responses differs in respect to the number of children and siblings in the XML tree, should be taken in account.

The response is shown in listing 5.1. For each message in the response a new `<ns2:reference />` XML element is added to `ActRetrieveMessages`.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:eRetreiveMessagesForPatientResponse xmlns:ns2="urn:hl7-
          org:v3" xmlns:ns3="http://soap.types.security.framework.
          mpower.eu">
         <ActRetreiveMessages>
            <!-- Repeated once for each message -->
            <ns2:reference>
               <ns2:observationEvent>
                  <ns2:id extension="38"/>
                  <ns2:text>Test message for patient</ns2:text>
                  <ns2:activityTime value="2009-05-05 16:22:47.648
                      "/>
               </ns2:observationEvent>
            </ns2:reference>
         </ActRetreiveMessages>
         <Status>
            <messageId>0</messageId>
            <result>0</result>
            <timestamp>1241769279779</timestamp>
         </Status>
      </ns2:eRetreiveMessagesForPatientResponse>
   </S:Body>
</S:Envelope>
```

Listing 5.1: Retreieve messages response example

The size of each response is given in table 5.1 together with the measurement results.

5.2.4 Results

The results from the benchmarks are plotted in figure 5.6, 5.7, 5.8, 5.9, and 5.10. The plots shows the maximum, minimum, and average duration of each benchmark operation listed in section 5.2.3.

The results show that updating the user interface have a much more constant time penalty than actually retrieving and parsing the responses.

Response	Size	Average duration	Throughput (kB/s)
Device			
20 messages	7kB (6725B)	199ms	33.8
100 messages	32kB (31604B)	641ms	49.3
500 messages	156kB (156004B)	2612ms	59.7
1000 messages	312kB (311504B)	5368ms	58.0
Emulator			
20 messages	7kB (6725B)	293ms	23.0
100 messages	32kB (31604B)	537ms	58.9
500 messages	156kB (156004B)	2114ms	73.8
1000 messages	312kB (311504B)	4068ms	76.6
Emulator, UMTS			
20 messages	7kB (6725B)	407ms	16.5
100 messages	32kB (31604B)	887ms	35.6
500 messages	156kB (156004B)	3107ms	50.2
1000 messages	312kB (311504B)	5869ms	53.1
Emulator, EDGE			
20 messages	7kB (6725B)	544ms	12.4
100 messages	32kB (31604B)	1428ms	22.1
500 messages	156kB (156004B)	5928ms	26.3
1000 messages	312kB (311504B)	11370ms	27.4
Emulator, GPRS			
20 messages	7kB (6725B)	836ms	8.0
100 messages	32kB (31604B)	3678ms	8.6
500 messages	156kB (156004B)	16571ms	9.4
1000 messages	312kB (311504B)	32662ms	9.5

Table 5.1: Android's message retrieval performance on large responses.

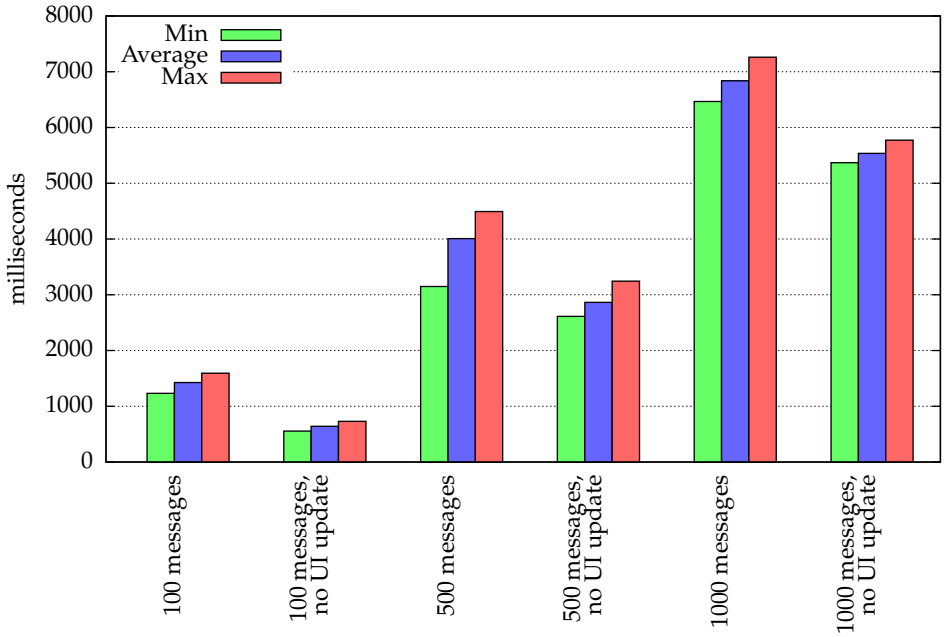

Figure 5.6: Benchmarks running on Nokia N800 with WLAN connection

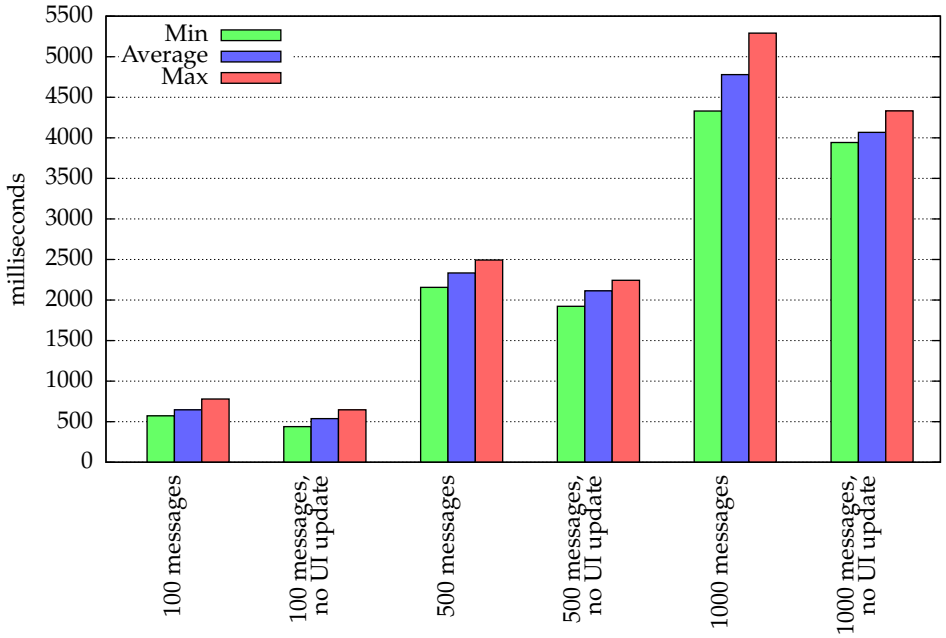

Figure 5.7: Benchmarks running on Android Emulator with LAN connection

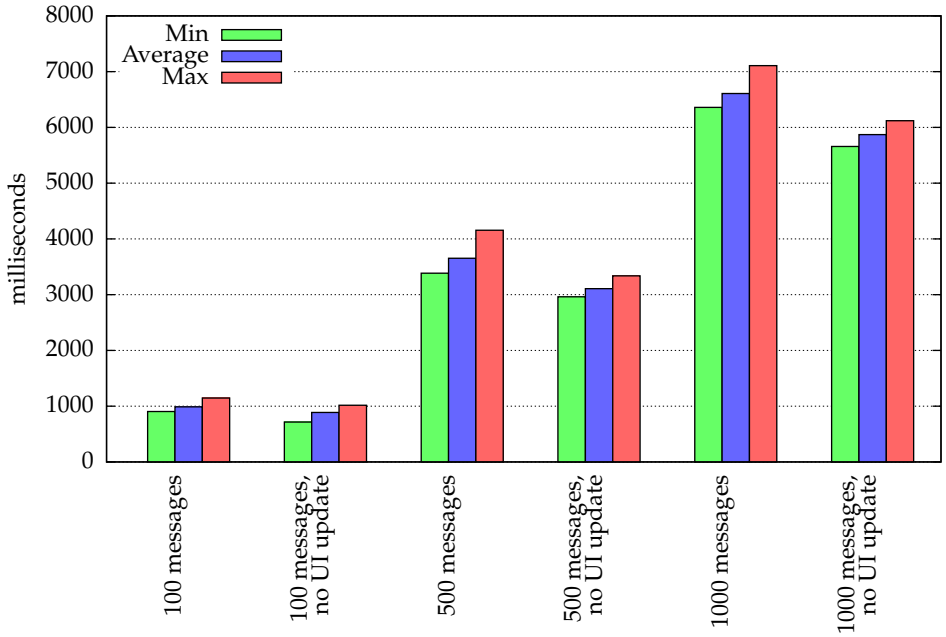

Figure 5.8: Benchmarks running on Android Emulator with UMTS simulation

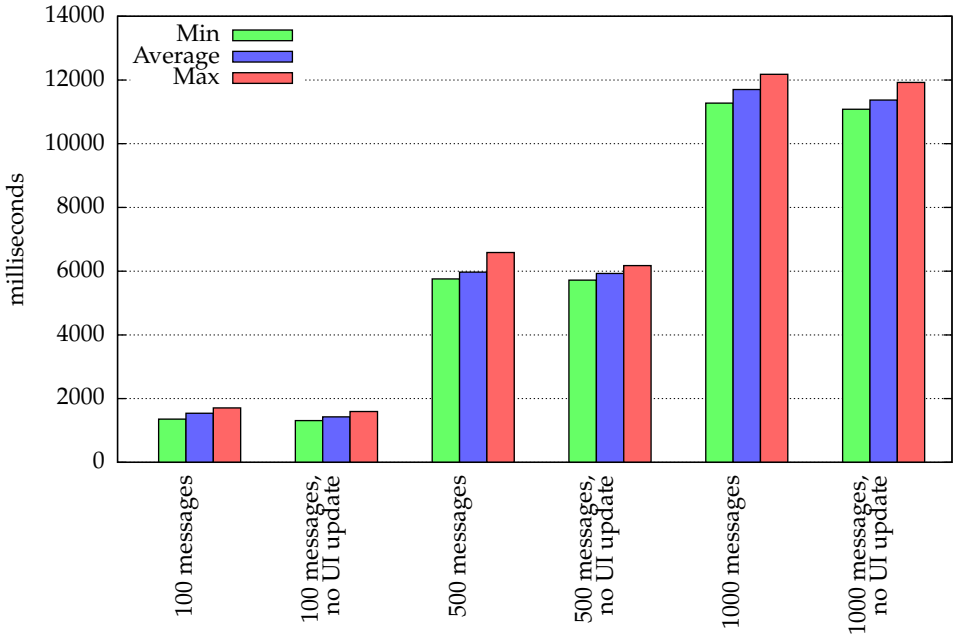

Figure 5.9: Benchmarks running on Android Emulator with EDGE simulation

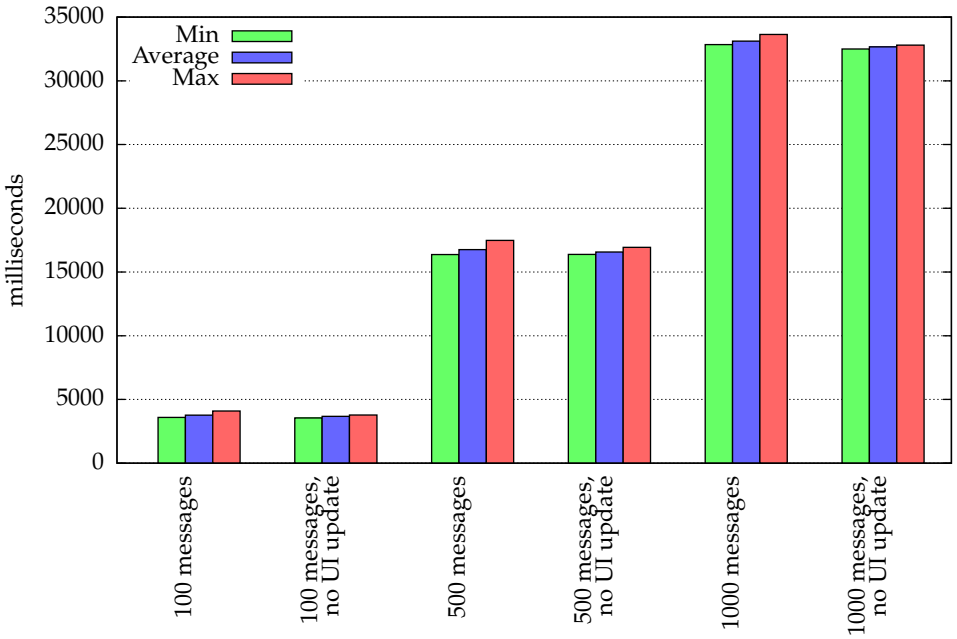

Figure 5.10: Benchmarks running on Android Emulator with GPRS simulation

5.2.5 Evaluation

In a real world application, very few responses would be as large as 500 or 1 000 messages. However, the results shows how Android actually handles such large responses. The given results supports the conclusion in section 5.1.5; data should be transfered in batches and larger responses are preferred over multiple Web service invocations.

As seen in table 5.1, even with UMTS simulation 500 messages were transferred in approximately three seconds. Such reasonable response times, allows mobile clients to prefetch data likely to be requested by the user.

Based on the calculated throughput of 500 and 1 000 messages, the transfer and parsing time seems to scale linearly when the responses are large enough. Constant operations like the creation of the SOAP request and network connection are getting relatively smaller to the entire operation as the response grows. Thus, these constants seems to be negligible on invocations with large responses.

In respect to Nielsen's response time guidelines, response time up to about ten seconds actually will keep the user's attention on the dialog[19]. In respect to this limit, even retrieving 1 000 messages with EDGE network simulation is close to acceptable.

Although most results are satisfiable, it should be noticed how poor the GPRS simulation performed. The throughput of less than 10kB/s would be a major issue for any real world application. However, is this not an Android specific issue and the throughput is actually on line with the theoretical maximum throughput of GPRS networks. It should also be noticed that most mobile networks today is upgraded to support EDGE network speeds or better.

5.3 Android compared to native Java performance

5.3.1 Research action, justification and goals

As a reference, the operations will also be run on a simple native Java client. The native Java client will use the same code as the proof of concept application, except that the user interface is removed.

The goal is to compare how a native Java desktop client would perform compared with the mobile Android application. It also helps to identify if the best practices described in section 5.4.3 are to be treated as Android specific practices or if they also are applicable to native Java clients.

5.3.2 Test environment

For the large service responses with 100, 500, and 1 000 messages, the same test responses are set up using SoapUI as in section 5.2.2. SoapUI is during these tests running on the same machine as the client application, causing no network latency in the results. All other requests will be sent to the MPower test server at Sintef.

The client application is executed on a desktop computer with an Intel Q6600 quad core CPU, 3GB of memory, Ubuntu 9.04, and Sun's Java SE Runtime Environment v1.6.0_13.

5.3.3 Benchmark operations

The basic benchmark operations will be the same as listed in section 5.2.3 and 5.1.3 and each operation will be run ten times.

Additionally will the large performance test operations described in section 5.2.3 be executed in two ways. One where the application is restarted for each of the ten requests and one where the ten requests will be performed in a loop. This goal is to show how native Java Just-In-Time (JIT) compilation[2] affects the performance. JIT is one of the *Java Virtual Machine* features not available on Android, but Dan Morill, Developer Advocate at Google, states that JIT *"is definitely on the Dalvik roadmap"* at the official Android developers forum, although not officially confirmed.

5.3.4 Results

The results from the measurements are plotted in figure 5.11 and 5.12.

Figure 5.11 shows rather large variations in the response times. This is believed to be a result of heavy load variations on the MPower server during the tests. Notice that the average values are much lower than the

[2]Java's Just-In-Time compilation is used to cache machine code representation of an application's byte-code. This compilation and caching is done dynamically at application run-time, but is not yet implemented on Android.

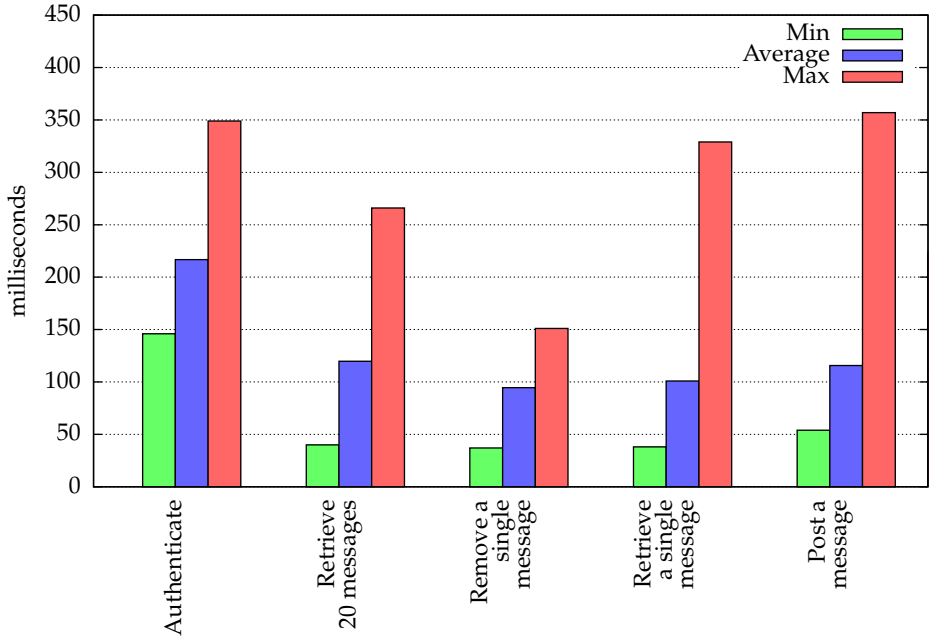

Figure 5.11: Benchmarks running on native Java.

maximum response duration, thus very few of the operations where close to the maximum. Manual tests from SoapUI to the server confirmed the variations in response times on the server. For more accurate results, these tests should be rerun in a more controlled server environment.

Retrieval of 1 000 messages took in average 196ms when JIT is not taken in account which results in a roughly estimated throughput of $312kB \div 0.196s = 1592kB/s$. When JIT is taken in account, and the SOAP requests where performed in a `for loop`, the average throughput was $312kB \div 0.54s = 5778kB/s$.

5.3.5 Evaluation

The results from the native benchmarks reveals how much the available computation resources differs between Android and a desktop computer running native Java. Increasing the response from 500 to 1 000 messages, only increased the average processing time from $158ms$ to $196ms$.

The calculated average throughput is more than 20 times higher on a desktop computer running native Java than the best result achieved on Android. These results where achieved even when JIT was not active and a application startup penalty is included in the native application measures. Sending the ten trail requests in a loop, an by this utilize JIT compilation, increased the throughput more than 3.5 times. Any real world application

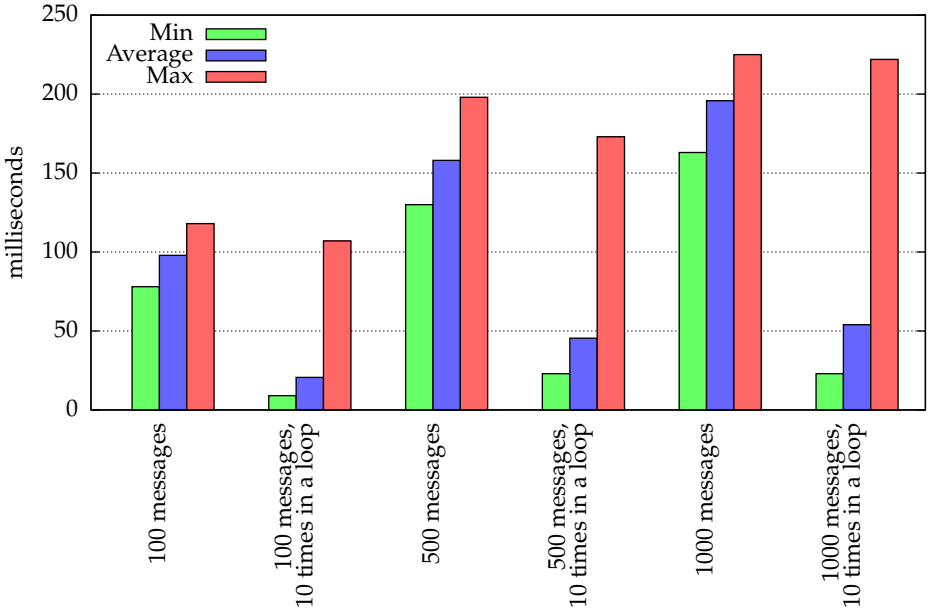

Figure 5.12: Performance benchmarks running on native Java.

would potentially utilize JIT compilation, thus if Android gets JIT support, it will become interesting to see if the same 3.5 times of performance increase is achievable.

By observing the huge differences in throughput on Android and native Java, it seems that native Java applications are not that dependent upon the actual Web service design. Having services responding with a response of 1 000 messages, will have almost no noticeable effect in applications running on native Java. However, an identical response on Android, will take several seconds to parse independently of the available network connection. Although the tests were running without adding any network transfer, the results shows that a native Java client better supports to invoke multiple Web services before presenting results to the user. Because of the available resources, such invocations can be performed in parallel. Android devices would have had problem handling such multiple parallel requests.

From the results, it is noticeable how much faster even small requests runs on native Java than on Android. For example do a request like *Remove a single message* last $95ms$ on native Java, while $195ms$ on the N800 and $316ms$ on the emulator. Additionally, the native Java client executes the fastest operation $23ms$ in contrast to the N800's $121ms$, and the emulator's $176ms$.

5.4 General development experiences

5.4.1 Android development

By having earlier experience both from enterprise Java development and iPhone development, the Android provided software development kit was real easy to getting started with. The provided Eclipse plug in, made starting a new project as easy as starting any other project. Additionally, deploying the application to the Nokia N800 device was as easy as starting the application on the emulator. This included normal debugging tools, which allowed debugging functionality like line stepping.

Android's use of Java as programming language was experienced as a lot easier to get started on than for example iPhone's Objective-C language. Additionally, a feature like Android's automatic garbage collection of objects made the programming experience feel more modern than the iPhone's C based approach.

5.4.2 KSoap2 and Android

The experience with KSoap2 was nothing but positive. The third party library seems not to be actively maintained, but is open source and available for modifications. In conjunction with Android, the KSoap2 worked without modifications by using the Java SE SOAP transport.

When building the proof of concept application, the deserilization mechanisms in KSoap2 were used and they proved to be flexible and well designed. In addition to provide basic serialization support, KSoap2 also allowed optimized parsers and logic to be plugged in where applicable.

5.4.3 Best practices in mobile SOAP clients

Based on the results and evaluations in the previous sections, I here propose a set of design principles for accessing Web services directly from Android with SOAP messaging. These principles are basic guidelines which affects both the client application in itself and the actual Web services' design on the server.

- Updates of the user interface should be minimized when possible.
- Data should be fetched in large batches, instead of multiple small requests.
- Data transfer should be minimized. When possible, only check for data changes and retrieve the changes instead of complete data sets.
- Minimize the number of service invocation necessary to display the result.

- Design for a slow server response and show progress dialogs at least when more than one SOAP request is issued or where a large response is expected.

Chapter 6

Conclusion

6.1 Contributions

This thesis contributes to identifying impacts of architectural decisions, and specifically evaluates challenges and possibilities in using Android in conjunction with a SOAP based service oriented architecture.

Further, this study contributes with a proof of concept application which shows that direct invocation of Web services with SOAP messages is possible on Android. Thus, the research shows how the Android platform can use an existing SOA infrastructure, like MPower, to create new and innovative applications.

By documenting performance measures of direct SOAP Web service invocations on Android, the research helps designing applications and Web services for optimal client responsiveness.

Additionally, this research contributes to the MPower project with documentation on how to access SOAP Web services and the proof of concept application source code which is available at `http://knutseninfo.no/mpower/android_poca.zip`.

6.2 Conclusion

Mobile device processing capabilities have increased remarkably through the latest years. This makes it possible to build more complex applications targeted for mobile devices. This study and its results, shows how architectures and systems mostly designed for desktop usage like Web service invocation with SOAP messaging, now also is possible to be used on mobile platforms like Android.

With the help from faster and more available mobile networks, accessing Web services directly with SOAP messaging is definitely possible on Android. However, the high network availability on such mobile devices, also makes an architectural alternative like a HTML frontend increasingly competitive. This is strengthen by the increasing number of mobile platforms application developers must support and by observing the trend on desktop computers where web-based applications like *Google Docs*[1] have become a strong alternative to native desktop applications.

The following sections presents the results and conclusions of the research questions given in section 1.1.

[1]Google Docs is a free web-based office suite featuring a word processor, spreadsheet, presentation, and form application available at `https://docs.google.com`.

6.2.1 Which architectural alternatives exists for using SOAP based Web services on Android, and how do the architectural choice affect the client application?

Three main architectural alternatives was found to support mobile SOAP clients on Android. The web application serving a HTML frontend, direct invocation, and gateway architectures were all found to have their strengths and weaknesses.

In respect to the gateway architecture, the performance results achieved with the direct invocation alternative, gives rise to the question whether the work of building a gateway architecture is justifiable in any project or if most of the gateway advantages can be achieved by designing the Web services specifically for mobile usage. Thus, the HTML frontend architecture and direct invocation remained as the two most interesting alternatives.

While the HTML frontend architecture utilizes already well known techniques of building SOAP capable clients, the direct invocation represented an architecture more dependent on the mobile device the client is running on both with concern to hardware resources and the available programming environment.

6.2.2 Is it possible to directly invoke SOAP Web services on Android, and will such invocation be effective enough?

Although the proof of concept application never was run on a device designed for Android, the benchmark results shows how well an SOAP application actually might work. Typical client operations, performed on an EDGE simulated network or faster, are all well within Jacob Nielsen's one second limit of response[19]. Based on the request and response complexity of the messages measured, these results are believed to be representable also for other client operations in a real world application.

Running the application on an Android device like HTC Dream, is believed to further improve the invocation performance. Especially since the performance seemed to be limited by the CPU resources and the HTC Dream has a faster CPU than the Nokia N800 used for testing.

While Android is proved to be usable for hosting SOAP client applications, the process of developing such applications are found to quickly become a tedious process of manual coding SOAP serialization and deserialization classes. Mainly because KSoap2 lacks any code generation utilities, but also because of MPower's complex WSDL files and the responses received from the MPower platform not necessarily validated against the provided WSDL. Notice however that MPower still is a development project not ready for production.

6.2.3 How can the design of SOAP Web services be optimized for use on mobile devices running Android?

Benchmark results from the proof of concept application indicates that the number of subsequent SOAP requests should be limited to a minimum, especially as network latency increases and transfer capacity decreases. This requires the Web services to be designed in such manner that the clients directly may obtain required data.

In respect to the development process of Android based SOAP clients, it is experienced that by limiting the depth of the SOAP messages XML structure, the parsing complexity on the client is reduced. Also by serving content easily interpreted as simple types like strings and integers instead of complex classes, removes the need of creating a deep parsing structure.

In a service oriented architecture, this kind of Web service specializations are typically easy to accomplish by creating intermediary and coordinator services which enables results from multiple services to be directly returned to the client. Such a design, would create a tighter coupling between the client application user interface and the services, just as by using a gateway architecture. However, in contrast to a gateway architecture, creation of client specific Web services is possible by only using the tools and server architecture already in use. Thus such customizations will introduce a minimal amount of overhead in development of the service oriented architecture platform.

6.3 Further work

It would be useful to test the developed proof of concept application on an actual Android device such as the HTC Dream. The same benchmarks described in this thesis, are repeatable and the results can be compared. Such a comparison enables the architectural choices and Web service design practices to be revised based upon more accurate results.

Further, a study on how SOAP invocations' affect battery lifetime will be valuable in order to evaluate how applications can be used in real world situations.

During this study, a set of guidelines for accessing Web services directly from Android was proposed in section 5.4.3. An analysis on how the MPower platform supports these guidelines, might help to identify how available Web services can be changed in order to better support direct invocation from mobile devices. Such a study might also be applicable to other SOA platforms, making it a valuable research contribution.

Bibliography

[1] NITdroid project website. http://code.google.com/p/nitdroid. Collected 2009-05-19.

[2] Amit Asaravala. Giving SOAP a REST. http://www.devx.com/DevX/Article/8155. Collected 2009-03-27.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, april 2007.

[4] Jon Box and Dan Fox. *Building solutions with the Microsoft .NET Compact Framework*. Addison-Wesley, 2003.

[5] Roberto Chinnici, Hugo Haas, Amelia A. Lewis, Jean-Jacques Moreau, David Orchard, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts. http://www.w3.org/TR/wsdl20-adjuncts. Collected 2009-04-09.

[6] Douglas Crockford. Introducing JSONThe application/json Media Type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627.txt. Collected 2009-04-09.

[7] Thomas Erl. *SOA: Principles of Service Design*. Prentice Hall, jun 2007.

[8] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[9] Apache Software Foundation. ADB Tweaking Guide. http://ws.apache.org/axis2/1_3/adb/adb-tweaking.html. Collected 2009-03-31.

[10] Alan R. Hevner, Salvatore T. March, and Jinsoo Park. Design science in information systems research. *MIS Quarterly*, 28:75–105, 2004.

[11] SINTEF ICT. MPOWER objectives. http://www.sintef.no/Projectweb/MPOWER/The-Project/Objectives. Collected 2009-03-23.

[12] SINTEF ICT. MPOWER project. `http://www.sintef.no/Projectweb/MPOWER/The-Project`. Collected 2009-03-23.

[13] Google Inc. Android developer's guide - What is Android. `http://developer.android.com/guide/basics/what-is-android.html`. Collected 2009-03-18.

[14] Sun Microsystems Inc. JSR-172 J2ME Web Services Specification. `http://jcp.org/aboutJava/communityprocess/review/jsr172`. Collected 2009-05-23.

[15] Ramarao Kanneganti and Prasad Chodavarapu. *SOA Security*. Manning Publications, jan 2008.

[16] Tomas Kozel and Antonin Slaby. Mobile access into information systems. In *Proceedings of the ITI 2008 30 Int. Conf. on Information Technology Interfaces*, pages 851–856, 2008.

[17] Simon St. Laurent and Michael Fitzgerald. *XML Pocket Reference*. O'Reilly Media, Inc, 3rd edition, aug 2008.

[18] Luqun Li, Minglu Li, and Xianguo Cui. The study on mobile phone-oriented application integration technology of web services. In *GCC (1)*, pages 867–874, 2003.

[19] Jacob Nielsen. Response Times: The Three Important Limits. `http://www.useit.com/papers/responsetime.html`. Collected 2009-05-05.

[20] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc, may 2007.

[21] Mahadev Satyanarayanan. Fundamental challenges in mobile computing. In *Symosium on Principles of Distributed Computing*, pages 1–7, 1996.

[22] The Internet Society. Hypertext Transfer Protocol – HTTP/1.1. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`. Collected 2009-04-09.

Appendices

Appendix A

Android API vs Java API

The list is based on `http://blogs.zdnet.com/Burnette/?p=504` and updated to match Android 1.1 r1 API.

A.1 Supported Java 2 Platform Standard Edition 5.0 API packages

java.io File and stream I/O

java.lang (except java.lang.management) Language and exception support

java.math Big numbers, rounding, precision

java.net Network I/O, URLs, sockets

java.nio File and channel I/O

java.security Authorization, certificates, public keys

java.sql Database interfaces

java.text Formatting, natural language, collation

java.util (including java.util.concurrent) Lists, maps, sets, arrays, collections

javax.crypto Ciphers, public keys

javax.net Socket factories, SSL

javax.security (except javax.security.auth.kerberos, javax.security.auth.spi, and javax.security.sasl) Security

javax.sound Music and sound effects

javax.sql (except javax.sql.rowset) More database interfaces

javax.xml.parsers XML parsing

org.w3c.dom (but not sub-packages) DOM nodes and elements

org.xml.sax Simple API for XML

A.2 Unsupported Java 2 Platform Standard Edition 5.0 API packages

- java.applet
- java.awt
- java.beans
- java.lang.management
- java.rmi
- javax.accessibility
- javax.activity
- javax.imageio
- javax.management
- javax.naming
- javax.print
- javax.rmi
- javax.security.auth.kerberos
- javax.security.auth.spi
- javax.security.sasl
- javax.swing
- javax.transaction
- javax.xml (except javax.xml.parsers)
- org.ietf.*
- org.omg.*
- org.w3c.dom.* (sub-packages)

A.3 Included third party libraries

junit.framework JUnit unit test framework

org.apache.http HTTP authentication, cookies, methods, and protocol

org.json JavaScript Object Notation

org.w3c.dom Provides the official W3C Java bindings for the Document Object Model, level 2 core.

org.xml.sax Core SAX APIs.

org.xml.sax.ext Interfaces to SAX2 facilities that conformant SAX drivers won't necessarily support.

org.xml.sax.helpers "Helper" classes, including support for bootstrapping SAX-based applications.

org.xmlpull.v1

org.xmlpull.v1.sax2

Appendix B

Android SDK tools

Collected from `http://developer.android.com/guide/developing/tools/index.html` and describes the Android SDK v1.1 r1.

Android Emulator A virtual mobile device that runs on your computer. You use the emulator to design, debug, and test your applications in an actual Android run-time environment.

Android Development Tools Plugin (for the Eclipse IDE) The ADT plugin adds powerful extensions to the Eclipse integrated environment, making creating and debugging your Android applications easier and faster. If you use Eclipse, the ADT plugin gives you an incredible boost in developing Android applications.

Hierarchy Viewer The Hierarchy Viewer tool allows you to debug and optimize your user interface. It provides a visual representation of your layout's hierarchy of Views and a magnified inspector of the current display with a pixel grid, so you can get your layout just right.

Draw 9-patch The Draw 9-patch tool allows you to easily create a NinePatch graphic using a WYSIWYG editor. It also previews stretched versions of the image, and highlights the area in which content is allowed.

Dalvik Debug Monitor Service (ddms) Integrated with Dalvik, the Android platform's custom VM, this tool lets you manage processes on an emulator or device and assists in debugging. You can use it to kill processes, select a specific process to debug, generate trace data, view heap and thread information, take screenshots of the emulator or device, and more.

Android Debug Bridge (adb) The adb tool lets you install your application's .apk files on an emulator or device and access the emulator or device from a command line. You can also use it to link a standard

debugger to application code running on an Android emulator or device.

Android Asset Packaging Tool (aapt) The aapt tool lets you create .apk files containing the binaries and resources of Android applications.

Android Interface Description Language (aidl) Lets you generate code for an interprocess interface, such as what a service might use.

sqlite3 Included as a convenience, this tool lets you access the SQLite data files created and used by Android applications.

Traceview This tool produces graphical analysis views of trace log data that you can generate from your Android application.

mksdcard Helps you create a disk image that you can use with the emulator, to simulate the presence of an external storage card (such as an SD card).

dx The dx tool rewrites .class bytecode into Android bytecode (stored in .dex files.)

UI/Application Exerciser Monkey The Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system- level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner.

activitycreator A script that generates Ant build files that you can use to compile your Android applications. If you are developing on Eclipse with the ADT plugin, you won't need to use this script.

Appendix C

Proof of concept application screenshots

Screenshots from the proof of concept application running on Android which accessses the MPower Messageboard Web service with SOAP messaging is shown in figure C.1, C.2, C.3, C.4, and C.5.

The dropdown box shown on the screenshots, is used for switching between different Web service end-points during testing.

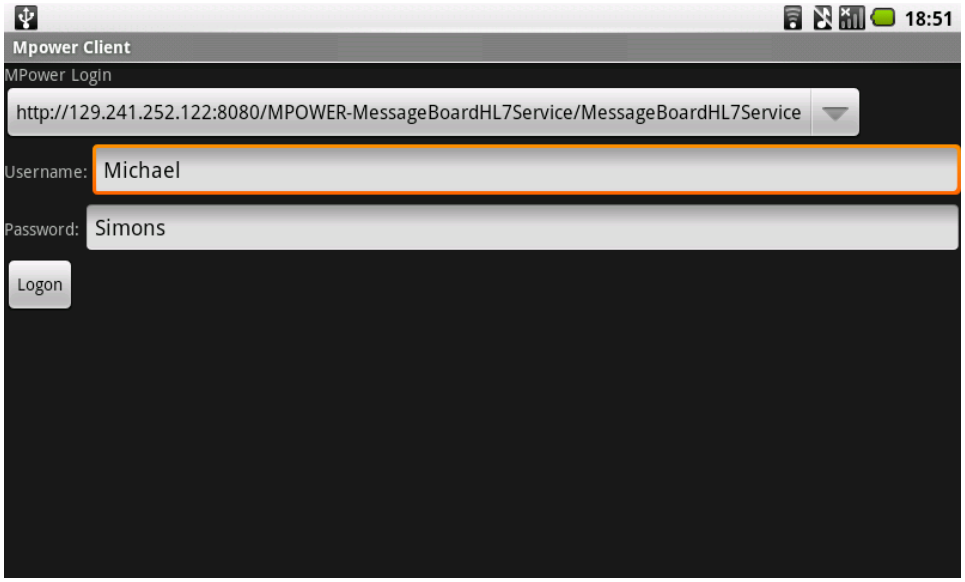

Figure C.1: Authentication screen.

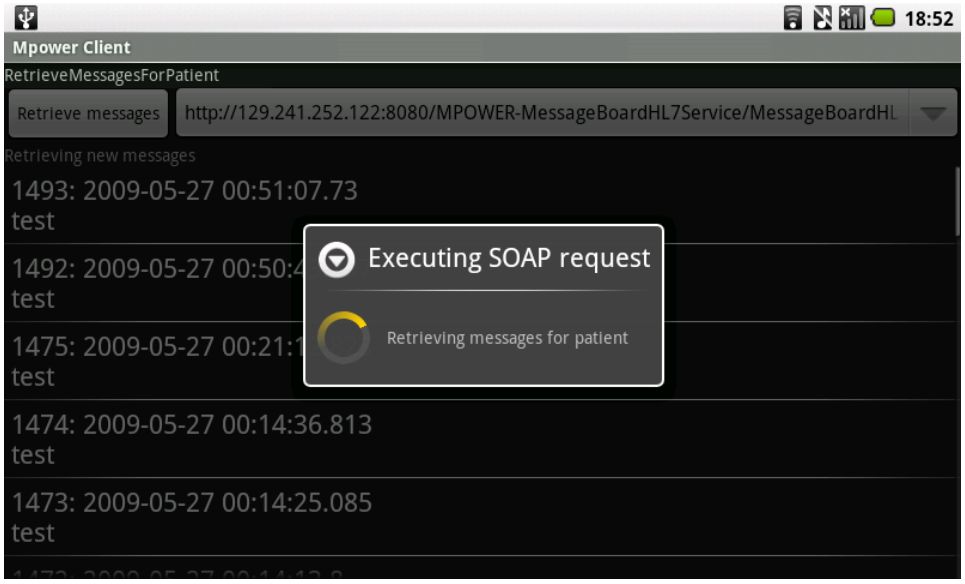

Figure C.2: Progress dialog while loading messages.

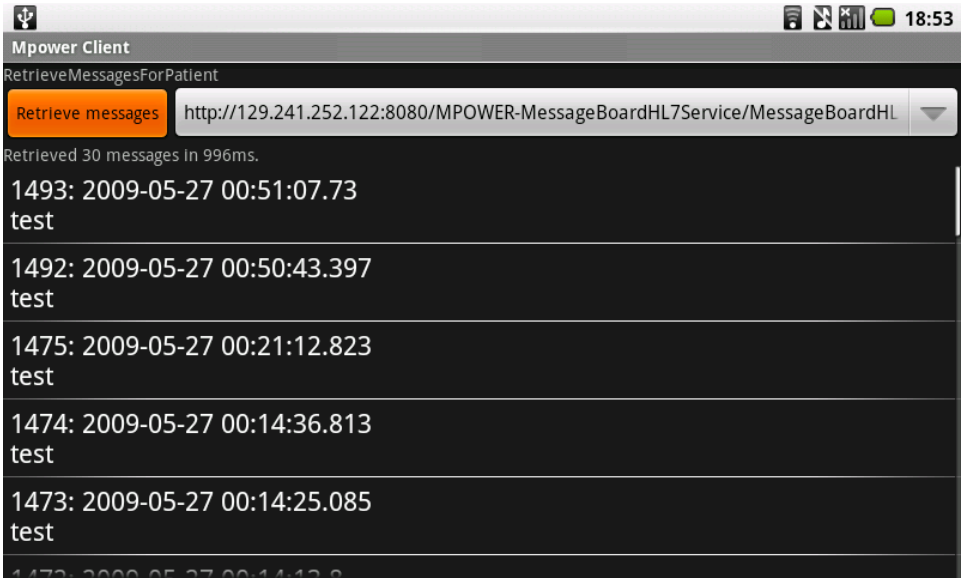

Figure C.3: List of messages retrieved from MPower. Available when authenticated as a patient.

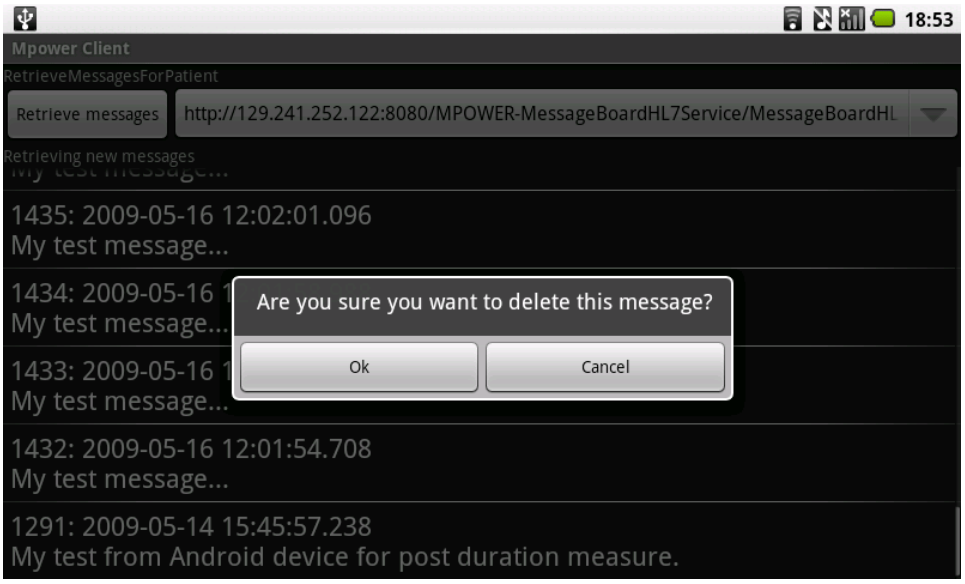

Figure C.4: Confirmation dialog when deleting a message.

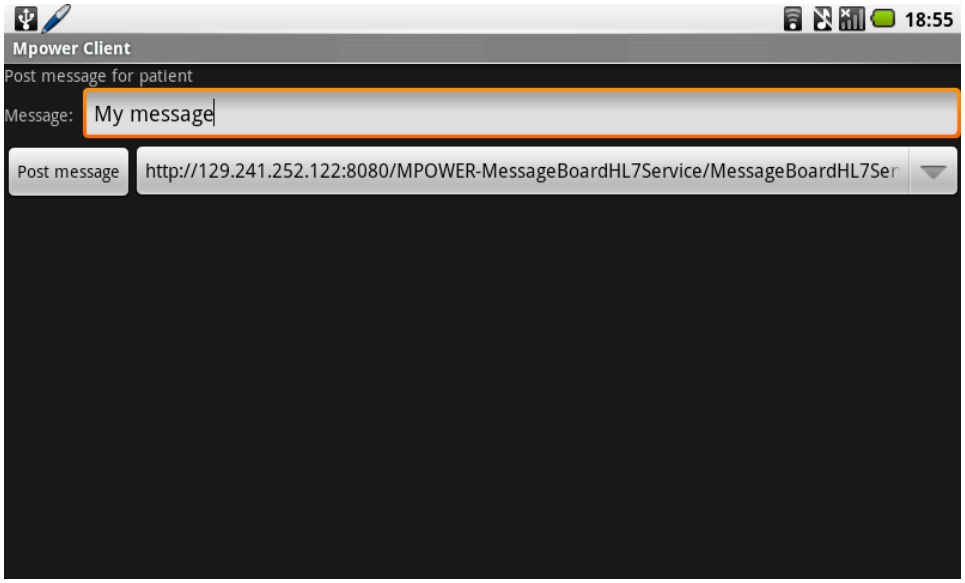

Figure C.5: Screen for posting messages. Available when authenticated as a doctor.