Realization and Performance Analysis of a SOAP Server for Mobile Devices

Linh Pham, Guido Gehlen

RWTH Aachen University, Chair of Communication Networks
Kopernikusstr. 16, Aachen 52074
e-mail: {lph,guge}@comnets.rwth-aachen.de

Abstract: **A middleware platform for mobile devices to efficiently build distributed applications in a heterogenous mobile environment is essential. Especially for mobile Peer-to-Peer (P2P) applications and applications which have to call back a mobile device, server functionalities for mobile devices are required.**

In this paper the server building block of our mobile Web Service based middleware is introduced. This server is integrated in the wireless Simple Object Access Protocol (SOAP) in terms of a HTTP server binding to SOAP. Thus, the mobile application can define services which will be published and used by other devices.

This paper presents a lightweight SOAP server architecture for Java2 Micro Edition (J2ME) devices as well as the implementation in Java. Finally, the results of a performance analysis are given in consideration of criteria like latency and memory usage.

1. Introduction

Wireless computing devices increasingly appear in our environment. So far, the interaction among mobile devices as well as the interaction between mobile devices and a fixed infrastructure is complicated to realize. To simplify the application development on top of distributed systems, a common layer (middleware) has to be introduced.

We have developed a mobile middleware [4] based on Web Service technologies. Using this middleware, mobile devices will be a part of the Web Services located in the Internet. This middleware enables not only the access of Internet Web Services, but also the provision of services. These services can be used either from Internet applications (mobile call back services) or from any other devices, fixed or mobile (P2P services) [5].

The work as been motivated from research projects dealing with mobile applications, especially in the logistics and vehicular sector, like INVENT-VMTL project[1] and IST-MYCAREVENT[2]. Moreover, the development of consumer applications will benefit from such a middleware.

Before the server building block will be described, a short introduction to the Service Oriented Architecture (SOA) and the Web Service technologies is given. A short overview of the mobile middleware and the placement of the server parts is presented. After the detailed description of the wireless SOAP server, results of a performance analysis are presented.

[1]Traffic management in transport and logistics (VMTL) is one project in the research initiative INVENT founded by the German government

[2]MYCAREVENT is a European research project dealing car breakdowns and roadside assistant services

2. Services Oriented Architecture and Web Services

The role of the SOA is going to be more and more popular in the software design for distributed systems. This middleware framework tries to simplify as much as possible the software architecture of distributed applications. The SOA is an architectural style to achieve loose coupling among interacting software agents realizing a service. Web Services are one realization of the SOA using open and platform independent Extensible Markup Language (XML) based protocols and data descriptions. Therefore, the Web Service middleware framework is also applicable for mobile environments.

The SOA defines three roles, a Service-Requestor (R), Service-Provider (P), and a Service-Broker. A software agent interacting with other software agents plays one or more roles. They communicate in the way as depicted in figure 1.

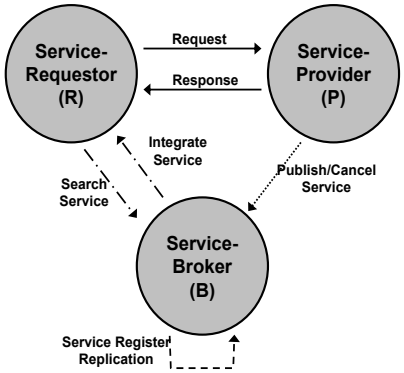

Figure 1: The Service Oriented Architecture

Providers publish their services to a service registry (service-broker). More than one service-broker within the environment have to replicate their service registries. Requestors use the Broker to search for services and integrate them by accessing the service description. This description include all information needed to access the service and is used to generate a proxy object. The proxy object represents the remote service and bridges the native messaging inside the client environment to the platform independent messaging in the SOA environment. The process of proxy object generation maps the platform independent description into a real software object for the client's system environment.

To achieve high interoperability, all SOA entities have to use a common language for service description, messaging and service registration. The XML is such an appropriate common language. On the basis of XML

the W3C specifies a middleware framework, called Web Services, following the SOA. For messaging the SOAP [6, 1] is used. It is based on standard internet protocols like HTTP or other arbitrary transport protocol like WAP [3] or Block Extensible Exchange Protocol (BEEP). The SOAP envelope is structured in XML. Interfaces are described in an XML subset, the so called Web Service Description Language (WSDL) [2]. This description includes all the information needed in order to invoke the service methods from other nodes.

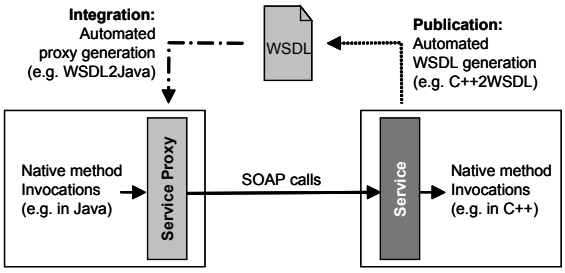

Figure 2: Publication and Integration of Web Services using the WSDL description

The WSDL description can be automatically generated from the service class at the provider node. Another device can use this service by generating a service-proxy object, see figure 2. After the publishing and integration phase, the application can call local methods of the service-proxy which will be transformed in remote SOAP calls. Thus, the service and service-proxy objects bridge the platform dependent method calls into a common platform independent language (XML).

3. HTTP-SOAP Binding

As mentioned in the previous section, SOAP can be coupled to an arbitrary session or transport protocol, but most of all Hypertext Transport Protocol (HTTP) is used. Our implementation for mobile devices is based on the open source packages kSOAP and kXML. At default, kSOAP only supports a client HTTP-Binding, but to publish services from the device to other devices, a server HTTP-Binding is necessary.

In figure 3 the relevant protocol stacks on one mobile device are depicted. We are considering an all Internet Protocol (IP) infrastructure. Thus, the lowest layer is IP. Inside the mobile middleware is the SOAP layer which couples to an arbitrary underlying protocol like session protocol or transport protocol. In this paper we look at the HTTP Binding. HTTP defines fixed client and server roles. Within one HTTP session, one device takes the client role while another device takes the server role. In our middleware, kSOAP has been extended so that each device can take both roles. There is an HTTP server listening on the HTTP server socket for incoming requests. The requests containing SOAP content are forwarded to HTTP-server Binding. Requests without SOAP content may access static resource, like HTML pages and files.

Services will be coupled to the HTTP-Server Binding to SOAP. The incoming SOAP requests are matched

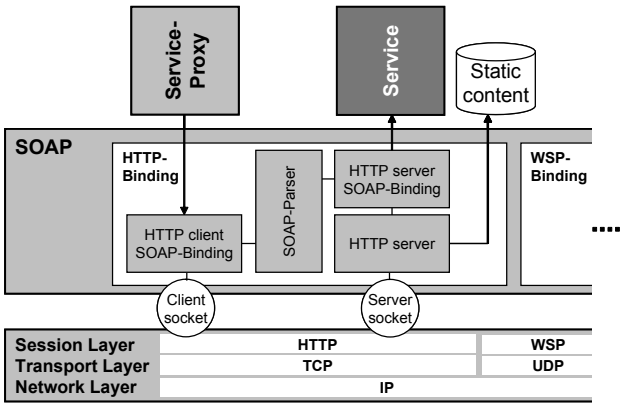

Figure 3: The HTTP Binding

within this Binding to method invocations of the corresponding services. The realization will be introduced in the following sections.

The Service-Proxy is coupled to the client HTTP Binding for SOAP. All local method calls to this proxy will be mapped into remote SOAP calls.

4. SOAP-Server Implementation

This section introduces a design and implementation of a light-weight SOAP server which is used to deploy Web Services on mobile devices with the J2ME platform. In addition, the performance characteristics of the light-weight SOAP server are described. The SOAP server currently makes services available via a HTTP interface, but also further transport bindings are imaginable.

Figure 4 shows the core structure of the light-weight SOAP server.

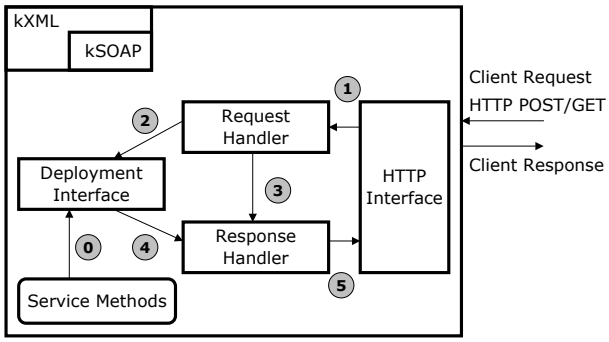

Figure 4: SOAP Server Core Structure

When a client connects to the SOAP server for sending a HTTP POST/GET request, the server socket accepts the client connection and returns a socket containing an input data stream. From this input data stream, a HTTP request message is extracted and given to the *Request Handler* (point 1 in figure 4) for further processing.

The *Request Handler* makes use of kSOAP and kXML to process the request message. After de-

serializing XML data structures to general Java objects, the *Request Handler* will pass these objects and necessary information (i.e: required parameters to invoke service methods like requested service method name, information about type and name of the general Java objects) to the *Deployment Interface* (point 2 in figure 4). Then, the *Request Handler* will call the *Response Handler* to handle a response to the client (point 3 in figure 4).

The *Response Handler* gets a result from the *Deployment Interface* (point 4 in figure 4) and sends the result to the client through an output data stream corresponding to the client connection socket. Depending on the type of result (a responded SOAP message or a static resource such as a file), the *Response Handler* will invoke a relevant method.

In order to serve the client requests, service methods that were initiated at the start-up phase of the SOAP server (point 0 in figure 4) communicate with the *Deployment Interface*.

4.1. De-serialization of XML Data into Java Data Objects

An important requirement for the SOAP server is that it must be capable of de-serializing any user-defined data types. In order to do this task, the SOAP server has to receive necessary information about the data types from developers who want to deploy services on the SOAP server. The kSOAP has a mapping method to map an XML data structure to a user-defined Java object. However, when de-serializing the XML data structure, only a general Java object (*Object* type) is retrieved no matter what data type the XML data structure is intended for. Hence, the SOAP server, at runtime, has to do the task of type-casting the de-serialized general Java object into its real type for further processing.

The figure 5 describes an example of the above situation. An XML data structure named *DataStruct* is at the input of the *Request Handler* module for being de-serialized. This XML data structure represents an instance of the user-defined Java class named *DataStructObject*. A *DataStructObject* object is defined with three data members that are *varInt* of Integer type, *varFloat* of Float type, and *varString* of String type.

The *Request Handler* de-serializes the XML data structure *DataStruct* into a general Java object. This general Java object contains information about the name and the real type of the expected Java object (*DataStructObject*). The above information can be extracted from the general Java object.

The SOAP server now has to type-cast this general Java object into its real type (the user-defined *DataStructObject* type) in order to access its data members.

Although the SOAP server can extract the information about the type of the expected Java object (an instance of the *DataStructObject* class), it can not do type-casting on run-time. This means we have to clarify type-cast task before compiling. In other words, the source code of the SOAP Server has to be changed each time users want to deploy new services with their own defined data types. In order to solve this problem, the SOAP server uses the *Deployment Interface* which can be thought as a bridge between the SOAP Server and the Business Logic module.

4.2. Deployment Interface

The *Deployment Interface*, as shown in figure 6, contains a static instance of the Hashtable class. This Hashtable is used to map a service method name to a service method object which has been deployed before.

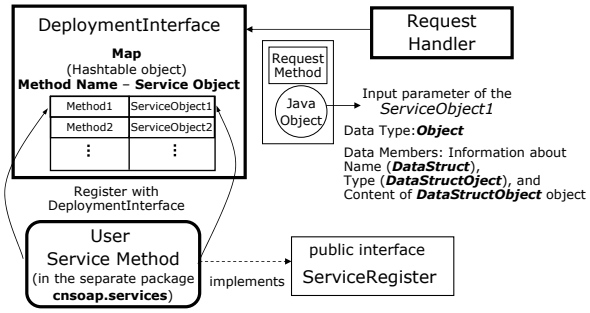

Figure 6: Structure of *Deployment Interface*

All service methods register their names and themselves (as instances of their classes) with the *Deployment Interface* once at the start-up phase of the SOAP server. These *Method Name - Service Object* pairs are stored in the Hashtable. It is noticed that the Hashtable variable was declared static, hence it is independent from instances of the *Deployment Interface* class. In other words, there is only a unique Hashtable for all instances of the *Deployment Interface* class. This aims at a memory saving purpose when increasing number of services.

After de-serializing the XML structure into the general Java object as described in 4.1., the *Request Handler* passes this general Java object, along with a client requested *Method Name* (which was retrieved from the XML structure by the *Request Handler*), to the *Deployment Interface*.

The *Deployment Interface* then will search the Hashtable for the requested *Method Name* and will invoke a corresponding *Service Object*. This *Service Object* is retrieved from the Hashtable as a Java object of the *Object* type. Hence, this invoking task can be done by asking all service methods to implement a public interface named *Service Register* which has the *invoke* method. Then all the *Service Object*s will be type-casted into *Service Register* to call the *invoke* method.

The *invoke* method mentioned above has three input parameters that must be known by the user for deploying services. The *DeserializedObject* is the general Java object passed from the *Request Handler*. The two String parameters *ObjectName* and *ObjectType* represent the name and the type of the *DeserializedObject* object.

When being invoked with the input parameters, the service method type-casts the *DeserializedObject* object into an object of its real type (in the above example, it is the *DataStructOject* type). By implementing this solution, the SOAP server code is separated from the Business Logic code.

A *Method Name - Service Object* pair which is stored in the Hashtable (figure 6) can be extended into a pair

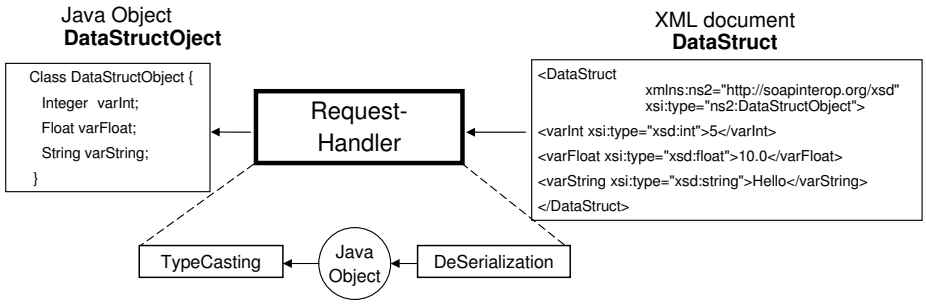

Figure 5: Type Casting of XML Data into Java object

of *Method Name - Services Group Object* when users want to deploy a group of several services under one name of the services group. In this case, the *Services Group Object* will be a Hashtable object containing pairs of *Method Name - Service Object*.

5. Performance Analysis

The light-weight SOAP server is implemented to run on J2ME CDC/PersonalJava, CDLC/MIDP and J2SE platform. It requires a small memory usage and is capable of handling concurrent client requests.

5.1. Strategies for concurrent request handling

Under limitation of J2ME API (Application Programming Interface), 2 strategies used to handle concurrent threads are implemented. These strategies are also used to limit a maximum number of running threads.

With the first strategy, the algorithm is summarized as following (figure 7a):

1. Start generating n threads to handle n client requests in parallel (n equals to 4 in this case).

2. Wait until the last thread (the n^{th} thread) finishes. No thread is created during this interval.

3. Start again generating n other threads and repeat the step 2.

The figure 7a explains how a maximum number of running threads is limited. In the graph, the vertical axis describes a life-cycle of a thread. The horizontal axis represents a thread index. The value of ϵ, which is an interval between two subsequent threads, is considered very small.

A group of only n (set to 4 in the figure 7a) threads are created continuously to process n client requests in parallel. The life-cycle of each thread can be different. If the threads finish in orders which they have been created (it means that first-created, first-finishing), the thread n will finish finally at the time t_n after a life-cycle of T_n. After the thread n has finished, a group of other four threads can be generated, starting at the index n+1. During the interval T_n (the life-cycle of the thread n), no thread is created.

It has been assumed so far that the threads finish in order. No thread in a group of n threads exists after the final thread of this group has finished. In other words, the number of running threads will not exceed the value n. In this case, n is the maximum number of running threads.

Now, the worst case is supposed that the thread n in a group of n threads always has the shortest life-cycle. In addition, all (n-1) previous threads have the longest life-cycle (the shortest and longest life-cycle of threads can be determined by measuring experimentally the time required to complete services). After the thread n has finished, another group of n threads is created, starting at the index n+1, while the first (n-1) threads of the previous group are still running. Hence, the number of running threads exceeds n.

Taking a reference to the graph in the figure, the thread 4 (n=4) in the group 1 has the shortest life-cycle while all previous three threads have the longest life-cycle. When the thread 4 finishes, the group 2 of four threads is created while the first three threads in the group 1 are still running.

As supposed, the final thread in group 2 (index=8) also has the shortest life-cycle. When it finishes, the group 3 of four threads is created while the first three threads in the group 2 are still running.

It is assumed that the first three threads in the group 1 now finish at the time t_N while the three threads in the group 2 and all the threads in the group 3 are still running. Before the first three threads in the group 1 finish, the number of running threads reaches the maximum number. The maximum number is a sum of the first three threads in the group 1, the first three threads in the group 2 and all of four threads in the group 3. After the time t_N, the number of running threads reduces by 3 (the first three threads in group 1).

A formula determining the number of running threads at the time t_N, when the maximum number of running threads is reached, is derived as following:

T_{max} is called the longest life-cycle that the first three threads in the group 1 have. T_{min} is called the shortest life-cycle that the final thread in each group has. The k coefficient is defined as followed:

$$k = \left\lceil \frac{T_{max}}{T_{min}} \right\rceil, \textit{ so that } \quad k - 1 < \frac{T_{max}}{T_{min}} \leq k \qquad (1)$$

The number of running threads at the time t_N:

$$k(n-1) + 1 \qquad (2)$$

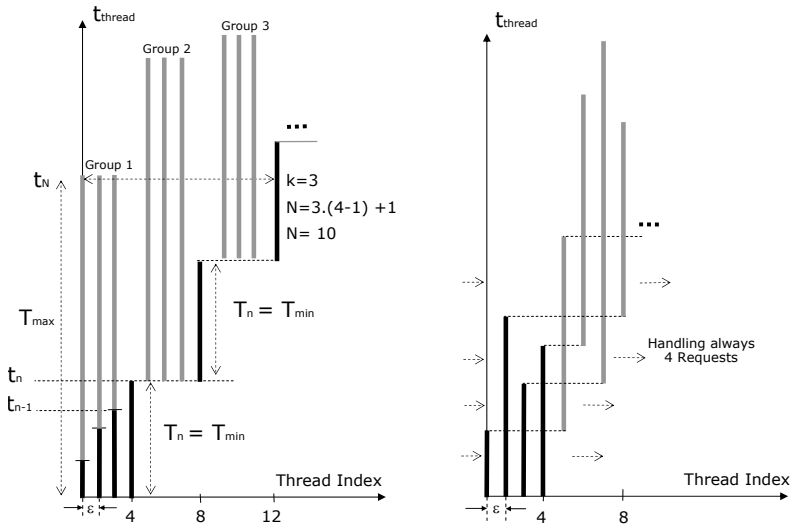

Figure 7: Concurrent Request Handling: a (left part)- Strategy 1 (the worst case scenario depicted); b (right part)- Strategy 2

If N is called the allowed maximum number of running threads that we expect, the below condition for n can be written:

$$k(n-1)+1 \leq N \Rightarrow n \leq \frac{N-1}{k}+1 \qquad (3)$$

The value k, according to the figure 7a, has the value of 3. If N is limited to 10, from equation 3, we have $n \leq 4$. In other way around, if n is chosen with 4 as in the figure 7a, the number of running threads at the time t_N can be be calculated from equation 2: $k(n-1)+1 = 10$.

The equation 3 can be used to limit the maximum number of running threads. Along with the chosen value of N, the maximum queue length of the Java server socket must also be considered. These two parameters decide how many concurrent incoming client requests can be handled by the SOAP server.

This strategy has its pros and cons. The pros is that it is very simple to implement. In addition, there are only few times that the SOAP server has to handle the client requests in parallel. Most of the time, the SOAP server only processes the client requests one by one. Therefore, this strategy is also an option for handling concurrent requests. The cons is that device's resources were not used efficiently because the device has to wait for all of n threads to be finished. Then, the device starts generating n other threads.

The second strategy is an implementation of a thread pool. The thread pool is given some fixed number of threads (an array of n_p worker threads, $n_p = 4$ in the figure 7b) to use. The thread pool will assign a task (processing a client request) to each of these n_p worker threads. When any of these threads finish its old task, a new task is assigned immediately. If the number of concurrent client requests is large than n_p, extra client requests are put in a queue for later processing by any free worker threads.

In comparison with the first strategy, this strategy uses device's resource more efficiently because it allows a fix number of worker threads to handle client requests continually. However, because the thread pool creates an array of n_p *Thread* objects to pool the tasks, more memory is allocated for this array. For example, to maintain the waiting state for client requests, the SOAP server with the Thread Pool mechanism consumes memory approximately 125 KB with 5 pooling threads whereas the first strategy requires about 105 KB of memory no mater how many threads in one group will be created at a time.

5.2. Performance of Serving Time

To evaluate the serving time from client side, a comparison test between the light-weight SOAP server and the Apache SOAP running on the Tomcat servlet engine was done. A client used Wingfoot[3] SOAP running on a Desktop PC installed Linux operating system and JDK 1.4.1. The SOAP Server runs on an iPaq HP 3870 which was installed Familiar Pocket Linux. The communication link between the client and the server was a Universal Serial Bus (USB) connection which could be fast enough to produce a low delay of network connection.

Two groups were tested. The first group used the light-weight SOAP server and J2ME CDC/Personal Java Profile that were installed on the iPaq. The second group used Apache SOAP 2.3.1 and Tomcat 3.2.4 running on Blackdown Java 1.3.1 ARM version (installation packages were considered to be suitable for hardware resource of the iPaq).

The figure 8 depicts the interop test result. As shown in the figure, the round-trip delay produced by the first group was approximately 50 percent of that produced by the second group.

The figures 9 and 10 depict interop test results of the SOAP server (Mobile Information Device Profile (MIDP) version) running on Sony Ericsson P900 and Siemens S55 phones. The tests were done with T-Mobile GPRS (General Packet Radio Service) network. The SOAP server running on the P900 has a processing time

[3] http://www.wingfoot.com/index.jsp

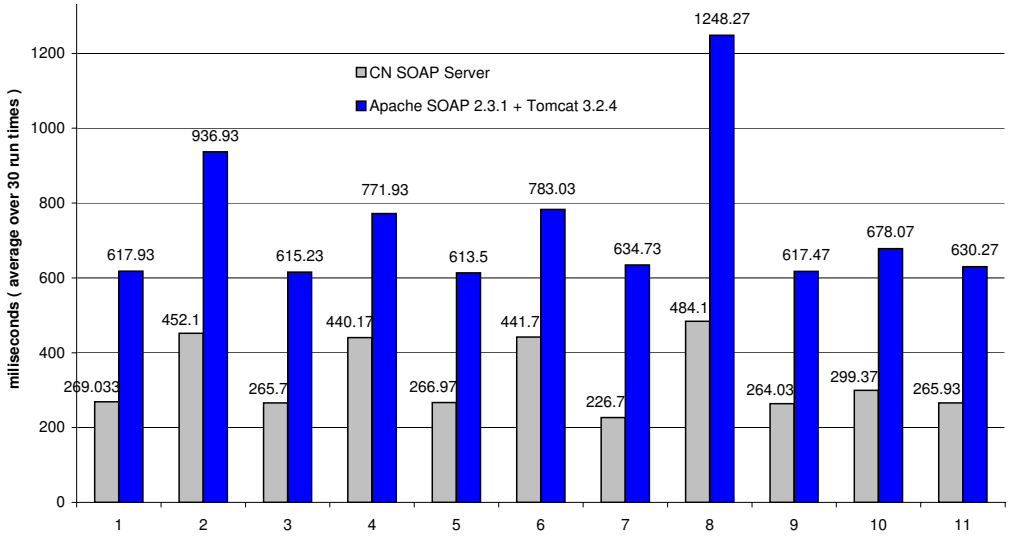

Figure 8: Comparison of Round-trip Time between CN-SOAP-Server and Apache SOAP Server on iPaq HP3870 PDA-Wingfoot SOAP client on desktop connecting to PDA via USB connection. Interop Services Methods: 1-echoString(); 2-echoStringArray(); 3-echoInteger(); 4-echoIntegerArray(); 5-echoFloat(); 6-echoFloatArray(); 7-echoStruct(); 8-echoStructArray(); 9-echoBoolean(); 10-echoDate(); 11-echoBase64()

approximately one-seventh total delay time that a client has to wait for a response (round-trip transmission time plus server processing time).

When running on the S55 phone, the processing time of the SOAP server increased obviously. The domination of the processing time over the transmission time shows that the SOAP server, despite being able to run on low-end phones, is rather heavy for them.

5.3. Memory footprint

For the version running on J2ME CDC/PersonalProfile (and also JVM of J2SE platform), the light-weight SOAP sever core package has a size of 22 KB. It is the size of the cnsoap.server package. The other packages like cnsoap.util, cnsoap.service can be variant in sizes. It depends on how many util-classes and service-classes are contained in the packages. At the moment, the following details can be listed:

- *cnsoap.server* has the package size of 22 KB
- *cnsoap.util* is about 19 KB including classes that support the Float, Double, Date, and Base64 data types.
- *cnsoap.services* has the *ServiceDescriptor* class of 1 KB and 21 service class files with 48 KB totally.

For the J2ME platform, MDIP/CDLC version, the light-weight SOAP sever running on the P900 and S55 mobile phones has the MIDlet jar file of 90 KB including 11 echo-test services. The detailed package sizes are:

- *cnsoap.server* has the package size of 29 KB
- *cnsoap.util* is about 20 KB including classes that support the Float, Double, Date, and Base64 data types.
- *cnsoap.services* has the *ServiceDescriptor* class of 1 KB and 17 service-class files with 40 KB totally.

For run-time memory test, the J-Sprint tool is used. It is a shareware Java profiler which does performance analysis of Java applications[4]. The peak of memory usage of the lightweight SOAP sever is approximately 295 KB and happens at the initialized phase when Java classes and dependent libraries are firstly called. When entering a stable working phase, about 125 KB of memory is needed to maintain the waiting state of the SOAP server. An additional memory amount of 45 KB is required by the SOAP server to handle each client request (echo service tests).

6. Conclusions

In this paper we introduce the HTTP server part of our mobile middleware based on Web Services. After the overview of the whole concept and architecture, a detailed description of the SOAP server is given. The realization have been evaluated in consideration of the performance criteria like latency, processing time and memory usage on mobile devices.

[4]Information available at http://www.j-sprint.com

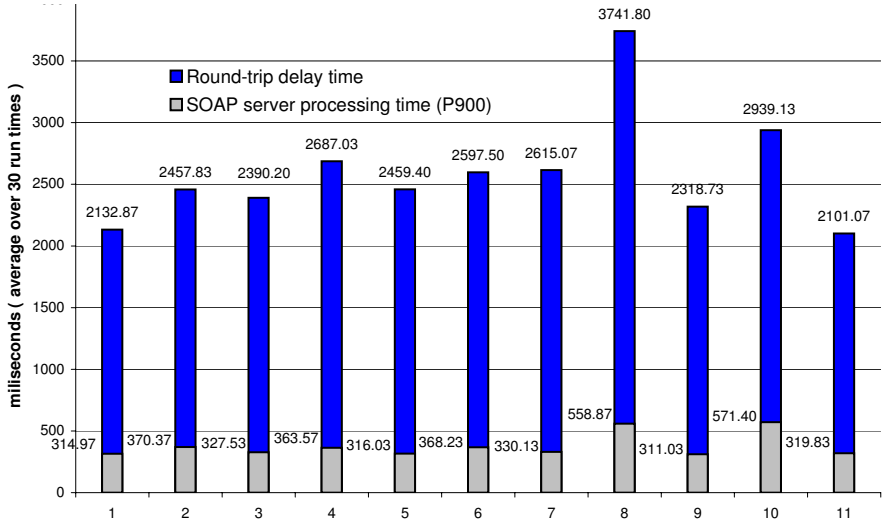

Figure 9: Comparison of Client Round-trip Delay Time and SOAP-Server Processing Time on P900 phone. Interop Services Methods: 1-echoString(); 2-echoStringArray(); 3-echoInteger(); 4-echoIntegerArray(); 5-echoFloat(); 6-echoFloatArray(); 7-echoStruct(); 8-echoStructArray(); 9-echoBoolean(); 10-echoDate(); 11-echoBase64()

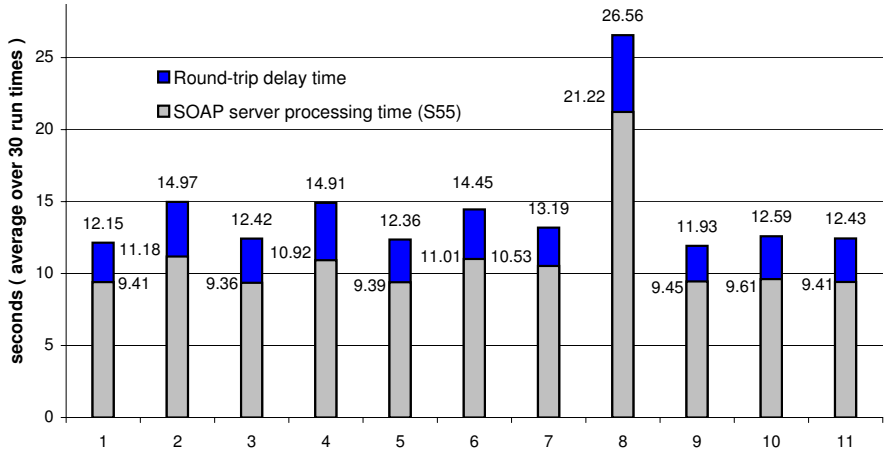

Figure 10: Comparison of Client Round-trip Delay Time and SOAP-Server Processing Time on S55 phone. Interop Services Methods: 1-echoString(); 2-echoStringArray(); 3-echoInteger(); 4-echoIntegerArray(); 5-echoFloat(); 6-echoFloatArray(); 7-echoStruct(); 8-echoStructArray(); 9-echoBoolean(); 10-echoDate(); 11-echoBase64()

REFERENCES

[1] Tom Clements. Overview of soap. Published on the internet. Available at URL `http://developer.java.sun.com/developer/technicalArticles/xml/webservices/`, January 2002.

[2] Roberto Chinnici et al. Web services description language (wsdl) version 1.2. Published on the internet. Available at URL `http://www.w3.org/TR/wsdl12`, June 2003.

[3] Guido Gehlen and Ralf Bergs. Performance of mobile web service access using the wireless application protocol (wap). 05 2004.

[4] Guido Gehlen and Geogios Mavrosmatis. Mobile web services based middleware for context-aware applications. Apr 2005.

[5] Guido Gehlen and Linh Pham. Mobile web services for peer-to-peer applications. Jan 2005.

[6] Nilo Mitra. Soap version 1.2 part 0: Primer. Published on the internet. Available at URL `http://www.w3.org/TR/soap12-part0/`, June 2003.