Master Thesis
Software Engineering
Thesis no: MSE-2002-28
August 2002

Performance of SOAP in Web Service Environment compared to CORBA

Causes and Improvements

Robert Elfwing
Ulf Paulsson

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Robert Elfwing
elfwing@home.se
Ulf Paulsson
ulf.paulsson@bredband.net

External advisor:
Pär Karlsson M.Sc., Lic. Tech.
par.karlsson@epk.ericsson.se
Ericsson AB
P.O. Box 518
SE-371 23 Karlskrona, Sweden

University advisor:
Lars Lundberg Professor
Department of Software Engineering and Computer Science

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/ipd
Phone : +46 457 38 50 00
Fax : +46 457 271 25

ABSTRACT

Web Services is one of the latest golden concepts that promise flexibility and unlimited potential for interconnection between systems of the same or different type. The communication is based on SOAP – Simple Object Access Protocol, which is founded on XML (eXtended Markup Language). We have made experiments with SOAP in a Web Service Environment to find out the performance in response time using SOAP compared to CORBA (Common Object Request Broker Architecture). Unsurprisingly CORBA is significantly faster but with the improvements that we implemented and an intelligent choice of XML parser the response times for SOAP was drastically lowered.

Keywords: SOAP, performance, Web Service, CORBA

Contents

Performance of SOAP in Web Service Environment compared to CORBA

Causes and improvements.

Robert Elfwing pt98rel@student.bth.se
Master's Degree Student in Software Engineering
Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Sweden*.

Ulf Paulsson pt98upa@student.bth.se
Master's Degree Student in Software Engineering
Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Sweden*.

Abstract

Web Services is one of the latest golden concepts that promise flexibility and unlimited potential for interconnection between systems of the same or different type. The communication is based on SOAP – Simple Object Access Protocol, which is founded on XML (eXtended Markup Language). We have made experiments with SOAP in a Web Service Environment to find out the performance in response time using SOAP compared to CORBA. Unsurprisingly CORBA (Common Object Request Broker Architecture) is significantly faster but with the improvements that we implemented and an intelligent choice of XML parser the response times for SOAP was drastically lowered.

1 Introduction

Web Services (WS) is one of the latest hypes in the IT-industry today. The promise is unlimited connection and interoperability in combination with new business opportunities. The idea of interoperability is not original. There have been a number of implementations that give solutions for this concept for example CORBA, RMI (Remote Procedure Invocation), OSI (Open System Interconnection), RPC (Remote Procedure Call) and so on. The definition of Web Services is:

"*A Web service is a software application identified by a URI (Uniform Resource Identifier), whose interface and binding are capable of defining, described and discovered by XML artefacts and supports direct interactions with other software applications using XML based messages via internet-based protocols.* " [28].

This clarifies the difference between WS and earlier solutions, which is that WS is based on commonly used standards that are platform independent.

The concept of WS is to publish a service on the Web enclosing the interface that is needed to find and use the service. When a consumer has found a service that fulfils his needs he uses the interface to connect to the service.

Possible usage scenarios are:

- Services that are done in-house may be outsourced to specialized entities and thereby decreasing the cost.
- It will be possible to connect legacy systems with a relative low effort.
- It will be possible to connect business systems to customers' and providers' business systems in order to increase control over payments and supply.

The WS concept is fairly new. It arrived in 1999 and was quickly adopted by Microsoft®. Since then it has grown remarkably fast and according to the Gartner Group all leading e-business platforms will support at least the basic Web Service infrastructure by 2003 (0.8 probability) [24]. The Meta Group has similar belief [23]. The Gartner Group also believes that Web Services will be involved in more than 40 percent of the revenue growth in IT-related markets through 2006 (0.6 probability) [22].

There are problems though with Web Services. The industry and involved communities are concerned about the performance and security aspects. The communication between provider and consumer is done by using SOAP (Simple Object Access Protocol). When creating SOAP there was a trade off where performance was sacrificed for simplicity and flexibility. The performance part will have a great impact because of the costly investments in new hardware. The focus of the security issue is the lack of commonly accepted standards that must be addressed if there is to be a standardised way to perform transactions.
In this paper we focus on the performance part of the communication between the provider and the consumer of a Web Service (see appendix A – Web Services). What is the capacity of the SOAP protocol? Is it really as low as the industry and communities fear? We will make an experiment where we measure the SOAP capacity in a Web Service environment and compare it to an index, IIOP (Internet Inter-ORB Protocol) in a standard configuration of CORBA.

* This work was performed at Ericsson AB, Karlskrona Sweden.

2 Purpose

The purpose is to evaluate the performance aspect of the SOAP protocol in a Web Service environment and to identify factors, which have impact on the performance, and improvements. Two ways to evaluate performance are either to make some static measurements for example number of bytes (payload) transmitted per second or it can be done by compare SOAP to a well-known protocol. We preferred the second alternative. The comparison we choose to do was between SOAP and CORBA. The reason for this is that the study was done at Ericsson and they preferred CORBA, which is well known and used in the selected environment for them.

The research questions are then:
What is the response time performance of SOAP in comparison to CORBA in a Web Service environment? What factors have impact on SOAP response time and is it possible to improve the response time?

3 SOAP

SOAP is a lightweight protocol based on the XML specification. The typical usage is to exchange structured and typed information between peers. The standardisation is the key to the interoperability, which is the fundamental principle of SOAP. In it simplest form SOAP is one-way but may also be used in more complicated transaction like EDI (Electronic Document Interchange) and RPC, which was one of the design goals when creating SOAP [1].

Figure 1. One-Way SOAP.

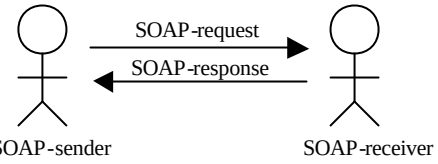


Figure 2. RPC SOAP.

3.1 THE SOAP MESSAGE

The SOAP message consists of [2]:

SOAP envelope
Encloses the SOAP-message.

SOAP header (optional)
The header contains information for the SOAP-node, the processor of the SOAP-message, how to process the SOAP-message. This may be authentication, routing, delivery setting and more.

SOAP Body (mandatory).
Contains information that is targeted to the ultimate SOAP-message receiver for example a service that an application makes requests to.

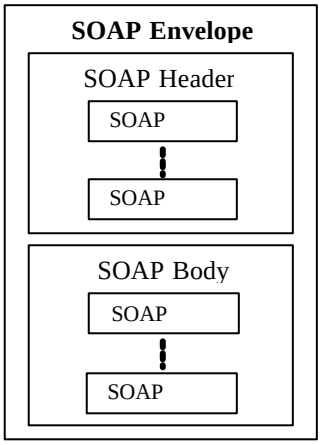


Figure 3. The SOAP Envelope.

As one can see in figure 4 SOAP is quite verbose. This is one of the main critiques against SOAP.

```
<?xml version="1.0" ?>
<env:Envelope
xmlns:env="http://www.w3.org/2001/12/soap-envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Figure 4. A SOAP-message.

3.2 TRANSPORT OF SOAP

The SOAP-envelope is independent of what transport protocol is used. Examples of protocols that may be used are HTTP (Hyper Text Transfer Protocol), SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol) or POP3 (Post Office Protocol). The protocol mostly used today is HTTP over port 80. The main advantage is that the SOAP envelope is integrated into a standard HTTP request and is interpreted as a Web request by firewalls. This removes costly reconfigurations of existing

communication management and security policies. One other advantage is that HTTP is a request-response-based protocol, which is compatible with the RPC-paradigm.

3.3 XML-PARSING

The SOAP messages must be parsed and interpreted before they can be invoked, which is done by the XML-parser. There are two different types of parsers, DOM (Document Object Model) and SAX (Simple API for XML).

3.3.1 DOM

DOM parsers parse the XML document and create an object-oriented hierarchal representation of the document, which can be navigated during run-time. The structure is intuitive and easy to manipulate but is very resource demanding both in memory and CPU usage [20].

3.3.2 SAX

SAX parsers [5] do not store any information. Instead they scan the information and calls handler functions that are associated with specific instructions and tags in the XML-document. The SAX parser is probably faster than the DOM parser and is not as resource intensive. The drawback is that the logic for handling the XML document instructions and tags is significantly more complicated and complex. There is a variant of SAX that is called Pull Parser but we do not make a distinction between them in this work.

4 Method

We set out with the assumption that SOAP was significantly slower than CORBA and that we should be able to identify the causes for this. Findings by Davis and Parashar [4] partly confirmed this assumption. The main goal was to find possible improvements that would make the usage of SOAP more palatable. The method to do this was divided into six steps:

1. Implement a prototype that simulates an industry application in a Web Service environment (see appendix B).

2. Measure the performance with and without load to identify discrepancies between CORBA and SOAP.

3. Analyse the discrepancies found.

4. Identify the causes for the discrepancies.

5. If possible find and implement improvements for the SOAP implementation.

6. Measure and evaluate the improvements.

The centre of the method is a prototype placed in a Web Service environment. In our case a charging system where it was possible to credit, debit, make reservations and do credit controls on accounts in a mobile phone system (see appendix B for further details). The prototype consisted of a client and a server part. The measuring was done in the client. The communication between the client and server were either conducted by using SOAP or CORBA. We put load on the server by using load generators that we tailor made for this purpose. The generators had different patterns of behaviour simulating four different types of load:

- Traffic.
  The generators followed a specific traffic composition where the requests were Poisson distributed with the intensity (lambda) set to 200 requests per second when using CORBA and 10 requests per second when using SOAP. The difference between CORBA and SOAP was caused by the fact that the Web server hosting the SOAP application were not able to process more requests. The requests (functions) are described in appendix B.

- Stress.
  Both generators started five threads where each of the threads sequentially sent as many requests as possible.

- Peak.
  Both generators sent bursts of requests with maximum speed and the length of 1 second. The bursts were Poisson distributed with an intensity of 1 burst every other second.

- None.
  The load generators were not active.

The set-up of the test configuration is shown in figure 5 and 6.

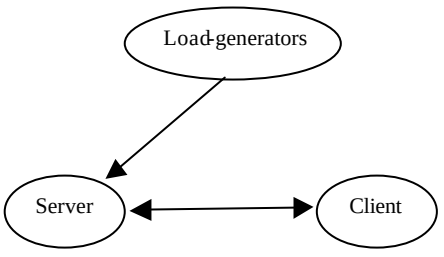


Figure 5. Test setup.

We placed the server, client and load generators on different machines. The load generators made requests to the server in the same manner as the prototype. On the client side we sent SOAP or CORBA based requests sequentially and measured the response time. This was done during all the different load patterns. The test runs were done during night to minimize the impact of the background noise on the LAN.

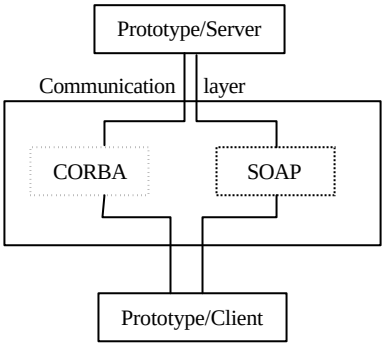


Figure 6. Model of the prototype.

We captured the traffic between the client and the server in order to make a detailed analysis of the packets exchange in the SOAP and CORBA requests.

4.1 VARIABLES

There were two variables that we wanted to examine closer when using SOAP as communication method. They were:

- HTTP 1.0 or HTTP 1.1
- Choice of XML-parser

4.1.1 HTTP

There has been some discussion in using either HTTP 1.0 or HTTP 1.1. The main difference is that version 1.1 use persistent connections where version 1.0 do not. This may have impact on the performance when there are several requests at the same time.

4.1.2 XML-Parser

The type of parser used may also have impact on the performance. The two main types are SAX and DOM. The SAX parser is believed to be faster than the DOM parser. Different implementations of the parsers may also have an impact. The SOAP implementation we used in the experiments applied the Xerxes SAX parser [29]. We compared this parser to the Crimson parser that is included in the JDK (Java Development Kit) 1.4 [30]. We made a test setup where we fed the parsers SOAP calls in the form of byte streams. This to simulate that the SOAP calls arrived by LAN. The SOAP calls tested were identical to those that were used by the SOAP prototype.

5 Experimental setup

5.1 ENVIRONMENT

The environment consisted of:

- One Compaq Deskpro Pentium II 400 MHz with 192 MB RAM, which was used as server.
- One Compaq Deskpro Pentium II 600 MHz with 256 MB RAM, which was used as client (measuring unit).
- One Compaq DeskPro Pentium II 600 MHz with 256 MB RAM, which was used as load generator.
- One SUN Ultra 10 with 512 MB RAM, which was used as load generator. We estimated that we needed this additional load generator to be able to make sufficient SOAP requests.

All PC:s were configured with Windows 2000 OS. The Unix machine used SunOS 5.8.

The connection was a 100 Mbps Ethernet LAN.

The SOAP environment consisted of Apache's AXIS beta release 1[37]. The web server that AXIS used was TomCat 3.3 in a standard configuration. The CORBA environment consisted of the CORBA package that is included in Sun's JDK version 1.3.
All programs were written in Java using JDK version 1.3 except the CORBA server that used JDK 1.4.

5.2 TCP/IP PROTOCOL ANALYSER

We used Ethereal, version 0.9.3, as the analyzer. We captured the traffic between the server and client for both CORBA and SOAP function calls.

6 Results

6.1 CORBA

We made 10 000 requests of each function to the server. The average times in table 1 are average time for each CORBA call and not for the complete function calls. Certain functions contained several CORBA calls. The calculated averages do not include the initializing cost for either CORBA or SOAP.

Type of interference	Average time (milliseconds)	STD
None	1.63	0.17
Stress	3.15	0.22
Peak	1.70	0.26
Traffic	2.83	0.32

Table 1. CORBA response time results

6.2 SOAP

Similarly to CORBA we started with 10 000 requests of each function. This showed to be very time consuming and took approximately 25 hours when the load generators were not active. We reduced the number of function calls to 5000 in the measuring thread when we used the load generators. The performance of SOAP was lower than expected even when the load generators were not active. The average times in table 2 are for each SOAP call and not for the complete function calls.

We met problems with the Tomcat server in the SOAP case. We over estimated the number of requests that the Tomcat server could process. In theory the traffic pattern should make less requests per second than the stress pattern but in reality the traffic pattern reached the limit of the Tomcat capacity and this caused that the results

Type of interference	Average time (milliseconds)	STD
None	679.2	3.14
Stress	749.2	12.0
Peak	711.9	12.7
Traffic	754.7	11.5

Table 2. SOAP response time results.

for stress and traffic to be similar.
The CORBA and SOAP implementation did not behave in the same manner when the load generators were active. When the CORBA implementation was stressed the response time was double but in the SOAP implementation the response time was increased by 10%. This implies that there were costs in the SOAP implementation that were independent of the load and this made us focus on other parts than the load cases. We changed our center of attention to the traffic analysis of the packet exchanges when there was no load in order to find discrepancies and causes for these costs (see 6.5).

6.3 HTTP 1.0/1.1

We were not able to configure Axis to use HTTP1.1. See discussion (section 7.2.2) for further details.

6.4 PARSERS

The parsers that we compared were the Crimson and the Xerces implementations. Xerces was used by Axis and Crimson was included in JDK 1.4. The results are shown in table 3 and in figure 7. The first parse times that contained

Parser	Average time (milliseconds)	STD
Xerces DOM	50.01	0.64
Xerces SAX	18.72	1.40
Crimson DOM	4.34	0.44
Crimson SAX	3.36	0.22

Table 3. Parse times for different parser implementations.

initiating costs were not included in the calculation of the average parse times. Predictably SAX parsers were significantly faster than DOM parsers from the same manufacturer. The large difference between different implementations of parsers was a surprise though. The Xerces SAX parser was on average 6 times slower than the Crimson SAX parser. Even the Crimson DOM parser was faster. Of course this is only applicable for the types of XML documents that we used, which

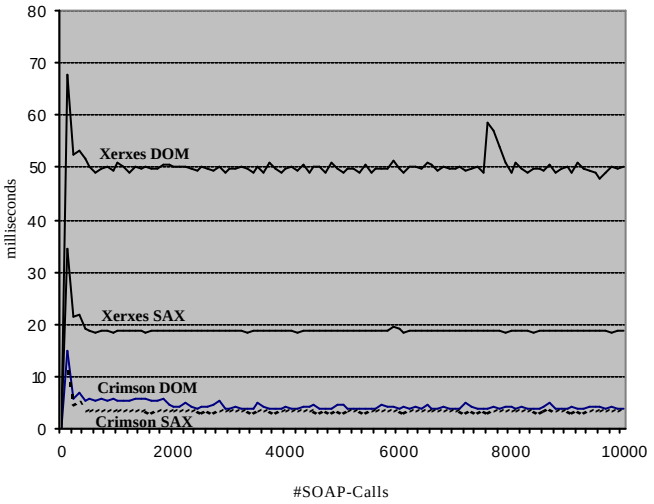


Figure 7. Parse times for different implementations.

were quite small (see appendix C for examples of SOAP calls). These findings are similar to findings in other comparisons [31,32,33].

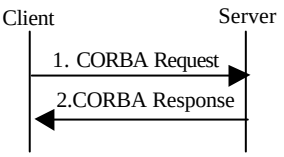


Figure 8: Traffic for a CORBA call

6.5 TCP/IP TRAFFIC ANALYSIS

The response time ratio between SOAP and CORBA was 400:1, when the load generators were not active. This was a larger difference than expected [4]. Putting load on the servers showed that the load itself were not responsible for the large differences between the CORBA and SOAP implementations. The response time for CORBA was doubled but the increase for

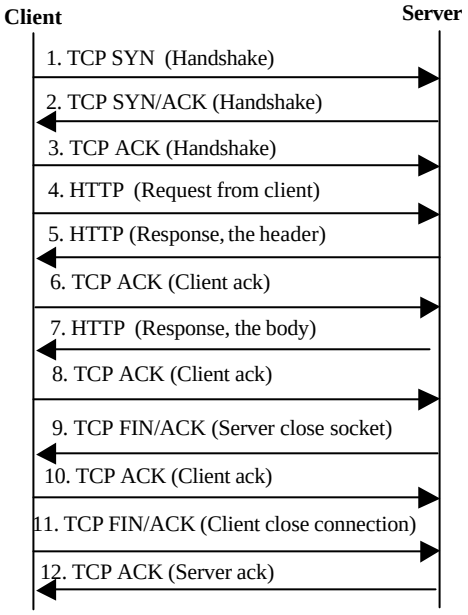


Figure 9: Traffic for a SOAP call.

SOAP was only 10 percent. We analyzed the TCP traffic in detail to find other explanations for the ratio. The traffic behaviour of SOAP and CORBA was captured and scrutinized when the load generators were not active. The behavior of CORBA requests is described in figure 8 and of SOAP requests in figure 9. Observe that these figures only show the behaviour for an atomic request. The initial traffic exchange during setup is not shown.

Axis Beta 1 implementation		
Packet	Average (milliseconds)	STD
1. TCP SYN	0	0
2. TCP SYN/ACK	0.2	0.02
3. TCP ACK	0.06	0.004
4. HTTP	3.4	3.6
5. HTTP	36.2	16.0
6. TCP ACK	152.7	27.7
7. HTTP	0.36	0.03
8. TCP ACK	199.9	0.04
9. TCP FIN/ACK	339.7	153.2
10. TCP ACK	0.1	0.01
11. TCP FIN/ACK	0.3	0.02
12. TCP ACK	0.8	0.3
Sum	**733.5**	

Table 4. Axis Beta 1, packet inter-arrival times.

The ackumulated average inter-arrival times for each packet in SOAP calls are described in figure 10 and table 4. Observe that the data in figure 10 and table 4 was caught during daytime when there was more background noise on the

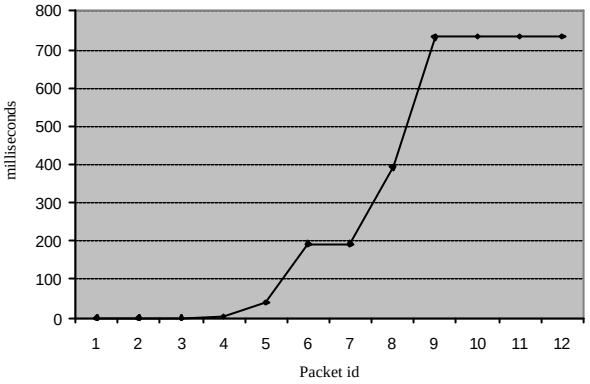


Figure 10. Accumulated mean packet arrival time (Axis).

LAN, which explains the higher values. The packets of interest are 5,6,8 and 9.

6.5.1 Packets 5,6 and 8

The HTTP packet (packet 4) from the client contains the request to the server. The server then returns two HTTP packets (packet 5 and 7) where the first packet contains the header and the second packet contains the body of the response.

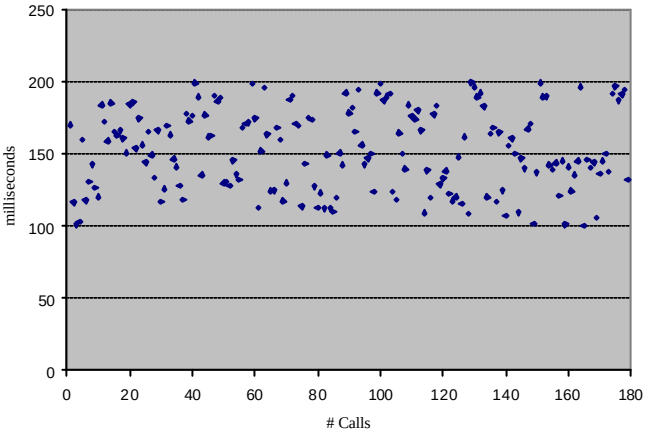


Figure 11. Distribution of delays for packet 6 (ACK).

It seems odd that the response from the server is divided in two parts. The reason for this may be that Apache did this to work around a problem in HTTP/1.0 when using persistent connections. The problem was that data sent in the same segment as the HTTP header was ignored. It seems reasonable to believe that this fx still exists in the Tomcat server.

When the client get the first HTTP packet (nr 5 in figure 9) the client will wait a certain time to

see if a new packet in the established TCP connection will be sent so the ACK can get a free ride. This is a feature called TCP delayed ACK [34,35,36] in the TCP specification [21] and the purpose is to reduce traffic. In the Axis beta 1 implementation the client will wait the whole delay time since no new data will be sent to the server. We measured this delay and the average delay time was 150 milliseconds. The range was between 100 and 200 ms and the delay times were evenly distributed (see figure 11) within this interval. It seems that the delay times are randomized rectangular within the interval. This is confirmed by Cardwell et al [12]. This happens again when the second HTTP packet (number 7 in figure 9) reaches the client. This delay though is almost constant to 200 milliseconds (see figure 12). The delayed ACK timer in Windows 2000 is set to 200 milliseconds by default.
The server will not send packet 7 (HTTP body) before the ack on the prevoius packet (HTTP header) has arrived. This behavior is caused by the Nagle algorithm, which was implemented to stop the sending of repeating small packages such as keyboard signals when using Telnet. The Nagle algorithm is described in RFC896 [3], RFC 1122 [34] and in [10]. The same behavior was noted by Davis and Parashar [4] and Heidemann [11].

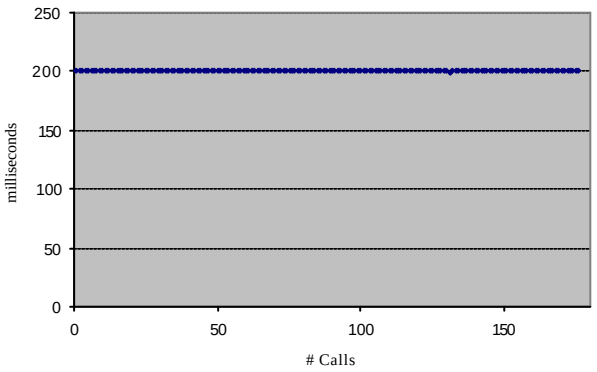


Figure 12. Distribution of delays for packet 8 (ACK).

The conclusion, for these packets, is that they on average make a delay of 350 milliseconds for each SOAP request. The source for these delays is that both the TCP delayed ACK on the client and the Nagle algorithm on the server is active at the same time. If either of them had been inactive the delays would not have an impact on the response time.

We tried to remove these delays by improving the client and server implementations. This is discussed in section 6.6.

6.5.2 Packet 9

This packet is responsible for the major part of the response time of a SOAP call. It ranges from 7,9 milliseconds to 439,0 milliseconds with an average of 342 milliseconds (see figure 13). This is caused by the implementation of the client. We checked the Axis source code and found strong indications on that the socket implementation read from the input stream until end of file was encountered. This means that the

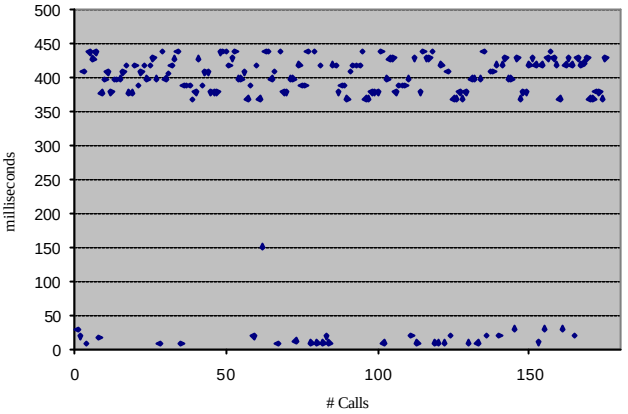


Figure 13. Distribution of delays for packet 9 (FIN/ACK).

client side waits until the server side closes the socket.

6.6 IMPROVEMENTS

There are a number of possible improvements that were identified:

1. Parse the content length in the first HTTP packet of the response from the server.
2. Send the server response in one HTTP packet.
3. Disable the Nagle algorithm.
4. Set the TCP delayed ACK to zero in the operative system.

6.6.1 Parse the content length

The main alternative is to parse the content length field in the HTTP header and stop listening to the socket and close it when we have received the specified number of bytes. This should remove the delay times for packet 8 and 9. To check this we first built a simple socket implementation at the client side that would copy the behavior of the Axis beta 1 client implementation to confirm that we had the same fundament as the charging prototype. The same behavior occurred here as in the prototype and the average response time was 660 milliseconds without load generators. If we add the average parsing time (approx. 19 milliseconds) at the client side we reach very similar values as

encountered in our measures (680 milliseconds). This is shown in figure 15. We made another client implementation to try the hypothesis of parsing the content length field. This worked very well. The TCP traffic in a SOAP call for this implementation is shown in figure 14 and table 5. Observe that when reaching the bold line in the table the client stops listening to the socket and is able to continue execution. The

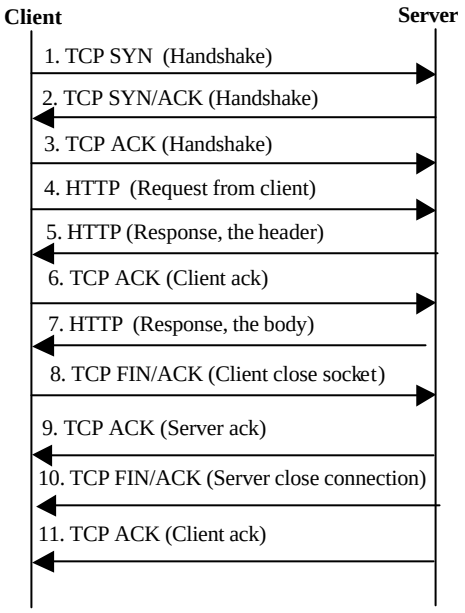


Figure 14. Traffic for a SOAP call when client close socket.

average response time of the SOAP calls where reduced to a third (200 milliseconds), which complies to that the average delay for packet 8 and 9 was two thirds of the total time. There is a problem with this solution though. The specification for HTTP 1.0 [16] do not demand that content length is included in the header.

Client Close Socket Implementation		
Packet	Average (milliseconds)	STD
1. TCP SYN	0	0
2. TCP SYN/ACK	0.3	0.018
3. TCP ACK	0.1	0.003
4. HTTP	1.4	1.1
5. HTTP	16.6	3.4
6. TCP ACK	180.9	4.8
7. HTTP	0.3	0.01
Sum	**199.6**	
8. TCP FIN/ACK	1.5	2.9
9. TCP ACK	0.2	0.008
10. TCP FIN/ACK	143.0	6.2
11. TCP ACK	0.1	0.008
Sum	**344.4**	

Table 5. TCP packets' inter-arrival times.

6.6.2 Response in one HTTP packet

We built a simple server which received the request from the client and return a response in one HTTP packet. This worked well. The result for this implementation was that packet 6 was removed and the delay of on average 150 milliseconds were eliminated. We were not able to do this improvement on the Tomcat and therefore could not implement the charging system with this improvement. In implementations where the request or the response will exceed maximum packet size of TCP this problem will recur but again if they are large enough the Nagle algorithm will be negated.

6.6.3 Disable the Nagle algorithm

It is possible to disable the Nagle algorithm in Java when using sockets. This should be done on the server side. The *tcp_nodelay* parameter for sockets is set to true and the delay for Nagle algorithm is set to zero. We tried this but were not able to do so in the Tomcat server implementation. Davis and Parashar [4] were successful though in doing this so the conclusion we make is that this should be considered in Web Services implementations.

Figure 15. Response time for different implementations.

6.6.4 Set the TCP delayed ACK to zero

We tried to remove the delay time for packet 6 by setting the value for the **TcpDelAckTicks** registry key in Windows 2000 to zero. We were not able to do this. The cause for this is that there is a bug in Windows 2000 where the default value of 200 milliseconds is not changed even though we set the key to some other value[14]. This solution has a tradeoff though. If the TCP delayed ACK is removed the load on the network will increase and this will probably affect the behaviour of other Internet applications.

7 Discussion

The goal of this discussion is to identify the performance discrepancies and the causes for them and to make viable proposals for improvements. Before this is done we must make clear that the Web Services technology is in its infancy and many solutions at this stage are often rough-hewn.

That there should be a significant difference between CORBA and SOAP in performance was no surprise. CORBA is a mature technology and SOAP is not. That the difference would be a factor of 400, when the load generators were not active, was a surprise though. But is it really this high?

We have identified two factors that had considerable impact on the performance for SOAP. They were the choice of parser and the implementation of the communication handlers.

7.1 PARSERS

Axis beta 1 used the Xerces SAX parser. This is not mandatory. The developer may choose which parser that is to be used when parsing the SOAP calls. There are savings to be done here in the server and the client execution time. In heading 6.4 we showed different parse times for two parsers when parsing the SOAP requests on a byte stream. The conclusion here is that it is highly recommended to check which parser that has the best performance with the type of XML documents (SOAP requests) that will be common when requesting the functions of the Web Service implemented. If we had used the Crimson SAX parser, which is included in JAVA 1.4, instead of the Xerces SAX parser the average parse time would be lowered by more than 80 percent. This improvement would lower the response time for a SOAP call on average with 2 percent in the original test setup. But if we introduced the proposed improvements discussed in result the choice of Crimson would lower the response time with 10% and if we manage to remove the delay of packet 6 (ACK) the response time would be lowered by 30%.

7.2 COMMUNICATION

7.2.1 TCP traffic analysis

In figure 12 and in table 4 we show the inter-arrival times for each packet in a SOAP call. As you can see packets 5, 6, 8 and 9 stands for the major part of the SOAP request response time. The delays are caused by an unfortunate combination of the TCP delayed ACK at the client and the Nagle algorithm at the server. We have discussed this in depth earlier and the conclusion is that the implementation of the communication handling at the server and the client is of high importance. The implementation in Axis is without finesse but it works. In the original test setup the response time was on average 680 milliseconds including server and client execution time, which were approximately 40 milliseconds. With improvements that were realizable to make in the prototype charging system the response time was lowered to 200 milliseconds and execution time on server and client was the same. If the response would be in a single HTTP packet we would remove the delay caused by packet 6 and the response time would be approximately 42 milliseconds. In real Web Services the execution time on the server and the roundtrip time will probably increase.

7.2.2 HTTP 1.0/1.1

The main problem is that HTTP 1.0 is not well functioning with the TCP protocol. Some of the problems that are applicable in our case are [25, 26, 27]:

- The TCP handshake causes delays.
- Nagle algorithm in combination with the TCP delayed ACK causes unnessecary delays.

HTTP 1.1 attempt to solve these problems and more [7,17,27] and will probably lower the response time [7,9,15,18,19], which was what we hoped for when designing the experiments. When we analysed the traffic using HTTP 1.0 we saw that the causes for the major delays can be removed by improving the implementation.

The communication cost for the TCP handshake is less than 1 % of the parsing cost, which is the real cost. This is in a LAN environment though where round trip time is short and packet loss rare. If the round trip time is high the situation would be reversed where the communication cost and the parsing cost would be substantial. HTTP 1.1 will behave like CORBA that is setting up a connection and when this is done all requests will be sent on this connection. This removes the costs for two roundtrips (three-way handshake and connection close) for each request done. The usage of HTTP 1.1 will also change the behaviour of the combination Nagle algorithm and TCP delayed ACK.

It seems that the benefits with HTTP 1.1 may be profitable, in theory. This will probably be strengthened when security aspects are implementeted in the Web Services.

HTTP 1.1 may have a negative impact on system design that may lengthen response time

though. This conclusion was derived by Barford [7,17] and Crovella [7]. They show in a Web Server study that HTTP 1.1 improves performance system as long as the disk system is not a bottleneck where the performance will drastically be lowered. Heidemann [11] points out implemetation errors that can make persistant HTTP 20 times slower than HTTP 1.0.

The choice of HTTP 1.0 or HTTP 1.1 may be a critical choice but other factors such as the communication implementation and choice of parser seems to be more important at this moment. The main effort should be in these two areas.

7.3 THEORETICAL MINIMUM

In the CORBA implementation (without load) the response time was 1.63 milliseconds. In the SOAP implementation (without load) with the suggested improvements the response time was lowered to 42 milliseconds, which includes two parses of a SOAP message, one for the request and one for the response. The total parse time is approximately 37 (2*18.72=37.4)[1] milliseconds when using the Xerces SAX parser. If we used the Crimson SAX parser instead of the Xerxes SAX parser the parse time would be lowered to 7 milliseconds (2*3.36=6.72). The response time for the SOAP implementation would now be 7 + (42-37) = 11 milliseconds. This would decrease the ratio in response time between SOAP and CORBA from 400:1 to the order of 7:1. This ratio would be the theoretical minimum.

8 Conclusion

We have compared the usage of SOAP over HTTP 1.0 and CORBA on a simple Web Service that used Apache Axis beta 1 as SOAP implementation and JAVA 1.3 implementation of CORBA. When we used Apache Axis as is the ratio between SOAP and CORBA was 400:1 in response time. After analysis of the Axis implementation behavior when handling the communication and choosing of parser we managed to lower the theoretical ratio to 7:1 in response time. The conclusion is that when implementing Web Services the developers must address these two important factors:

1. How to handle the communication as efficient as possible.

2. Choose a parser that is suitable for the type of XML documents (SOAP calls) used in the Web Service.

[1] See table 3 in section 6.4.

9 Future work

In this work we have produced several usable results but there are still questions unanswered. These will hopefully be answered in the future. The questions that we suggest to be examined are:

- Make performance testing for several different implementations of Web Service systems using our proposed improvements.

- What impact does different sizes of the SOAP message have on the response time performance?

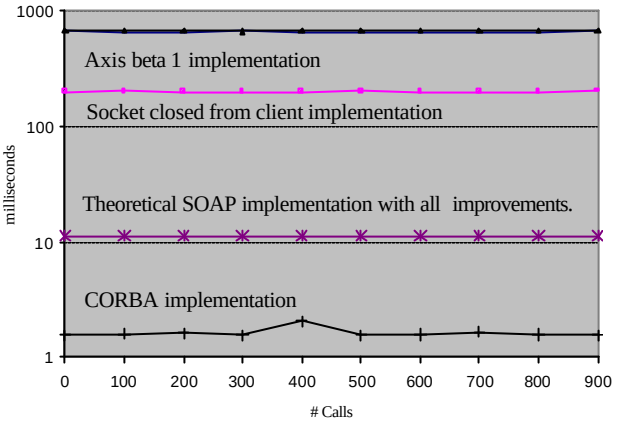


Figure 16. Response time SOAP vs. CORBA (log scale)

- Performance testing using other protocols instead of HTTP 1.0 for example HTTP 1.1, SMTP etc.

- Performance testing in different network environment for example over the Internet to find the real communication cost in the surroundings where Web Services will be placed.

- What impact security additions to the Web Services have on the performance?

- How is the performance of a Web Service affected by different TCP implementations in different operative systems?

10 Acknowledgements

This master thesis was made in co-operation with Ericsson AB at Karlskrona Sweden. We want to express our gratitude to professor Lars Lundberg at the department of Software Engineering and Computer Science, Blekinge Institute of Technology and to Pär Karlsson at Ericsson AB for their support and help.

References

1. *SOAP Version 1.2 Part 1: Messaging Framework*. http://www.w3.org/TR/soap12-part1/

2. *SOAP Version 1.2 Part 0:Primer*. http://www.w3.org/TR/soap12-part0/

3. Nagle, J. *Congestion Control in IP/TCP Internetworks.* RFC 896. January 1984. http://www.faqs.org/rfcs/rfc896.html

4. Davis, D. Parashar, M. *Latency Performance of SOAP Implementations.* IEEE Cluster Computing and The Grid 2002.

5. The official website for SAX Parsers. http://www.saxproject.org/

6. Nielsen, H F. Gettys, J. Baird-Smith, A. Prud'hommeaux, E. Wiul Lie, H. Lilley, C. *Network Performance Effects of HTTP 1.1, CSS1, and PNG.* In Proceedings of ACM SIGCOMM ´97, Cannes, France, Sept 1997.

7. Barford, P. Crovella, M. *A Performance Evaluation of Hyper Text Transfer Protocols.* Proceedings of the ACM SIGMETRICS '99 Conference, Atlanta, Georgia, May 1999.

8. Barford, P. Crovella, M. *Generating representative Web Workloads for Network and Server Evaluation*. Proceedings of ACM SIGMETRICS '98.

9. Chiu, K. Govindaraju, M. Bramley, R. *Investigating the Limits of SOAP Performance for Scientific Computing*. The Eleventh IEEE International Symposium on High Performance Distributed Computing Proceedings of HPDC 11, 2002.

10. W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1994.

11. Heidemann, J. *Performance Interactions Between P-HTTP and TCP Implementations.* ACM Computer Communication Review, 27 2, 65-73, April 1997.

12. Cardwell, N. Savage, S. Anderson, T. *Modelling TCP Latency*. INFOCOMM 2000. March 2000.

13. E. Nahum, T. Barzilai, and D. Kandlur. *Performance issues in WWW servers*. ACM SIGMETRICS Performance Evaluation Review, 27(1): 216-217, 1999.

14. Microsoft Product Support Services. The TcpDelAckTicks Registry Value Has No Effects on Ack Timeouts (Q311833). http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q311833

15. Binzhang, L. Fox, E A. *Web Traffic Latency: Characteristics and Implications.* Journal of Universal Computer Science. Volume 4, Issue 9. 1998.

16. Berners-Lee, T. et. al. *Hypertext Transfer Protocol -- HTTP/1.0*. RFC 1945. May 1996. http://www.faqs.org/rfcs/rfc1945.html

17. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee. *Hypertext Transfer Protocol – HTTP 1.1.* RFC 2068, January 1997. http://www.faqs.org/rfcs/rfc2068.html

18. Mogul, J. *The Case for Persistent-Connection HTTP.* In Proceedings of the SIG-COMM '95, pages 299-313. ACM, August 1995.

19. Barford, Paul; *Modelling, Measurement and Performance of World Wide Web Transactions*, Doctoral Dissertation, December 2000.

20. W3C Architecture Domain. Document Object Model (DOM). http://www.w3.org/DOM/

21. E.J. Postel, *Transmission Control Protocol*. RFC 793, September 1981.

22. Plummer, D. *Key Entry Points Into Web Services Markets*. The Gartner Group. 21 Feb. 2002.

23. Roth, C. *Web Services: The Vendor Landscape*. Meta Group. 10 Oct. 2001.

24. Plummer, D. Smith, D. Andrews, W. *What Web Services Will and Won't Do.* The Gartner Group. 19 Feb. 2002.

25. Opyrchal, L. *HTTP-ARDP: Faster, More Flexible HTTP?* EECS 598 Final Project Report.

26. Spero, S E. Analysis of HTTP Performance problems. http://www.ibiblio.org/mdma-release/http-prob.html

27. Touch, J. Heidemann, J. Obraczka, K. *Analysis of HTTP Performance.* USC/Information Sciences Institute. 1996. http://www.isi.edu/lsam/publications/http-perf/

28. W3C, Web Services Architecture Requirements, www.w3.org/TM/wsa-reqs#N100CB

29. The Apache XML Project. Xerces. http://xml.apache.org/xerces-j/

30. Java Technology and XML. http://java.sun.com/xml/javaxmlpack.html

31. Violleau, T. *Java™ Technology and XML Part 2: API Benchmarks.* 2002. http://developer.java.sun.com/developer/technicalArticles/xml/JavaTechandXML_part2/

32. Sosnoski, D M. A look at features and performance of XML document models in Java. September 2001. http://www-106.ibm.com/developerworks/xml/library/x-injava/index.html

33. Slominiski, A. *On Performance of Java XML Parsers*. www.cs.indiana.edu/~aslom/exxp/

34. Braden, R. (editor). *Requirements for Internet Hosts -- Communication Layers*. RFC 1122. October 1989. http://www.faqs.org/rfcs/rfc1122.html

35. Clarke, D. *Window and acknowledgement strategy in TCP*. RFC 813. July 1982. http://www.faqs.org/rfcs/rfc813.html

36. Allman et. al. *TCP Congestion Control*. RFC 2581. April 1999. http://www.faqs.org/rfcs/rfc2581.html

37. The Apache Organisation. The Axis Project. http://xml.apache.org/Axis

A. Appendix: Web Services

A.1 What is Web Services?

Web Services is basically a mean to establish business-to-business relationships (B2B) by publishing services and using established and accepted technologies such as XML, HTTP and TCP/IP to access the services.

The main vision for the future is to be able to automatically search, find and bind/connect business system either between different actors or between internal departments or systems.

In the future it is very plausible that the focus on business-to-business will expand to include the business-to-consumer area.

The Web Services concept works as follows (see figure 1.):

1. **Publish services.**
This includes the programmatically interfaces needed to access the Web Services. The interfaces are specified by the Web Services Description Language (WSDL), which is based on XML.

2. **Search/Find the service.**
The Web Services are published on an UDDI (Universal Description, Discovery, and Integration), which works as a directory service and repository for Web Services available.

3. **Bind the service.**
The requester utilizes SOAP (Simple Object Access Protocol) in order to connect to and use services of interest.

A.2 Possibilities and risks

We made a study of the market to identify possibilities and risks Web Services is believed to bring. The possibilities we identified were:

1. **Build/link systems inside organizations.** Web Services seems to be able to bridge the gap between the different systems in a flexible and relatively cheap way without introducing rigid measures.

2. **Build/link systems outside organization.** Businesses may connect their business systems to providers' and customers' systems in order to facilitate control over supplies and payments.

3. **Access information on any device.**
Web Service providers may offer to host personal or vital data and make it available on any device and thereby ensure availability.

4. **New business opportunities.**
Companies will be able to create and publish Web Services that provides services based on core competences.

The risks identified were:

1. **Lack of standards.**
All vendors have agreed on using TCP/IP, UDDI, WSDL and SOAP. Unfortunately there are some vital standards that are not agreed on for example security, integrity and transaction management.

2. **Lack of independent organisation.**
At this moment there is not a single organisation that promotes and protects Web Services standards that all major vendors supports. IBM and Microsoft among others (not SUN) have taken the first steps to remedy this in creating WS-I (Web Services Interoperability Organisation, www.ws-i.org).

3. **Bandwidth**
The SOAP protocol is based on the XML standard, which is text based. The SOAP messages have a large overhead. In combination with the fact that companies probably are going to out-source services that were provided from internal sources may push the bandwidth on the Internet to its limits. This will be a critical problem when messages must be transmitted quickly.

4. **Acceleration of hardware requirements.**
The increase in bandwidth need will also have impact on the hardware that are dependent on the bandwidth for example servers. One probable solution is to compress the XML-documents, which will increase the demand on the server CPUs. The time consuming parsing of XML documents will also increase the demands.

5. **Increased dependency on professional services.**
The Gartner Group indicates that professsional help is initially needed to make Web Services mainstream. This is probably due to the lack of best practises in combination with that there are more and more systems that need to be integrated and that the knowledge is not easily accessible.

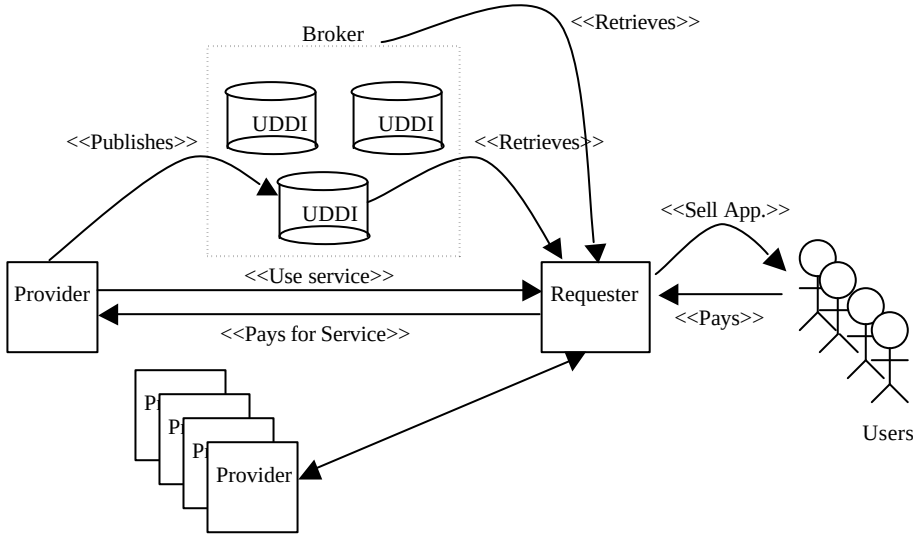


Figure 1. Web Services Model

B. Appendix: Prototype

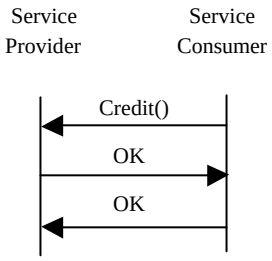
B.1 Description

The Web Service that we implemented was a simple charging system. There were five basic functions available for the service consumer. These were:

- Debit an account.
- Credit an account.
- Reserve an amount on an account.
- Remove an earlier reservation.
- Make a credit control on an account.

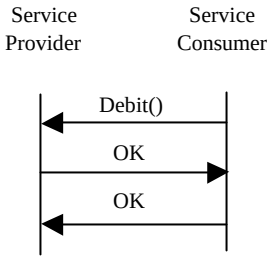
B.1.1 Credit

Credit deposits a specified amount of credits on a specified account. The behaviour of credit is:



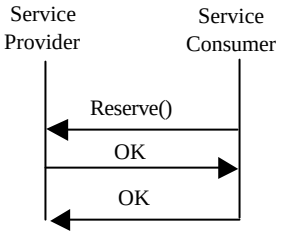
B.1.2 Debit

Debit withdraws a stated amount of credits from a specified account. The behaviour of debit is:



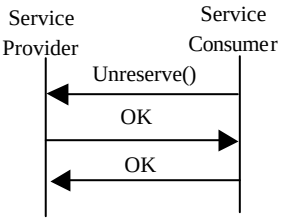
B.1.3 Reserve

Reserve reserves a specified amount of credits on an account. The amount will not be withdrawn until an associated debit call is performed. The debit call contains a reference to the reservation. The behaviour is:



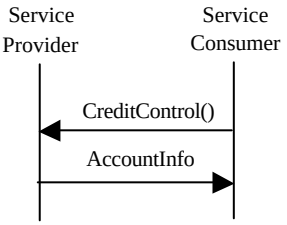
B.1.4 Unreserve

Unreserve removes a prior reservation and restores the account balance. The behaviour is:



B.1.5 Credit Control

The credit control returns the balance of an account. The behaviour is:



B.2 Usage of the functions

We had to pair some of the function calls when using them from the client and load generators. They were:

- Debit with prior reservation.
  A reservation followed by a debit.
- Release of a reservation.
  A reservation followed by a release.

The ack (OK) from the client side was implemented by a confirm function. This mean that the number of SOAP calls differed between the function calls.

Function	#SOAP calls
Credit Control	1
DirectDebit	2
Credit	2
ReservationRelease	4
ReservationDebit	4

C. Appendix: SOAP CALLS

C.1 Reserve

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:EnvelopeSOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Body>
        <reserve>
                <id xsi:type="xsd:int">2</id>
                 <am xsi:type="xsd:int">520</am>
        </reserve>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

C.2 DebitReservation

A debit that follows a prior reservation.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
        <debitReservation>
                 <id xsi:type="xsd:int">2</id>
                <am xsi:type="xsd:int">520</am>
                <res xsi:type="xsd:int">12407</res>
        </debitReservation>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

C.3 Directdebit

Debit without prior reservation.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
        <directDebit>
                <id xsi:type="xsd:int">8</id>
                <am xsi:type="xsd:int">438</am>
        </directDebit>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

C.4 Creditcontrol

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
        <creditControl>
                <id xsi:type="xsd:int">6</id>
        </creditControl>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

C.5 Credit

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
        <credit>
                <id xsi:type="xsd:int">8</id>
                <am xsi:type="xsd:int">147</am>
        </credit>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

C.6 ReleaseReservation

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
        <releaseRes>
                <res xsi:type="xsd:int">12418</res>
        </releaseRes>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

C.7 ConfirmAction

This is the ack from the client side.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
        <confirmAction>
                <cn xsi:type="xsd:int">76430</cn>
        </confirmAction>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```