7 Performance measurements

In this Chapter, we present our performance measurements. In the first measurement, we determine the impact of the Web service overhead on web servers and mobile clients. We present our solutions that improve the performance of mobile Web service access. In the second measurement, we demonstrate the advantages of the WSB when the lookup and selection of a Web service offer from many competing offers is processed at runtime. In the third measurement, we demonstrate the performance gain achieved through our QoSProxy that maps clients' QoS requirements of transport network at runtime. In the last measurement, we demonstrate the advantage of our WS-QoS framework that prevents Web service server from overload through adaptive WS-QoS offers.

7.1 Performance impact of Web service overhead

In contrast to traditional web interaction, Web services incorporate some additional overhead. In particular, due to the usage of XML, requests and replies are larger compared to traditional web interactions and the need for parsing the XML code in the requests adds additional server overhead. We present a typical web application that requires the transmission of four to five times more bytes if implemented as a Web service compared to the same service implemented as a traditional dynamic program, in our case as an Active Server Page (ASP) application. Therefore, compression of Web service interactions is attractive. It is easy to imagine that in the future clients using mobile devices will generate a large percentage of all Web service requests. Although the computing power of handheld devices is increasing rapidly the CPU time required for decompression

might eliminate the benefits of compression for these types of devices. In this section, we present experiments that quantify the decompression overhead on a handheld computing device with constraint processing capabilities. As expected, mobile clients benefit from compression when the available bandwidth is scarce, for example when the client is connected via GPRS. But even when resource-constrained devices have better connectivity, the performance loss caused by decompression is almost negligible. Note that mobile clients also might prefer compressed responses since they are often charged by volume rather than by connection time, e.g. in the case of GPRS [49].

A lightly loaded server can afford the extra cost of compressing responses. We present measurements that show that the throughput of a heavily loaded server can decrease substantially when it is required to compress Web service responses. At the same time the response times experienced by the clients increase. We propose a simple scheme that allows clients to specify whether they want to receive data compressed when requesting a Web service. Depending on the current server load, the server compresses only the requests of the clients that required such a service. We present experiments that demonstrate that this approach works as expected and that both servers and clients with poor connectivity benefit during high server demand.

7.1.1 Web Service overhead

Since both SOAP and WSDL are XML-based, XML messages have to be parsed on both the server and the client side and proxies have to be generated on the client side before any communication can take place. The XML parsing at runtime requires additional processing time, which may result in longer response time of the server hosting Web services.

In order to demonstrate the quantity of the additional bytes Web services generate for transfer, we have implemented the same "service" both as a traditional dynamic program, in our case as an Active Server Page (ASP) application, and as a Web service. The implemented application is an electronic book inventory system. The clients send the ISBN of a book to the server and the server returns information about the book such as the title, author name, price, and so on.

When sending small amounts of content using SOAP on HTTP, such as sending an ISBN for querying book information, the major part of the entire conversation will consist of HTTP headers, SOAP headers including the XML schema as well as brackets. In our case, the Web service accepts the ISBN of a book as input parameter and returns the book information in form of a dataset. The actual content of both the request and the response consists of a total of 589 bytes, thereof 10 bytes for the ISBN and the rest for the information about the book. But more than 3900 bytes have to be sent and received for the entire conversation. Figure 30 depicts the bytes on the wire for the actual content and the overhead when it is transmitted as HTML or XML. The disproportion is not as big for traditional web interaction with HTML. The total amount of the

request and response for transferring the same information value is about 1200 bytes.

The overhead of the Web service stems mainly from the usage of XML producing human readable text and is employed since the interoperability with other Web services and applications is essential [50]. Others have compared XML's way of representing data with binary encodings. They quantify the overhead as 400% [41].

7.1.2 A dynamic approach for reduction of Web service responses

The growth of the Web service message size, which results in higher data transmission time, creates a critical problem for delay sensitive applications. One way to achieve a compact and efficient representation is to compress XML − especially when the CPU overhead required for compression is less than the network latency [50]. Compression is both useful for clients that are poorly connected as well as for clients that are charged by volume and not by connection time by their providers. The latter group comprises mobile users connected with handheld devices such as people accessing a service via GPRS. This group of users is expected to increase rapidly in the next years. However, the Web service application on the server does not have any information about the delay, for example the current round trip time estimated by TCP, and about the available bandwidth between client and server.

Thus, we have decided to let the Web service users specify whether they want the response compressed. Mobile users usually know if they are charged by volume and often know how they are connected. Thus, it seems reasonable to let them decide whether they want the server to compress the response. In our current design we let users (or the clients' software) decide between three options:

- Do not compress the response
- Compress the response
- Compress the response if possible

If users choose the last option, the server is free to choose what the server considers best. To give users an incentive to choose this option, commercial Web service providers could decide to charge a lower price for this option. The choice of the users is reflected in the request. When the last option is chosen and the server demand is low, the server compresses the responses to all clients that have asked for compressed replies and to those clients that have not specified a preference. During high server demand, the server compresses only responses to clients that have asked for compressed data. Since compression requires mainly CPU time, we regard the server demand as high when the CPU utilization of the server exceeds a certain threshold.

Note that in this approach, the server can still become overloaded. Mechanisms for server overload protection have been studied elsewhere [18].

7.1.3 Experiments

In this section we describe our experimental setup and the application we have implemented. We introduce the experiments and the corresponding results in the next section.

7.1.3.1 Testbed

Our testbed consists of three 1GHz Pentium III machines with 256 MB memory, a Pentium III laptop with 700MHz and 384 MB RAM and an iPAQ Pocket PC 3970 running Windows CE 3.0 with a 400MHz XScale Processor (see Figure 46). Our Internet server is a standard Internet Information Server version 5.0 with the default configuration. The other two Pentiums run Linux. One is running the sclient traffic generator (see below) and the other runs NIST Net [51]. NIST Net emulates a wide variety of network conditions such as low bandwidth and large round trip times. The iPAQ handheld device is connected to the server via the laptop and the machine running NIST Net.

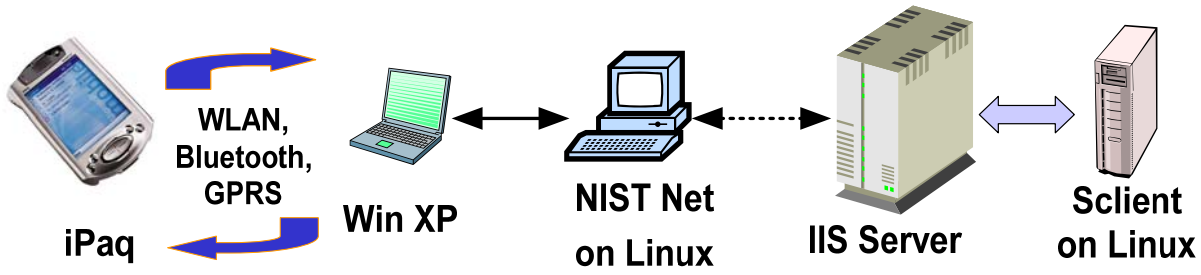

Figure 46. Testbed: measurement of Web service overhead

For background load generation, we use the sclient traffic generator [52]. Sclient is able to generate client request rates that exceed the capacity of the Internet server. This is done by aborting requests that do not establish a connection to the server in a specified amount of time. This timeout is set to 50 milliseconds in our experiments. The exact timeout value does not impact the results, as long as it is chosen small enough to avoid that TCP SYNs dropped by the server are retransmitted. However, the larger the value, the higher the risk that the request generator runs out of socket buffers. Sclient does not take the aborted requests into account when calculating the average response time.

In order to emulate poorly connected clients we use the host running NISTNet to add additional delays and decrease the available bandwidth. We emulate a GRPS network based on the results from measurements in a real GPRS network by Chakravorty et al. [53]. They measure the delay on the uplink to about 500 ms and on the downlink to about 800ms. We do not take the variations into account in our experiments. For the bandwidth the theoretical values are 40.2 kbit/s on the downlink and 13.4 kbit/s on the uplink meaning that the mobile device listens simultaneously on three downlink channels while sending on one uplink channel as many mobile telephones do. The values measured by Chakravorty vary substantially based on the current network conditions with the best conditions coming very close to the maximum values. We use both the theoretical values as the best case from the clients' point of view as well as the values they measured

when link conditions were poor. The latter are 12.8 kbit/s on the downlink and 4 kbit/s on the uplink.

7.1.3.2 Test application

The implemented application is a modification of our electronic book inventory system described in Section 7.1.1 that returns responses of different sizes depending on the request. The additional requests are for a small Hello World service, and more detailed ("heavy") information (including more detailed information, user ratings and hints on similar books etc.) about one, two, three and five books. The corresponding sizes of the SOAP body in both compressed and uncompressed form are shown in Table 4. Note that additional bytes are needed for the SOAP header (approx. 150 Bytes), the SOAP envelope (approx. 200 Bytes) and the HTTP header. We only compress the SOAP body in our experiments. We see that except for the small response the compression factor is about three.

The Web service running on a Microsoft IIS is implemented with the .NET framework 1.1 beta [54]. The Web service client is implemented with the .NET compact framework and is deployed on the iPAQ. Since SOAP is used for the client server interaction, we have extended the SOAP headers with the SOAPExtension class of the .NET framework class library in order to modify (on the client side) and inspect (on the server side) SOAP messages. The client sets a parameter instructing the server either to compress or not to compress the data part of the SOAP response or to let the server decide by it. For compression we use the SharpZipLib library [55], but other compression algorithms or strategies may be used [56].

Table 4. Response size without and with compression and decompression

	Original data size (byte)	Data size after compression (byte)	Compression time on server (ms)	Decompression time on client (ms)
"Hello world!" response	209	256	1	41
Lite information for 1 book	3366	1390	12	200
Heavy information for 1 book	16055	6038	24	497
Heavy information for 2 books	28153	10222	36	747
Heavy information for 3 books	36049	12350	79	877
Heavy information for 5 books	50205	15470	89	1122

7.1.4 Experimental results

In this section, we present three different sets of experiments. In the first subsection we evaluate the performance of Web services for different wireless networks. These experiments show that mobile clients can gain from compression when their connectivity is poor. However, compression requires server resources and, therefore, we quantify the server overhead for compression. In the second subsection we demonstrate that compression can degrade server performance severely. The experiments in the final subsection validate our dynamic approach, where during high server load the server compresses only responses for clients that have indicated that they want the server to compress the response.

7.1.4.1 Web service performance for handheld devices

Due to the large message sizes of Web services we assume that compressing Web service responses is useful. However, the cost of decompression on resource-constrained devices may invalidate this assumption.

In the following we investigate the performance of Web services on handheld devices when connected to different networks, namely with 802.11b wireless LAN, Bluetooth as well as GPRS networks with both good and poor connectivity, as described in 7.1.3.1. The server in this scenario is lightly loaded because the client is the single user. Table 4 shows the compression times on the server and the decompression times on the client for different data sizes. The results indicate that the compression time on the server is much lower than the decompression time on the resource-constrained handheld device. On the iPAQ,

the decompression time is more than one second for the largest response while compressing on the server requires less than 90 ms.

Figure 47 to Figure 50 show the experimental results with different network connections. The x-axis shows the original message size of Web service responses ranging from 0 to 50000 bytes. The y-axis is the service time in millisecond. The service time denotes the time interval between the moment the client requests the service, e.g. by clicking on a button and the moment the client has received and processed the result. We expect that mobile clients will benefit from compression when the bandwidth is scarce but experience small performance degradation when the available bandwidth is larger, i.e. when the service is requested over the wireless LAN or Bluetooth.

Since the available bandwidth in a wireless LAN is higher than in a Bluetooth network, we expect the service time in the wireless LAN to be lower than the service time over Bluetooth. Indeed, as shown in Figure 47 and Figure 48 the service time in the wireless LAN scenario is about 2 seconds shorter than in the Bluetooth scenario for all message sizes. There is no significant difference between service time for compression and no compression when the iPAQ is connected via a wireless LAN or Bluetooth network. At its maximum, the time difference for receiving compressed or non-compressed responses is about 4% for the largest request over the wireless LAN. This shows that the overhead caused by compression is not severe in these scenarios.

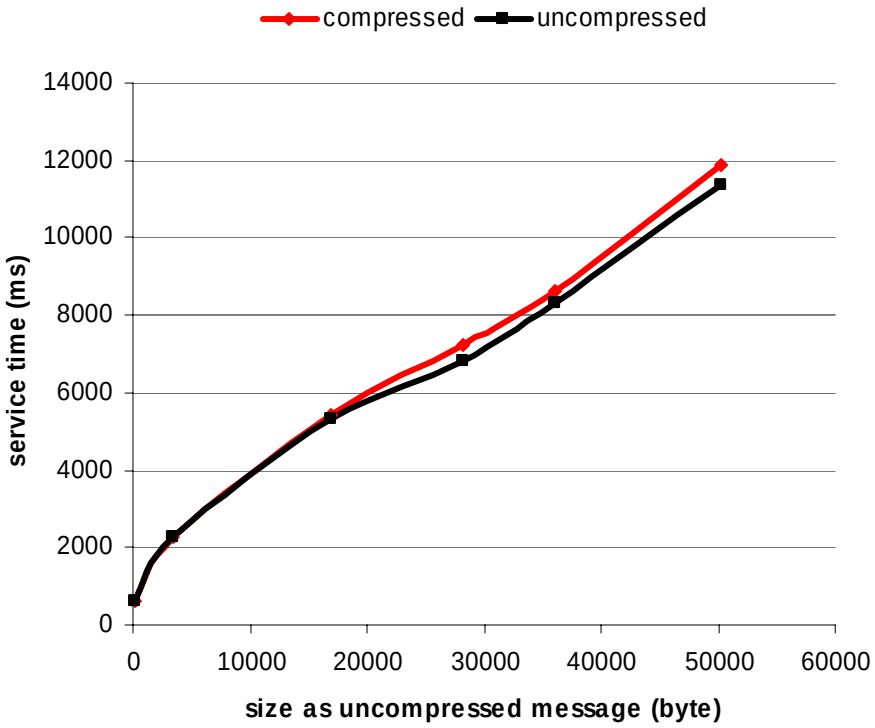

Figure 47. iPAQ service time over wireless LAN

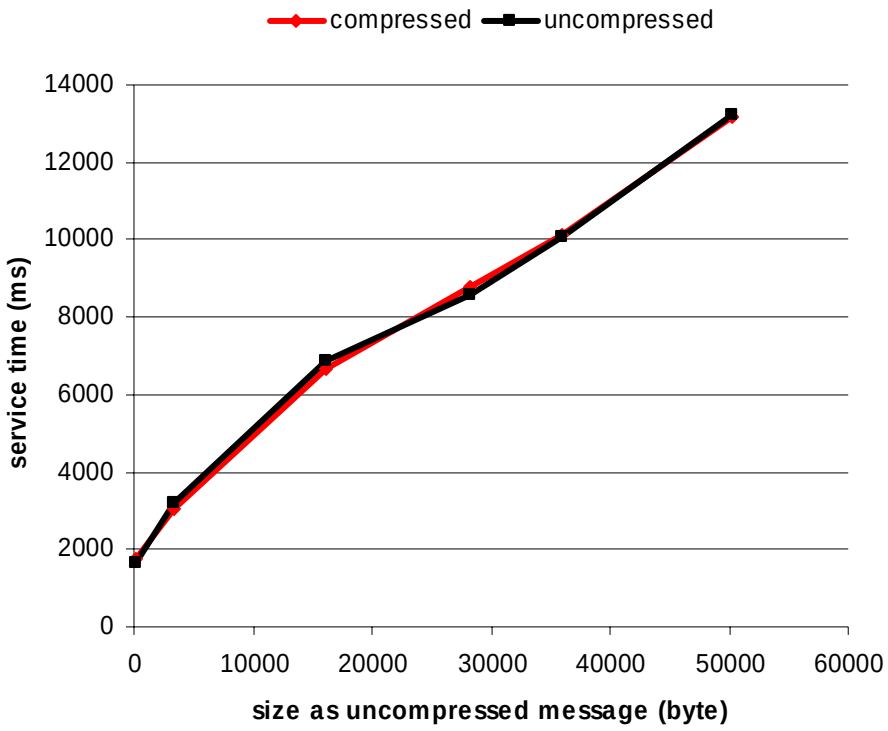

Figure 48. iPAQ service time over Bluetooth

When the personal digital assistant (PDA) iPAQ is connected via a low bandwidth network such as GPRS, the service time is lower for larger response sizes when the response is compressed. This means that the benefit of compression is higher than the cost of decompressing the response. As Figure 50 shows, when connectivity is poor the service time is halved when compressing the largest response.

These experiments demonstrate that compressing Web service responses is useful when the available bandwidth is scarce even for clients using resource-constrained devices.

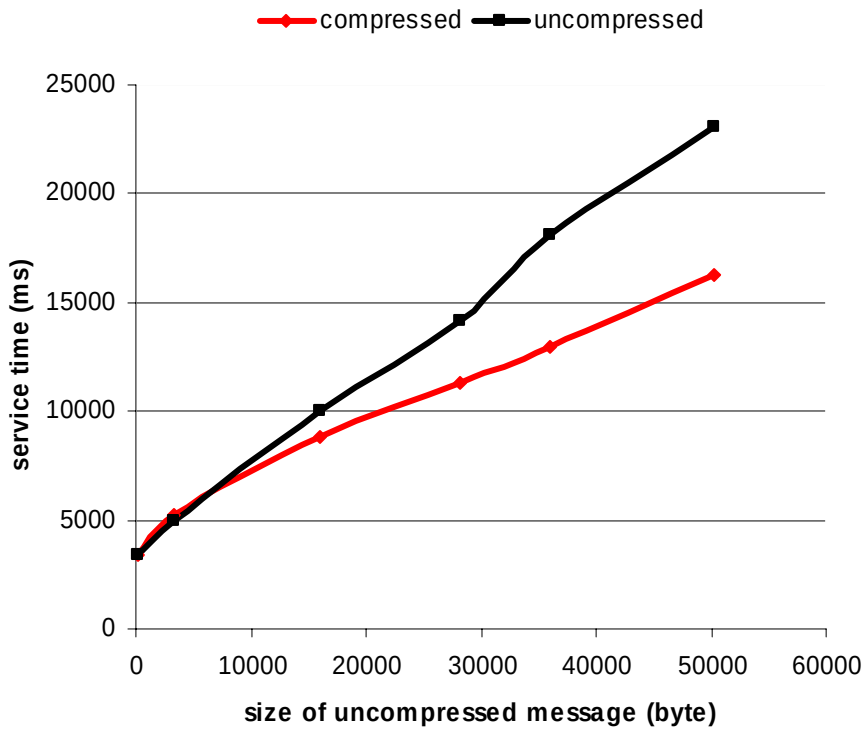

Figure 49. iPAQ service time over emulated GPRS network

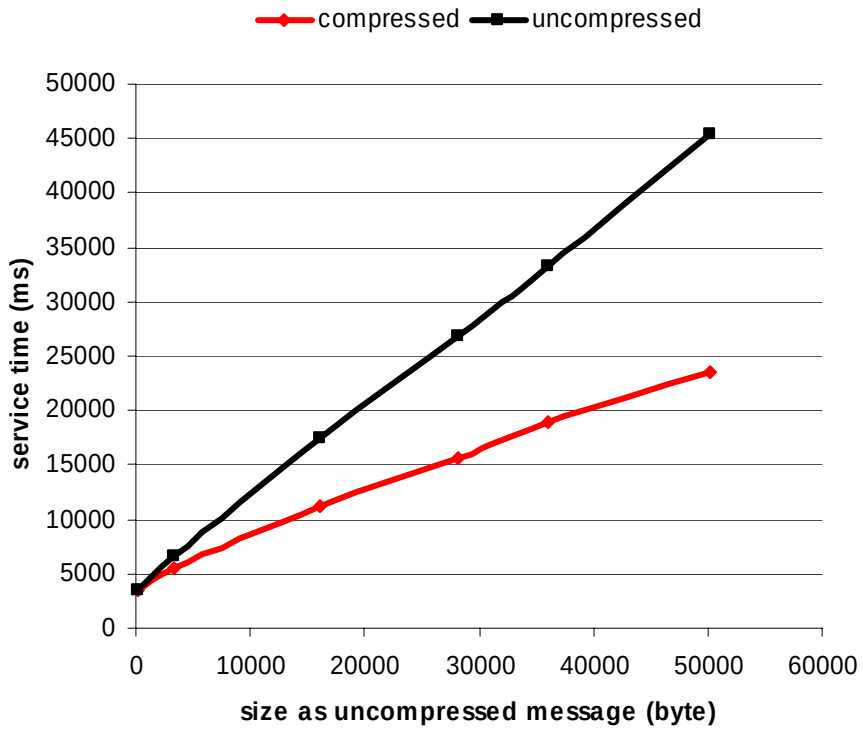

Figure 50. iPAQ service time over emulated GPRS with poor connectivity

7.1.4.2 Impact of compression on server performance

Compression requires also CPU time at the server. In this section, we evaluate the impact of compression on server performance. We use the sclient workload

generator to sustain a certain request rate independent of the load on the server. The traffic generator makes requests at a certain rate for the electronic book Web service described in the previous section.

The test scenario increases the request rate across runs and conducts three runs for each request rate with each of the runs lasting for three minutes. We measure the average throughput and response time. We expect that the response time will be quite low when the request rate is below the capacity of the server, no matter whether compression is used or not. However, using compression we expect that the server will reach its maximum capacity at a lower request rate. When the request rate is above the capacity of the server, the response time will increase rapidly due to the waiting time that requests spend queuing before they can be processed. Also, the throughput will increase with the request rate until the maximum server capacity is reached. When the request rate is higher than the capacity of the server, the throughput will not increase anymore. During severe overload the throughput might even decrease since CPU time is wasted on requests that cannot be processed and are eventually discarded [52].

For compression we use the SharpZipLib library. This way, we can reduce the overall number of bytes for the "lite information" scenario from more than four kbyte to around two kbyte (refer to Table 4). Without compression, three TCP segments are needed for the response while the compressed response fits into two TCP segments. Note that in our experiments the client, i.e. the traffic generator, is not required to decompress or to process the received data in some other way.

Due to the additional CPU time the server spends on compressing data, we assume that the response time increases and the throughput (measured in connections per second) decreases when the response is compressed. Figure 51 depicts that it is indeed the case. The response time during overload is about three seconds higher and the throughput is about 45 conn/sec lower when compression is used. Figure 52 shows that the maximum server throughput decreases from about 135 conn/sec to 90 conn/sec when compression is used. These experiments show that when a server compresses all replies, the maximum server throughput decreases substantially and the response time experienced by the clients is affected negatively. This gives reason for our approach that is based on the assumption that servers should only compress replies to clients that can benefit from compression.

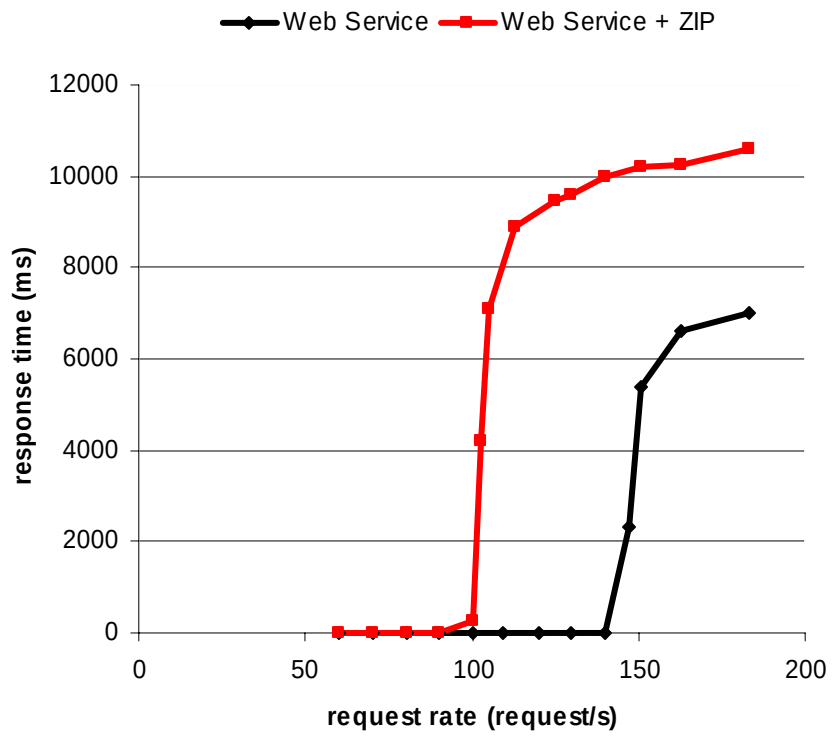

Figure 51. Comparison of response time with and without compressing the response

Figure 52. Comparison of throughput with and without compressing the response

7.1.4.3 Dynamic server compression

The first experiments have shown that mobile clients with poor connectivity benefit from compression while the experiments in the previous section have demonstrated that compression reduces server performance during high demand. In the experiment in this section, we investigate if the dynamic compression approach described in last subsection is able to give us the best of both worlds, i.e. compressed data for clients that wish to receive compressed data while achieving high server throughput.

The next experiment compares the server performance when

- all responses are compressed,
- no responses are compressed and
- the server decides which responses to compress.

As in the experiment in previous section, we use sclient to request the information on a book. Sclient first requests that all responses are to be compressed, then that no responses are compressed and finally 50% of the responses to be uncompressed and for the other 50%, the server is asked to decide. In the latter setting, which we call dynamic, when no compression indication is given for a request, the server compresses the response when the CPU utilization of the server is below a threshold of 80%. If the CPU utilization is higher than 80% it does not compress such a response in order to save processing time. Using this approach, the performance should be almost as high as without compression. Figure 53 and Figure 54 show the response time and the throughput, respectively. As expected, using the dynamic approach the server performance is almost as high as when the server does not use compression. Further inspection of the results reveals that the server compresses all responses it is allowed to compress (50% of the requests that have indicated that they do not have any preferences) until a request rate of 60 request/s is reached. When the request rate reaches 120 request/s only 20% of the requests without preference indication are compressed while no requests are compressed at request rates larger than 140 request/s.

However, the performance gap should be smaller, i.e. the difference between the dynamic approach and no compression should be almost nothing, since the only extra task required from the server is to check the current CPU utilization when processing a request that has not indicated any preference. This indicates that this task is more expensive than one would expect.

In the next experiments we want to validate that a client with a poor connection to the server may indeed benefit from this approach. In these experiments, the sclient varies the request rate and requests the "lite information for one book" for two different scenarios. In the first scenario sclient requests compressed responses while in the second scenario sclient requests 50% of the responses as compressed and 50% of the responses without compression preference. In both

scenarios, the iPAQ requests the "heavy information for one book". The mobile client is connected via the emulated GPRS network to the server.

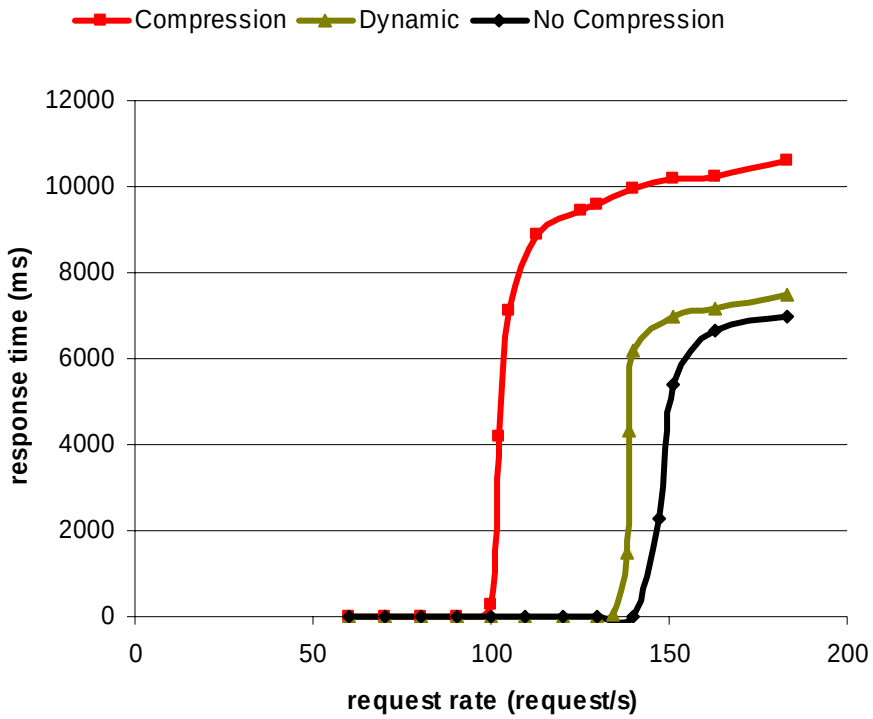

Figure 53. Response time of the dynamic approach

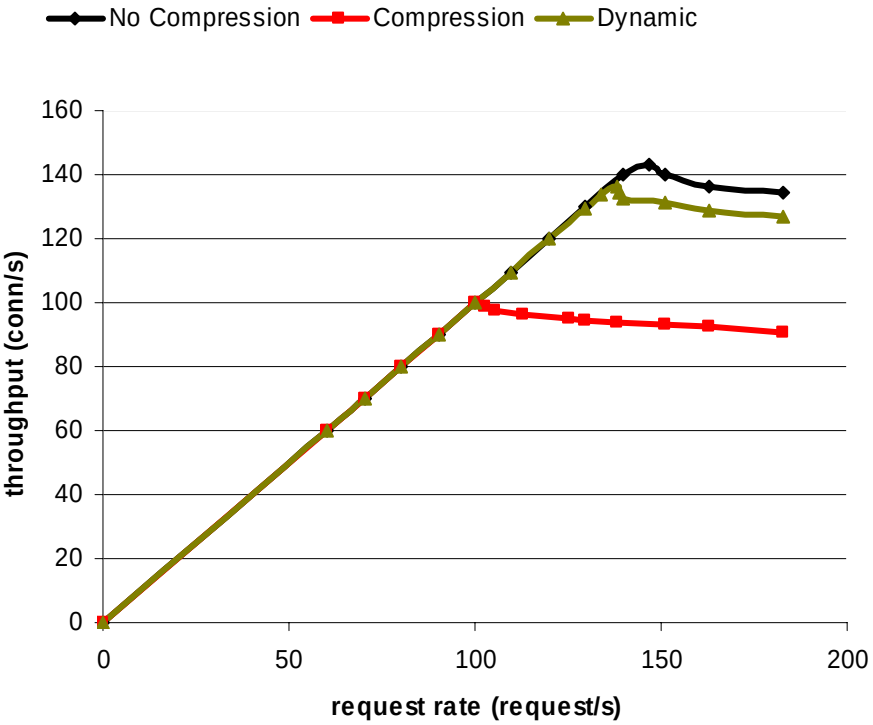

Figure 54. Throughput of the dynamic approach

The results shown in Table 5 indicate that our dynamic approach is beneficial for both the server and for mobile clients with poor connectivity. As expected for some sclient request rates, the service time experienced by the mobile client is much better when the server uses the dynamic approach, namely when the sclient request rates are between 100 and 140 requests/s, which corresponds to the results in Figure 53. This is the request range where the CPU time required for processing would overload the server and would degrade performance but the server still performs well when it does not need to compress the responses.

Table 5. Impact of the dynamic approach on the response time of the mobile client

Sclient rate (request/s)	**80**	**100**	**120**	**140**	**160**
All compressed (ms)	9985	10495	16492	17104	17176
Dynamic (ms)	9370	9850	10468	10538	17131

7.2 Performance measurement of the Web service broker

In this section, we discuss the performance of the WSB. In the first test run, the WSB compares available service offers inside the application codes, while a data base containing the available offer information is applied in the second test run. The WSB achieves a much better result in the second scenario.

7.2.1 Testbed and test application

Our testbed for both scenarios consists of three 3GHz Pentium IV HT machines with 1GB RAM as shown in Figure 55. Lab1 runs Sclient on Debian Linux with kernel 2.6.9. The traffic generation tool Sclient simulates Web service clients in order to stress the WSB. The WSB on Lab2 runs Windows server 2003 with standard configuration. The Web services providers are hosted on Lab3 running Windows server 2003 with standard configuration. Four Web service providers with 22 services and 27 offers run on Lab3.

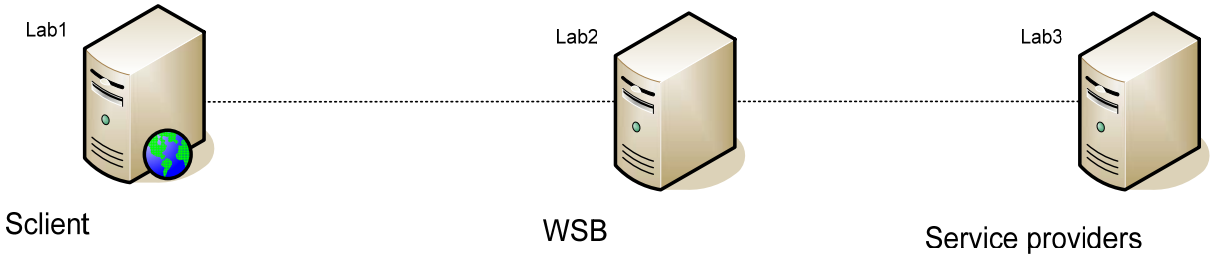

Figure 55. Testbed: measurements of WSB

7.2.2 WSB without database support

The Sclient simulates clients requests' sent to the WSB at different request rates. The WSB has already available offers in its local cache, so it needs not to contact any UDDI or service providers to update its local cache. The information of the available offers is hold in the random access memory (RAM). Figure 56 shows the response time of the WSB at request rates ranging from 10 to 300 requests per second.

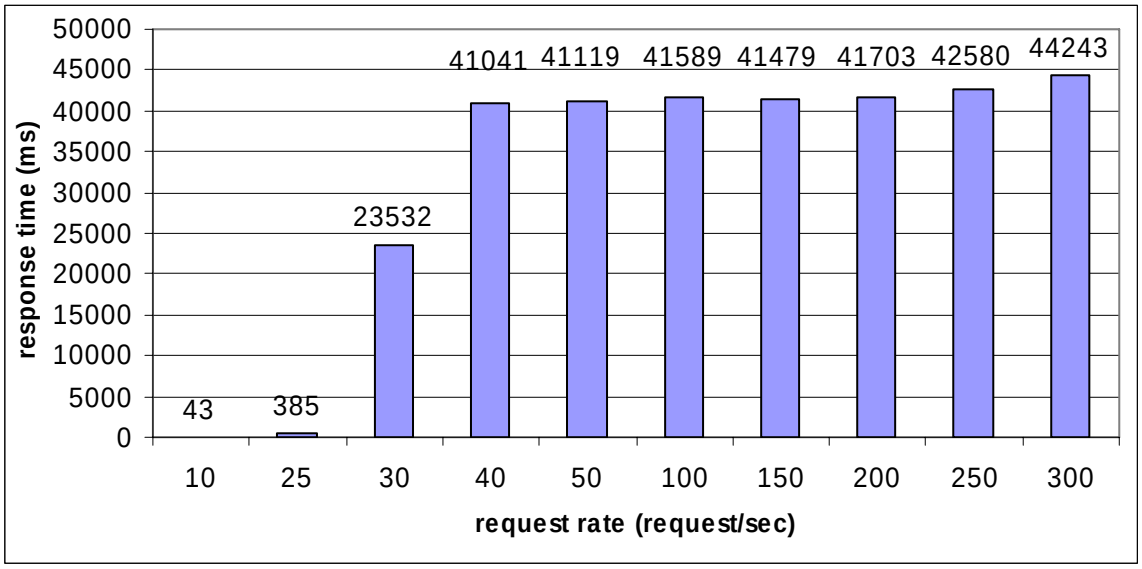

Figure 56. Response time of the WSB at different request rate

The response time is short at low request rate until 30 requests per second. From this point the response time increases fast and remains constantly on a high level.

Figure 57 shows the throughput of the WSB. The maximum throughput is approximately 25 conn/s. The throughput does increase with the request rate, since the CPU usage is almost 100% at the request rate of 30 conn/s.

The reason of the poor performance of the WSB is due to its prototypic implementation. The WSB holds information about available offers of different service providers in RAM resulting in low-performance when comparison is performed. In the next subsection, we show the performance improvement when a database is applied.

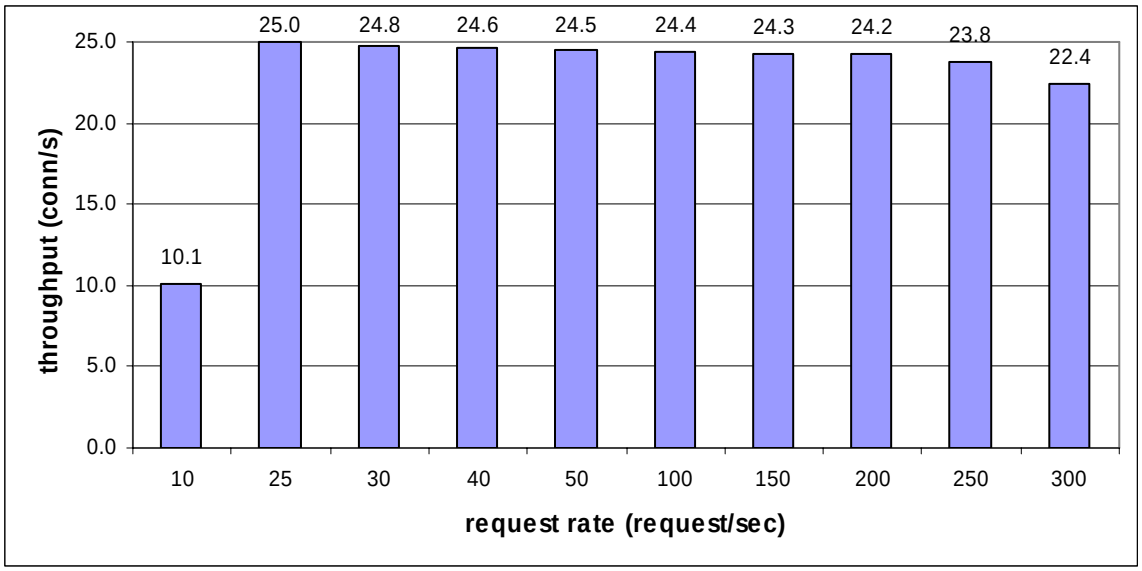

Figure 57. Throughput of the WSB at different request rate

7.2.3 WSB with database support

Due to the low-performance of the performance measurement described in the previous subsection, we have reimplemented the WSB with database support. We applied the Microsoft SQL Server 2000 with default configuration. As Figure 58 and Figure 59 show the performance of the WSB can be improved significantly.

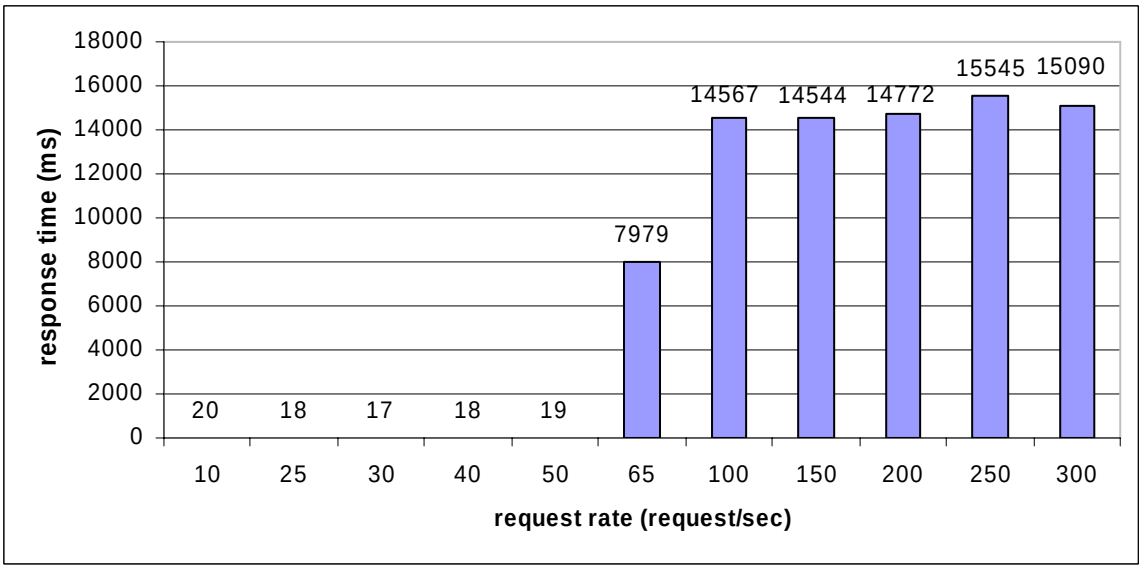

Figure 58. Response time of the WSB with database support at different request rate

The response time is lower than 21ms for the request rate until 50 requests per second. With higher request rate the response time increases up to approximately 15 seconds and remains stable, which is much better than the scenario before.

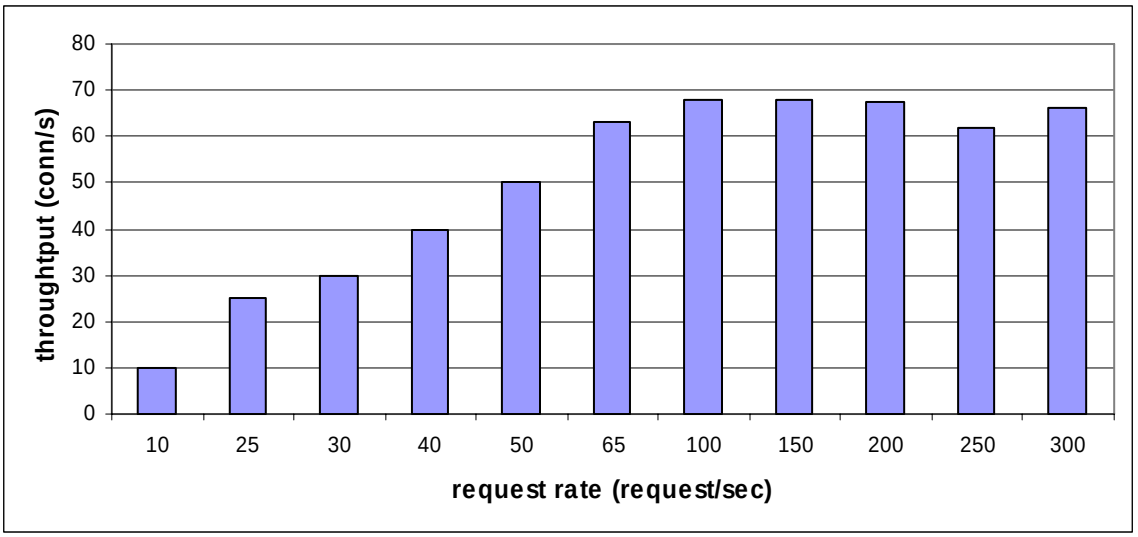

Figure 59. Throughput of the WSB with database support at different request rate

The throughput increases with the increasing request rate up to 65 requests per second linearly. The maximum throughput is reached at the request rate of 100 requests per second. The reason is that the CPU usage is fast 100% at this request rate, as depicted in Figure 60.

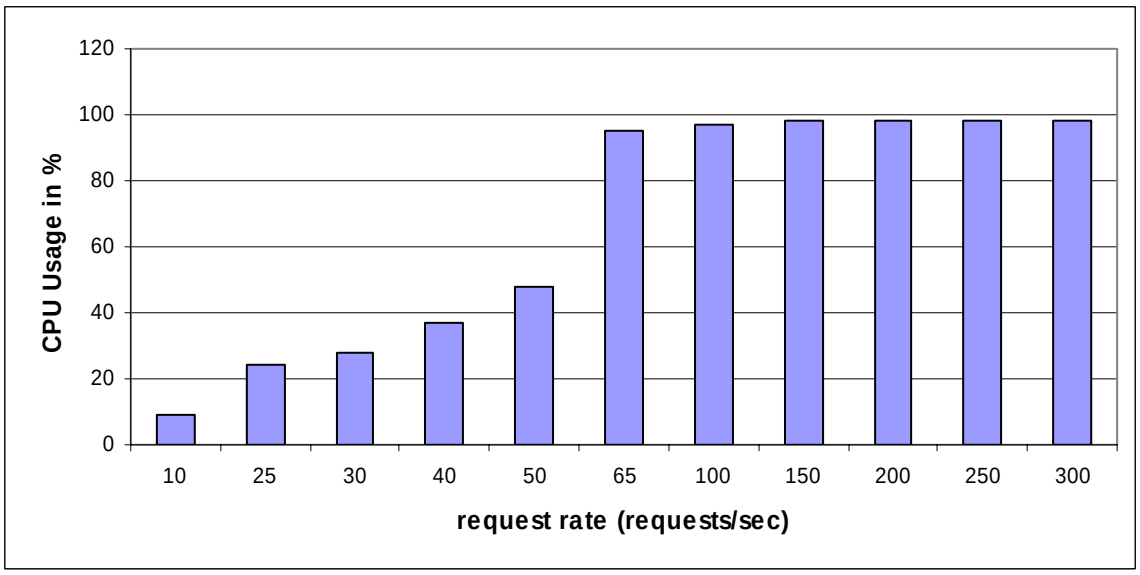

Figure 60. The CPU usage of the server hosting the WSB

Note that the implementation with the database support is still a prototype demonstrating the feasibility of the WSB. One can implement an own strategy for the WSB. The most important issue is that the WSB is located outside a UDDI server making the usage of the WSB more flexible and independent of any UDDIs.

7.3 Performance measurement of the QoSProxy

In this section, we discuss our testbed, test application, experiments and the corresponding results of the QoSProxy. The goal of this measurement is to certify the functionality and the performance of the QoSProxy. Since it consumes resources, it is important to know if the QoSProxy contributes to the overall performance of Web service communication.

7.3.1 Testbed and test application

Our testbed consists of five 3GHz Pentium IV HT machines with 1GB RAM as shown in Figure 61. Lab7 and Lab8 run Windows Server 2003 with standard IIS version 6.0 hosting our Web service applications. The other three machines (Lab3, Lab5, and Lab6) run Debian Linux with kernel 2.6.9. We use Lab3 and Lab8 to measure the performance of our Web services and Lab5 and Lab7 to generate background traffic in order to stress the network.

Lab6 is configured as a DiffServ router simulating a DiffServ network. All traffic between the clients and servers passes it. The DiffServ router uses the traffic control features of the Linux kernel. To shape the traffic according to our requirements we attach the Hierarchical Token Bucket queuing discipline to the network interface. The maximum bandwidth in the simulated network is limited to 1024 kilobyte/s. We define only two traffic classes: Assured Forwarding (AF) and Best Effort (BE) for our measurements. For the first one we reserve 30% of the available bandwidth, for the second one we reserve 70%. If there is no BE traffic the AF traffic may use up to 100% of the available bandwidth. All forwarded IP packets are investigated on the existence of DSCP marks. Packets marked with DSCP value 0x0A are assigned to the class AF, all other are assigned as default to the class BE.

The implemented Web service application is a simple electronic book inventory system. The client sends the ISBN of a book to the server and the server returns information about the book such as the title, author name, price and so on. The application applies the WS-QoS framework in order to define the QoS metrics for the transportQoSPriorities.

The application runs on Lab8 with the IIS. A QoSProxy marking the DSCP of all outgoing IP packets runs on the same machine. The DSCP values are marked according to the WS-QoS definition located in the SOAP headers. The size of the SOAP message with the book details and the SOAP headers is 8830 bytes. Note that additional bytes are needed for the HTTP header (approx. 270 bytes).

Another Web service application without any QoS issues runs on Lab7.

Lab3 and Lab5 run sclient traffic generator. The timeout is set to 50 ms in our experiments. Lab3 sends SOAP requests with QoS requirements in the SOAP header to the Web service running on Lab8 while Lab5 sends the SOAP requests to Lab7 without any QoS issues.

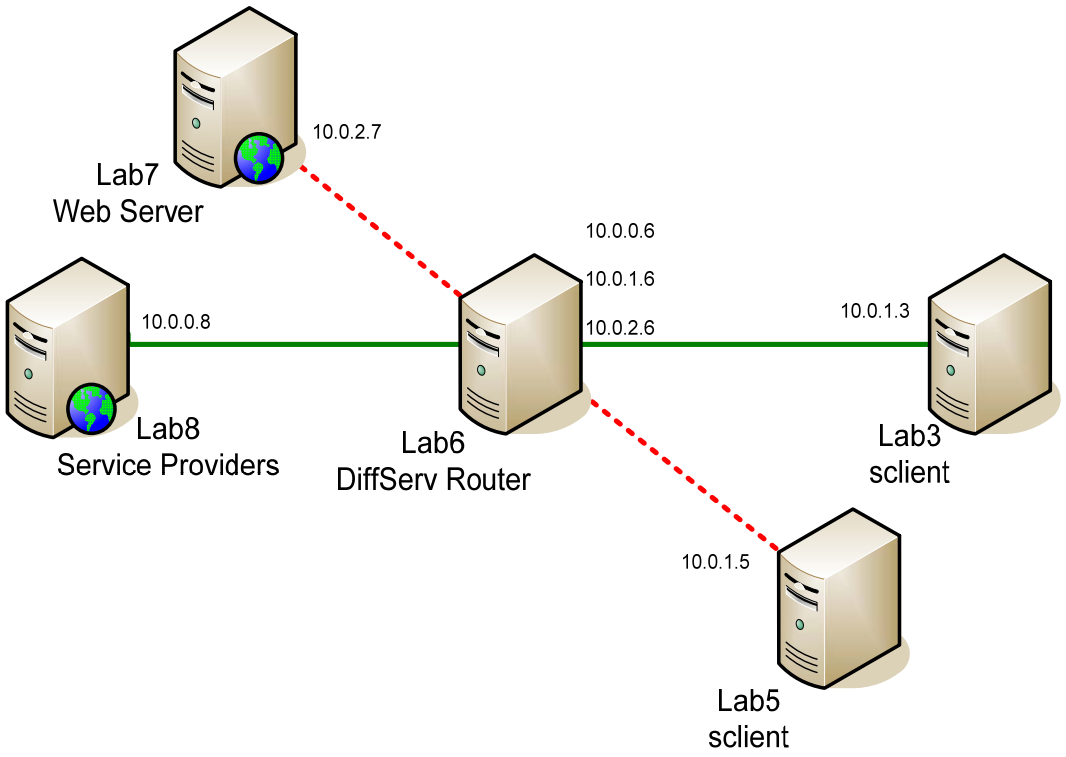

Figure 61. Testbed: performance measurement of QoSProxy

7.3.2 Test results

In this section, we present three sets of experiments. In the first subsection, we evaluate the performance impact of our prototypically implemented QoSProxy on the Web service. In the second subsection we evaluate the total performance gains by applying our QoSProxy when the (DiffServ) network is on high demand. The experiments in the last subsection demonstrate the theoretically achievable performance gains by applying an ideal QoSProxy.

7.3.2.1 Performance impact of the QoSProxy

We assume that the mapping of the QoS requirements results in an overall performance gain in a Web service communication process, especially when the network and the server are overloaded. However, the additional cost of the QoSProxy may invalidate this assumption.

In the following we investigate the impact of the QoSProxy on the server performance. We use the sclient traffic generator on Lab3 to generate a certain request rate on the server. The traffic generator makes requests for the electronic book Web service hosted on Lab8 as described in the previous section. The DiffServ router on Lab6 is disabled and no traffic shaping is performed in these test series.

In this test scenario the request rate is increased across runs and three runs are conducted for each request rate with each of the runs lasting for thirty seconds. During the first experiment we do not use the QoSProxy and do not overload both the network and the IIS. In the second experiment the QoSProxy is active and

processes the SOAP responses from the Web service and marks the outgoing IP packets.

In both cases we measure the average throughput of the Web service (response per second).

We expect that the overall performance of the web server is lower when the QoSProxy is applied. At low request rates there should be no significant difference between the two scenarios. However at high request rates the throughput should decrease when the QoSProxy is used.

Figure 62 shows the results of our experiments. The x-axis represents the request rates ranging from 20 to 70 requests per second. The y-axis shows the resulting throughput of the Web service.

Up to the request rate of 50 requests per second the throughput of the web server is equal in both cases. As expected, the QoSProxy is overloaded when more than 50 requests per second are submitted and the throughput is getting worse.

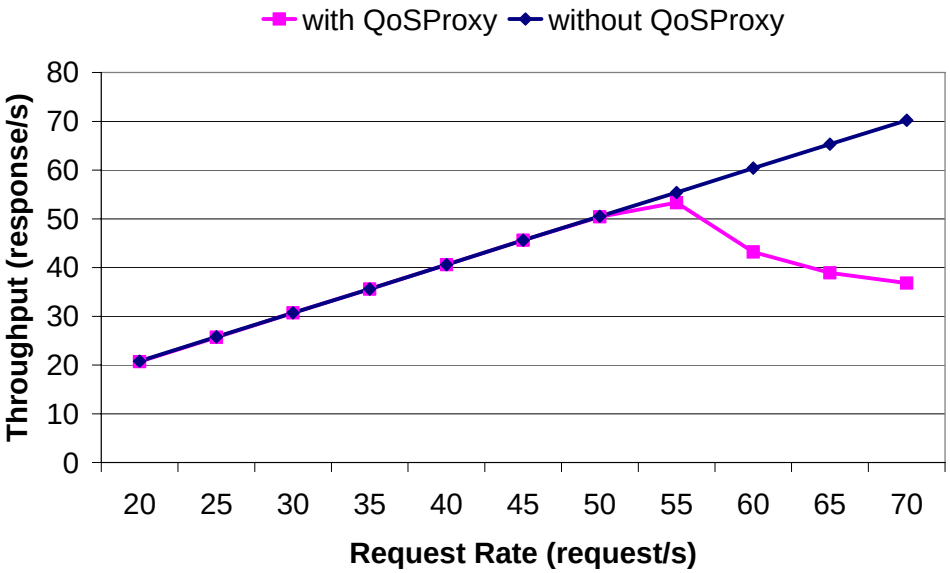

Figure 62. Performance impact of QoSProxy on the web server

There are many reasons for the negative impact of the QoSProxy on the server performance:

- The QoSProxy has to cache server SOAP messages, to parse, and evaluate them in order to find the WS-QoS-specific parameters.
- It has to mark all outgoing IP packets according to the mapping table.
- The QoSProxy has to manage the TCP/IP connections between the clients and the server. At high request rates multiple TCP connections and more dedicated threads have to be maintained simultaneously.

All these tasks result in performance loss.

7.3.2.2 Total Web service performance improvement due to QoSProxy

In the last subsection we have shown that the QoSProxy has performance impact on Web services. We assume that the use of the QoSProxy results in an overall performance gains in case of overloaded (DiffServ) network and Web service servers.

In our test scenario we use Lab5 and Lab7 to generate background best effort traffic in order to overload the DiffServ router and therewith the simulated DiffServ network. Similar to the first experiments we use sclient as traffic generator running on Lab3 to measure the throughput of the Web services running on Lab8.

During the first run the QoSProxy is not used and thus no packet marking is performed. The SOAP responses from the Web service are assigned to the BE class on the router and share the bandwidth with other BE traffic between Lab 5 and Lab7.

In the second run we activate the QoSProxy running on Lab8. All IP packets sent from Lab8 are marked with the DSCP value 0x0A and the DiffServ router handles them as Assured Forwarding (AF) class.

We run two test series with different background BE traffic (800 kbyte/s and 1024 kbyte/s) on the network, which is generated between the Lab5 and Lab7.

Figure 63 and Figure 64 show the experimental results for the different network load. In almost every case the use of the QoSProxy ensures slightly better performance. As expected, the benefit of the QoS mapping performed by the QoSProxy is higher than the additional cost of the QoSProxy.

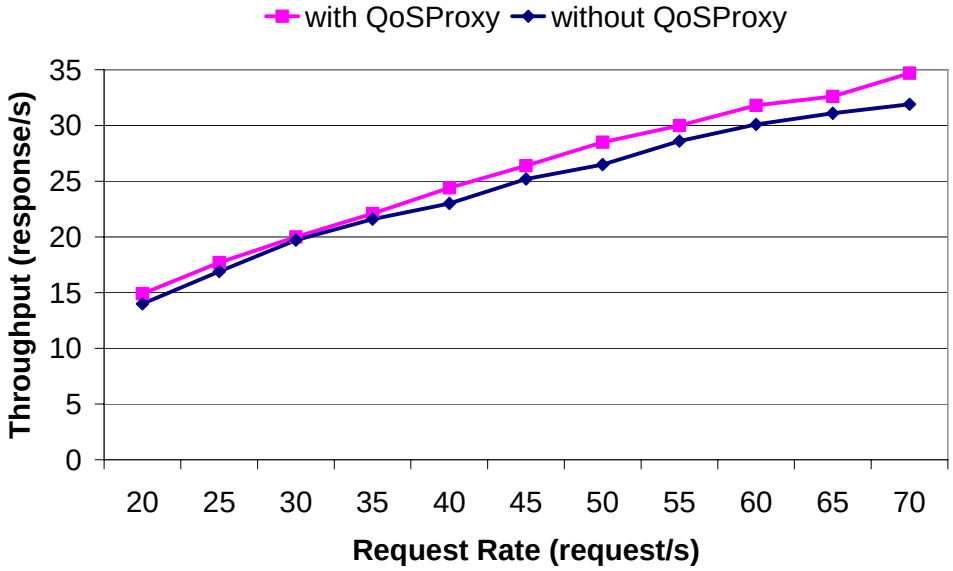

Figure 63. Performance gains with QoSProxy, background traffic: 800 kilobyte/s

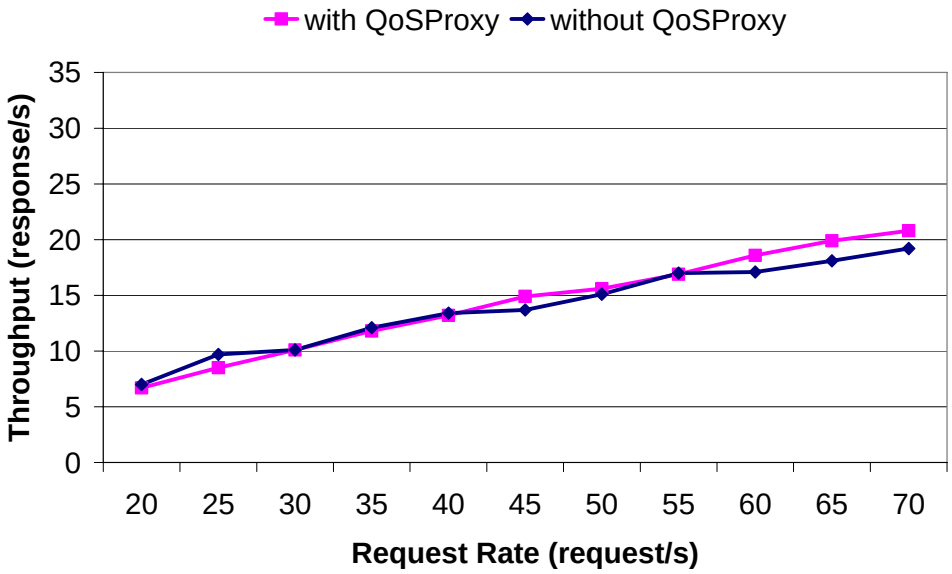

Figure 64. Performance gains with QoSProxy, background traffic: 1024 kilobyte/s

The experiments demonstrate that the QoSProxy is useful in a DiffServ domain and guarantees performance gains when the network is on high demand.

However, the performance gains are not as significant as expected. In the best case we can identify an improvement of 2.8 response per second (refer to Figure 63 at 60 request per second). Therefore, we present further test series with an ideal QoSProxy in the next subsection.

7.3.3 Performance of an ideal QoSProxy

The current implementation of the QoSProxy is a proof of concept implementation having potentials of improvement. E.g. it starts a thread for each connection resulting in significant additional costs when many connections are open simultaneously. We believe that the cost of the QoSProxy could be reduced substantially by an improved implementation. In order to demonstrate the maximum achievable benefit of the QoS mapping concept we simulate an ideal QoSProxy implementation – with no performance impact on the server, no delays and proper handling of limited bandwidth – in the following experiment.

Since we host our WS-QoS-aware Web service on the server Lab8 we disable the QoSProxy and modify the DiffServ router to handle all IP traffic coming from Lab8 as AF. The test scenarios are similar to that of the last subsection, but we investigate the performance of the ideal QoSProxy.

Figure 65 and Figure 66 show the total performance gains at different background traffic (800 kbyte/s and 1024 kbyte/s). As expected, the ideal QoSProxy performs much better than our prototypic implementation. The performance difference is more significant when the background traffic is higher, i.e. the network is more overloaded. The throughput of the AF traffic is in both cases the same when the ideal QoSProxy is applied while the throughput of the

best effort traffic is about 10 responses per second lower when the background traffic is higher when no QoSProxy is applied.

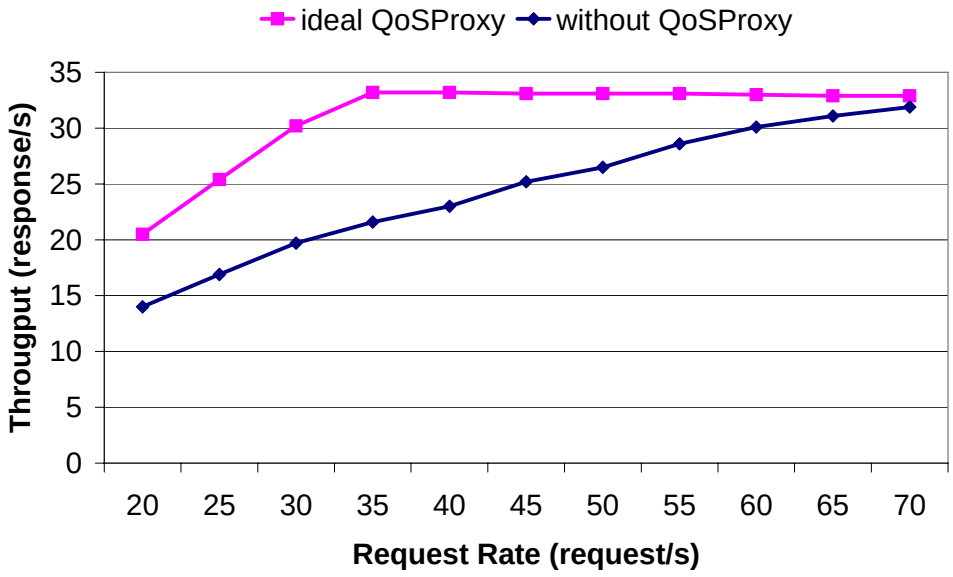

Figure 65. Performance gains with ideal QoSProxy, background traffic: 800 kbyte/s

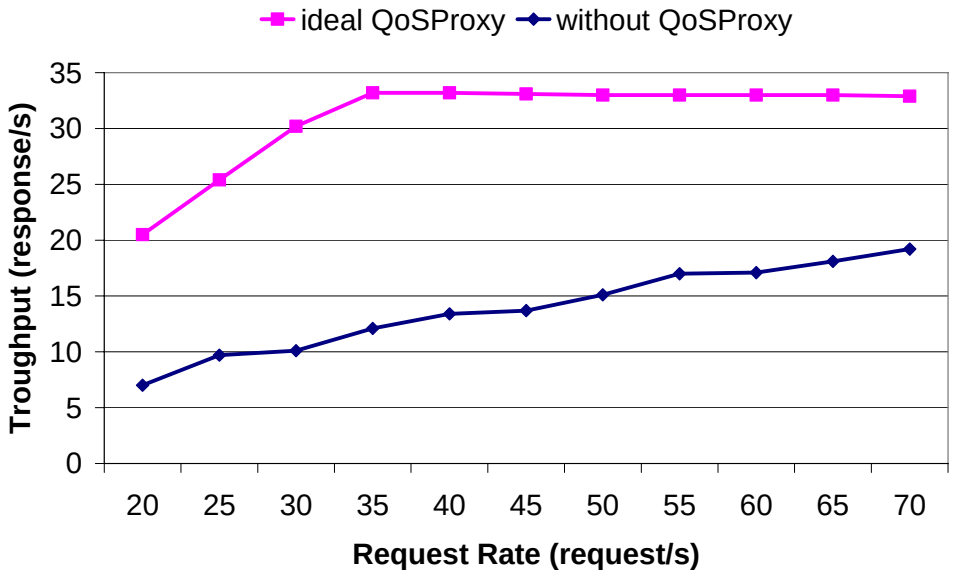

Figure 66. Performance gains with ideal QoSProxy, background traffic: 1024 kbyte/s

7.4 Protecting servers from overload with adaptive WS-QoS offers

In this section, we show the advantages of our WS-QoS framework for service providers. Service providers always strive to serve customers with high performance on the one side and to keep their operating expenses low on the other side. In order to save the server capacity, one could either limit the number of customers or expand the server capacity with new hardware. In the first case, the customer satisfaction can be jeopardized when customers' inquiries are

refused resulting in losing customers. In the second case, server capacity could be unused when less customers use the service than forecasted.

We demonstrate in this experiment that service providers can use our WS-QoS framework in order to control the customers' access. The criteria of the decision can be based on several factors and can be defined by service providers. All QoS metrics can be modified at runtime. The pricing decision for a service with a certain QoS level can be based on the current server load, average response time or the queue length of the Web service server or application server. The latest information about the server state can be published as an integrated part of service offers in order to inform potential customs. This way, the customers receive always the latest information about the server and can make decisions based on that information.

It is worth noting that the test scenario is based on the assumption that the customers always want the latest information about a service offer before invoking the offer. An analogy can be experienced in a stock market where people buy or sell shares.

7.4.1 Test application

We demonstrate in this performance measurement how our WS-QoS framework can be applied to protect the server from overload. We simulate several clients, which dynamically make decision whether or not to use a service. The decision is only based on the current price of the service. The service provider applies our WS-QoS framework to adjust the price depending on the current server load. The WSB that is located on each service client always informs the client about the latest price development.

In order to simplify the test scenario and to strengthen the advantage of the WS-QoS framework, we simulate a network with several clients and a service provider. The clients make decision only based on the price of the service offer. The server changes the price dynamically only depending on the server load. It is worth noting that both the client and the server could make decisions on much more factors, such as network load, average server response time, queue length of the web and application server.

The Web service accepts the ISBN of a book and provides information of inquired books such as title, review, authors, and price. The Web service is implemented with .NET version 1.1 and C#. The cost of the service is 0 EUR at the beginning, as far as the 80% threshold of the CPU time is not reached. A thread in the background observes and logs the CPU load and the average CPU load of the passed 30 sec. If the threshold of the average CPU load reaches 80% the price is increased by 10 Cent. If the threshold is decreased to 70% the price decreases by 10 Cent. The pricing information is always updated by using the XML schema of the WS-QoS framework.

The clients only accept a preset price. If the service price becomes higher than the preset one, the client will not invoke the service. When the service price becomes lower or equal the preset one, the client will use the service once again.

The WSB matches the client's requirement against the service offer and informs the clients whether to use the service or not.

The goal of this experiment is to demonstrate the usability of the WS-QoS framework. The framework is applied to protect the server from overloading. The framework is used for informing the clients about the current server state and QoS offers. In case of over loading, the server could not serve the clients with the promised QoS. We expect that the number of customers using the service will shift with the server last.

7.4.2 Testbed

We use six Pentium IV HT machines, each with 1GB RAM, as shown in Figure 67. All machines are connected to a 100Mbit switch and form a LAN. Both Lab7 and Lab8 run Windows 2003 Server, .NET 1.1, and ASP.NET 1.1. Lab7 hosts a UDDI server. Lab8 hosts the Web service.

Lab2, 3, 5, and 6, run Windows XP with .NET 1.1 and SP 2. We ported the Linux tool sclient to .NET, called Sclient.NET. Sclient.NET runs on each of the client machines, which generate requests to the Web service server running on Lab8. Each of the clients uses a local WSB for lookup. In this performance measurement, the clients only look for the book information service with pricing as the single QoS metric. The offered price should be lower than the client's definition. When an offer is found the Sclient.NET, which is integrated in the client, begins to send requests to the Web service. This way, the server load is increased.

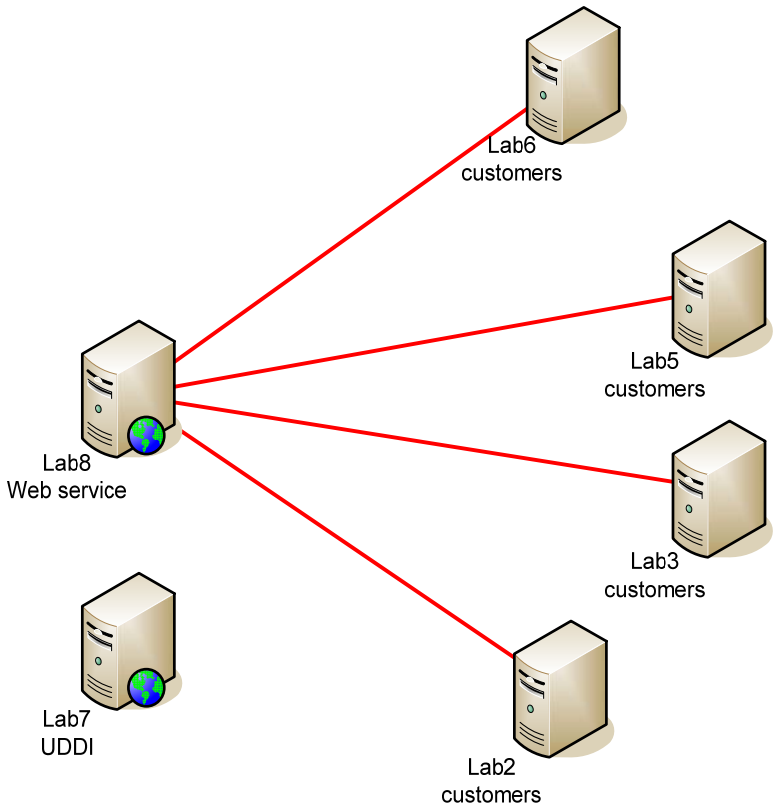

Figure 67. Testbed: adaptive WS-QoS offers

Table 6 shows the configuration of the four clients. Maximal price denotes the highest price the client is willing to pay. Request per second means the number of requests the Sclient.NET will send to the server per second when a suitable offer is found.

Table 6. Client configuration

Client	Max. price	Request per second
Lab2	0.10 €	15
Lab3	0.40 €	15
Lab5	0.50 €	15
Lab6	0.70 €	15

In the following, we will present our measurements in three parts. The first part shows that the increasing number of clients leads to higher server load, resulting in higher price. In the second part, we demonstrate that the server uses the WS-QoS framework to keep the server load stable. The last part shows when the number of clients decease, the server is hold back from over load. In this case, the server uses the WS-QoS framework to announce a lower price in order to gain customers again. It is worth noting that we assume that the clients always ask their WSB for the cheapest price before sending each request.

Part I

Figure 68 depicts the measurement result of the first part. The cost of the service is 0 Cent at the beginning (00:00 sec.). After 20 sec. the clients begin to send requests to the server. The CPU load begins to increase up to 30%. The increase of the average CPU load follows. However, the price remains 0 Cent since the threshold of 80% CPU load is not reached. After 1 min more and more clients send requests to the server. The server loads increases continuously. After 2 min the CPU load reaches 90%. At the point 02:12 the average CPU load reaches the threshold of 80%. Therefore, the server increases the price up to 10 Cent by changing the offer. The price increases another 4 times up to 50 Cent. The CPU load falls few seconds later. That means the price of 50 Cent is too high for most of the customers (refer to Table 6). Therefore, the WSB does not return the Web service to the clients. This way, the Web service is protected from overloading. At this point, the CPU load begins to decrease and the price will go down. New customers are expected.

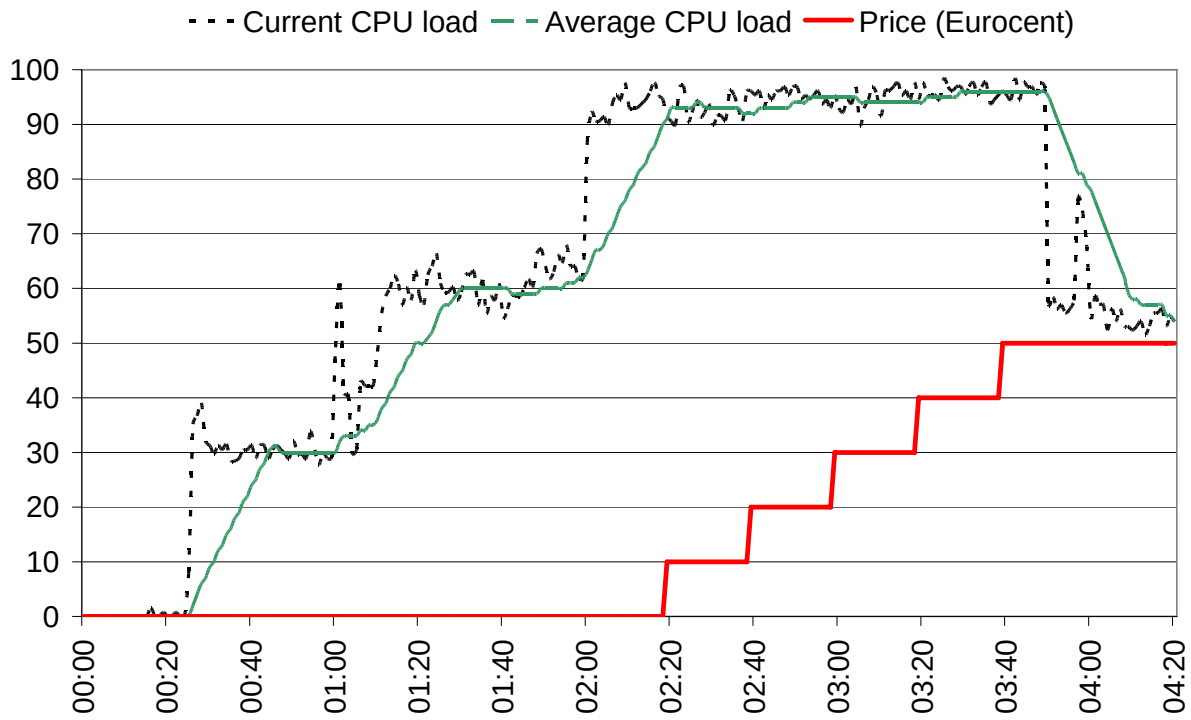

Figure 68. Behaviour of the system in case of increasing CPU load

Part II

In this phase, we present the measurement result that proves that the server load can be controlled by using the WS-QoS framework. As Figure 69 depicts, the price directly influences the server load. Looking at the interval 11:00 to 12:20, the increase of the prices results in the decrease of the server load because less users accept the higher price. After the average CPU load reaches 50% (y-axis), the server begins to reduce the price in order to gain more customers.

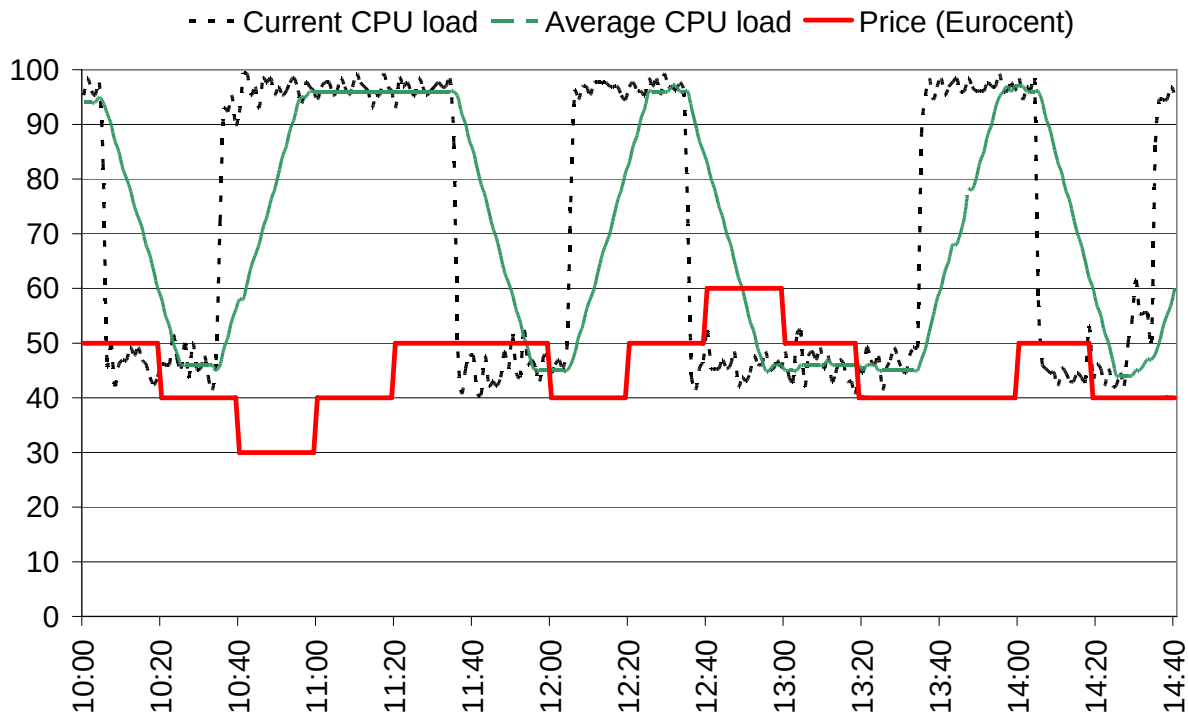

Figure 69. Server in a stable state

In this measurement, we choose a simple model in order to simplify the test and to demonstrate the advantage of the WS-QoS framework. The criteria for changing the price are only based on the CPU load. However, other factors such as the length of the ASP.NET queue, the length of the IIS or the processing time of requests could also be considered. Using these parameters, one can describe the server performance more precisely. The first two parameters can be read out from the so called Performance Counter of Windows 2003 Server. The processing time of requests has to be calculated.

It is worth noting that it is hard to forecast the number and behaviors of clients and the request rate. The WS-QoS framework is the solution to control and adjust the server load dynamically.

Part III

In the last part, we present the behavior of the server when the number of requests decreases. In this case, the server has to make its offer more attractive by decreasing the price. Of course, one can apply other parameters in order attract users.

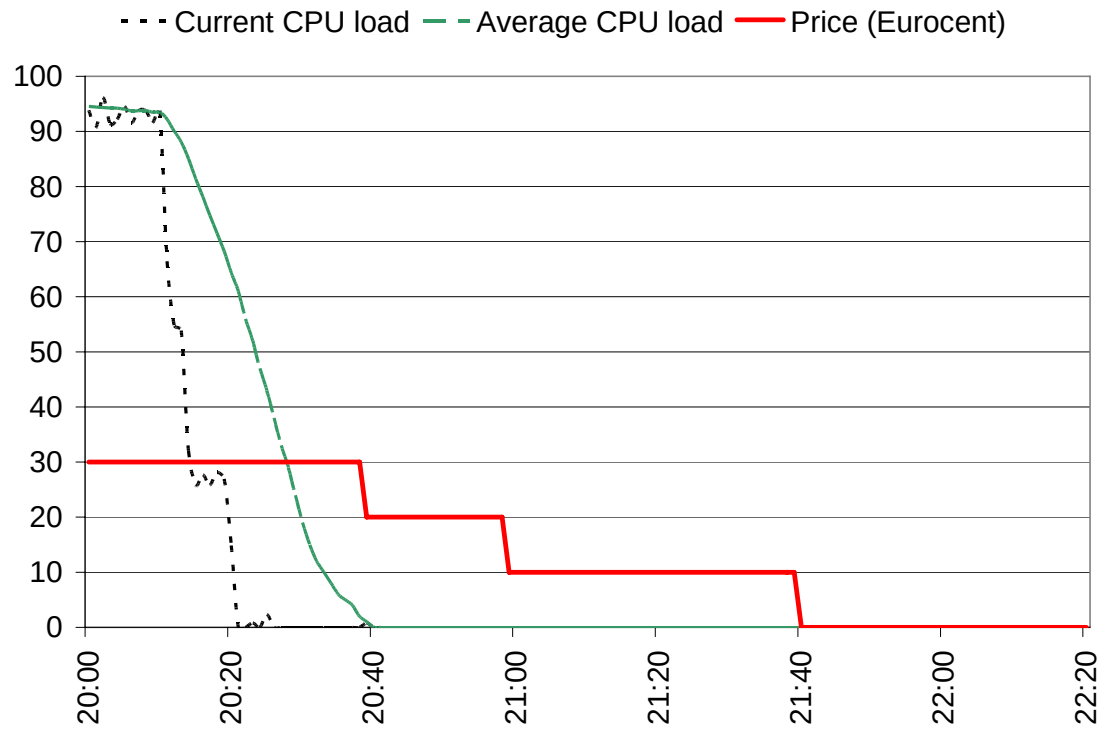

Figure 70. Behavior of the server in case of deceasing number of requests

The Sclient.NET on Lab2, 3, and 5 stops to send requests to the server between 20:00 and 20:20 because the service price is too high for their clients. At the point 20:20 the CPU load on the server decreases down to 70%. Therefore, the server decreases the price in order to gain more clients. In case of no more requests, the price continuously falls down to 0 Cent. New customers are expected.

7.4.3 Conclusion

In this Chapter, we presented four series of performance measurements. In the first measurement, we determined the impact of Web service overhead on web servers and mobile clients. Compression is one way of dealing with the problem of large message sizes of Web services. We showed that compression is useful for poorly connected clients with resource-constrained devices despite the CPU time required for decompressing the responses. Compression also decreases server performance due to the additional CPU time required. In the approach presented in this measurement, we let the clients decide whether they want their responses compressed. During low demand, the server compresses the responses for all clients that have asked for compressed responses as well as for clients that have not indicated any preference. During high server demand, only responses to clients that have asked for compressed response are compressed. Our experiments have shown that both the server and the clients, in particular clients that are poorly connected, benefit from this approach.

In the second measurement, we demonstrated the performance of the WSB. The WSB serves Web service clients to lookup and select a Web service offer according to various QoS requirements. We have shown that the performance of ‘WSB can be increased significantly when a database is applied.

In the third part, we measured the performance of the QoSProxy. We ascertained the cost of the QoSProxy in the first part. The second part showed that the overall performance of Web service communication is always better when a QoSProxy is applied. However, the performance gain is not that much as expected. We think that a reimplementation of the QoSProxy could result in better performance. Therefore, we conducted the third measurement in order to reveal the theoretical performance improvement when an ideal QoSProxy was applied, i.e. the ideal QoSProxy consumes no or very few resources. In this case, we can observe a significant performance improvement.

In last measurement series, we demonstrated the advantages of our WS-QoS framework for web service providers. Service providers can easily use our framework in order to control the customers' access. The criteria of the decision can be based on several factors and can be defined by service providers themselves. All QoS metrics can be modified dynamically at runtime. The pricing decision for a service with a certain QoS level can be based on the current server load, average response time or the queue length of the application server. This way, service provider can always prevent their servers from overloading in order to service clients efficiently.