5 The implementation of the WS-QoS framework

This chapter describes the prototypic implementation of the WS-QoS framework introduced in Chapter 4. Figure 32 shows the implementation of the WS-QoS framework, which supports both wired and mobile devices. It encompasses the Offer Broker, the Requirement Manager, and the Base and Supporting Functions.

The first subsection of this chapter introduces a scenario for QoS-aware service selection. Section 5.2 discusses the WS-QoS XML schema, which is the core of the whole architecture. Section 5.3 introduces the WS-QoS editor for editing QoS requirements. Section 5.4 and 5.5 deal with the WS-QoS monitor and the Requirement Manager. Section 5.6 discusses the Web service broker, which improved the lookup and selection of services. Section 5.7 focuses on the mapping of the QoS requirements from the higher layer onto the network layer. Section 5.8 highlights our proposal that all participating domains of Web service communication should take QoS into consideration. Section 5.9 demonstrates how to apply the WS-QoS framework in order to support mobile Web service communication. Section 5.10 gives a conclusion of this chapter. All the implementation is based on .NET and Windows platforms including Windows XP and Windows Server 2003.

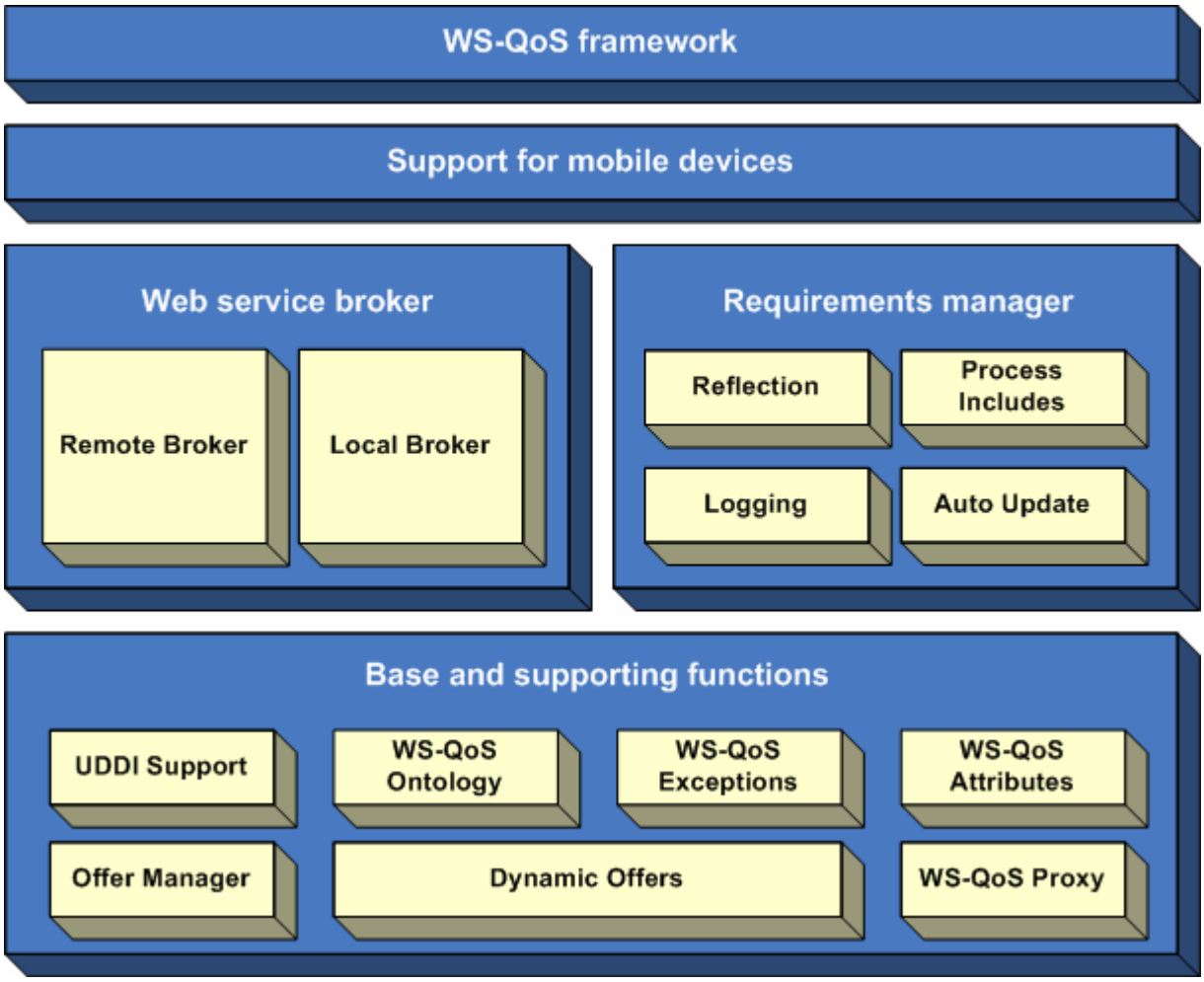

Figure 32. Implementation of the WS-QoS framework

Due to the requirements discussed in Section 4.1, we implemented the WS-QoS framework with the following objectives:

- The WS-QoS API that allows C# and ASP .NET application developers to define WS-QoS requirements for both client applications and Web service offers
- The WS-QoS Editor that allows the editing of the QoS parameters through a graphic user interface (GUI)
- The Requirement Manager that is responsible for retrieving clients' requirements
- The WSB, which is responsible for the QoS-aware service selection
- The WS-QoS Monitor, which is used for examining the compliance of offers

5.1 A scenario for QoS-aware service selection

From the client point of view, a client application can use one or more types (*tModel*) of Web services. The interfaces described in WSDL of Web services are known at the implementation time. A proxy class (in the context of Microsoft Visual Studio .NET also known as a Web Reference) is generated from the tModel's WSDL description for each service type. Static WS-QoS custom attributes or *import* attributes referencing dynamic requirements in a WS-QoS XML file can now be assigned to the newly created proxy class and its methods (known as web methods in Visual Studio .NET). Finally, the proxy class is handed over to an instance of the WS-QoS Requirement Manager, which will retrieve the attributes through the reflection technique and thus holds a representation of the current client requirements. One can also use the Requirement Manager to adjust the QoS requirements at runtime without recompiling any code.

The process flow of a dynamic QoS-aware service selection is depicted in Figure 33. On initialization, the client application creates an instance of a WS-QoS Requirement Manager. A WS-QoS Broker (WSB) is already running in the same network. Before the service invocation, the client application will use the Requirement Manager to state its current QoS requirements and then inquire the WSB for the most appropriate service offer available that fulfills its requirements. The WSB selects the most appropriate offer on behalf of the client from the WSBs local database. (Note we assume in this case that the WSB has already a local and up-to-date cache of the services the client is asking for). Therefore, the WSB does not contact any UDDI and service providers for searching the required service. This model results in a short response time. Once the client gets the required offer from the WSB, the client will invoke the service with the desired QoS properties.

The QoS properties are transmitted in the SOAP header to the service provider that can treat the request based on the QoS properties. For example, it could set the thread priority or, as a load balancer, forward it to one of various possible application servers. Yet, the information is not only intended for the Web service provider: A WS-QoS proxy is able to interpret the desired transport QoS priorities and mark the outgoing packets accordingly so that the higher layer applications can take advantages of the QoS support that the underlying QoS-aware transport technologies provide. Furthermore, the information in the SOAP headers can be used to perform encryption or digital signatures.

Having processed the client's request, the service provider will send the response back to the client. The service provider has to ensure that it can carry out the client's requirements about the transport and security. The service provider should set the requirements in the SOAP header so that they can be evaluated and carried out by the participating components on the way back to the client. The components are e.g. the QoS proxy on the server side, WS-QoS-aware routers in the network, or an access point for mobile devices.

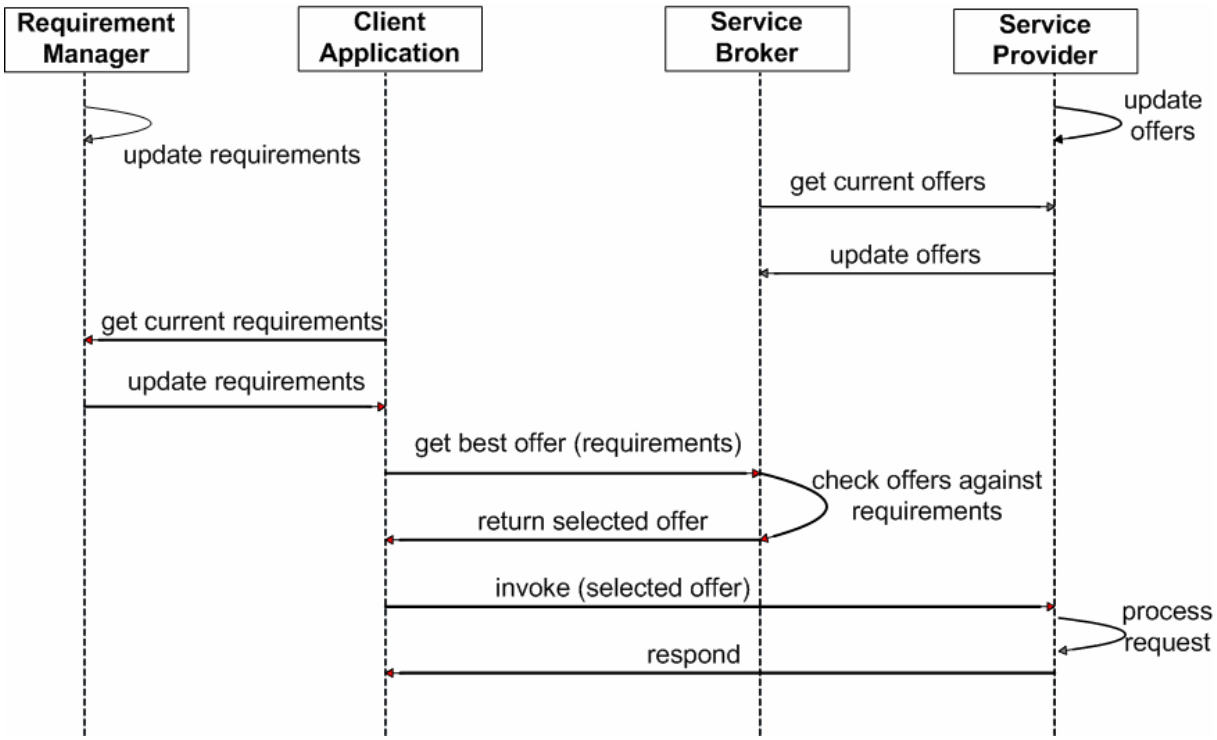

Figure 33. Scenario for the initial request of offers for a specific service type

5.2 WS-QoS XML schema

Figure 34 illustrates the implemented classes of the WS-QoS XML schema. Users can declare WS-QoS elements with static custom attributes which are assigned to a service proxy class generated from a WSDL file. QoS elements for specific operations of a service can be assigned to the corresponding Web method. All WS-QoS attributes inherit the abstract class *WSQoSAttribute*, which provides a basic infrastructure for managing the WS-QoS information. In a concrete class the method *getInnerXML()* has to be overridden in order to provide an XML representation of the attribute. The method *updateProperties(XMLElement)* has to be overridden as well in order to create attributes from an XML source.

According to the WS-QoS XML schema, the WS-QoS API provides the following custom attributes:

- *ServerQoSMetricsAttribute* and *CustomServerQoSMetricAttribute,*
- *TransportQoSPrioritiesAttribute* and *CustomTransportQoSPriorityAttribute,*
- *SecurityAndTransactionAttribute* and *ProtocolSupportAttribute,*
- *ContractAndMonitoringAttribute* and *ThirdPartyAttribute,*
- *PriceAttribute.*

Furthermore, instances of the classes *DefaultQoSInfo* and *OperationQoSInfo* (both inheriting the class *QoSInfo*) and a *QoSRequirements* object (which is derived from the class *QoSDefinition*) are created to hold the attributes retrieved from the proxy class. Apart from holding the values for certain WS-QoS elements, these objects implement the functionality deciding whether one object provides an equal or better QoS level. The function *Includes()* is invoked from the WSB when testing an offer against a client requirement definition.

While the values of most WS-QoS elements can be compared in a simple way, the custom attribute *PriceAttribute* implements a function *IsCheaper()*, which calls a currency converter object in case that two price statements are declared with distinct currencies. The WS-QoS API defines the abstract class *CurrencyConverter*, which holds a static property *ActiveCurrencyConverter*. This property has to be set to an instance of a class implementing this abstract class before the WSB can make use of the currency converter. The package *WS-QoSUtil* provides the class *WebServiceXCurrencyConverter*, which uses a Web service from the service provider WebserviceX.NET [46] to obtain current exchange rates.

Furthermore, the *import* attribute is defined to reference WS-QoS XML files containing further requirements, which can be changed dynamically at runtime. When an *import* attribute is initiated, the imported file is read and a corresponding QoS definition object is built using the attribute's *getInnerXML()* method. Then a thread is initiated to perform regular checks on whether (the content of) the imported file has been changed. If so, the Requirement Manager is informed and the overall requirement object is rebuilt using the updated representation of the dynamic attributes.

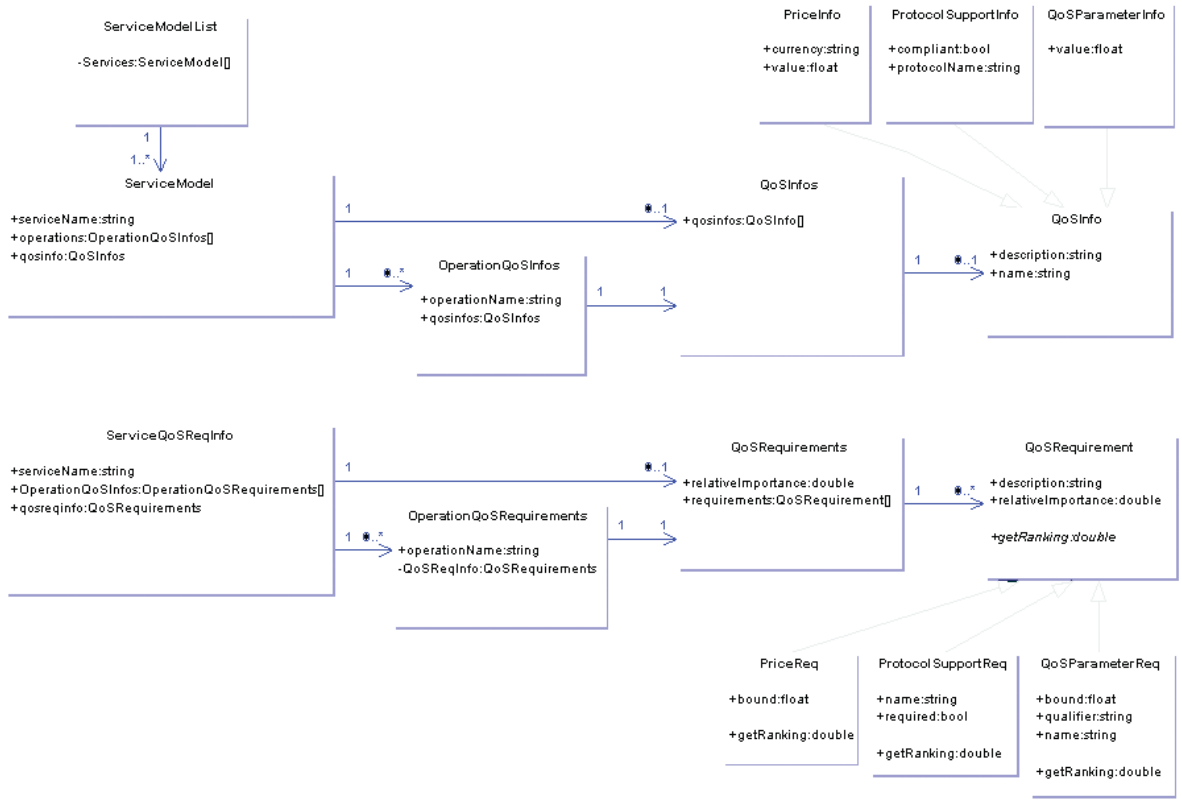

Figure 34. Classes of the WS-QoS API

5.3 WS-QoS editor

The WS-QoS Editor allows both the service client and the service provider to easily edit their QoS requirements or offers, respectively. They neither need to know the details of the WS-QoS XML schema nor have any programming skill. One or more XML-based *.wsqos* files are generated automatically. The WSDL files are normally generated automatically by a tool such as wsdl.exe in case of the .NET runtime. In case of a service offer, one or more references of the *.wsqos* files are added manually into the WSDL file of the service. In case of a service request, the WS-QoS Requirement Manager will retrieve the values defined in a *.wsqos* file.

Figure 35 shows the GUI for defining custom QoS properties. One can define

- the name of the requirement,
- the scope in which the requirements are valid, possible scopes are individual operation of a service or the whole service,
- the standard metrics of standard QoS aspects such as processing time, request per second, availability, and reliability as server QoS metrics,
- the price for the service usage the client is willing to pay or the service provider is going to charge, and

- custom metrics (shown in the GUI on the left side of Figure 35) by applying ontology.

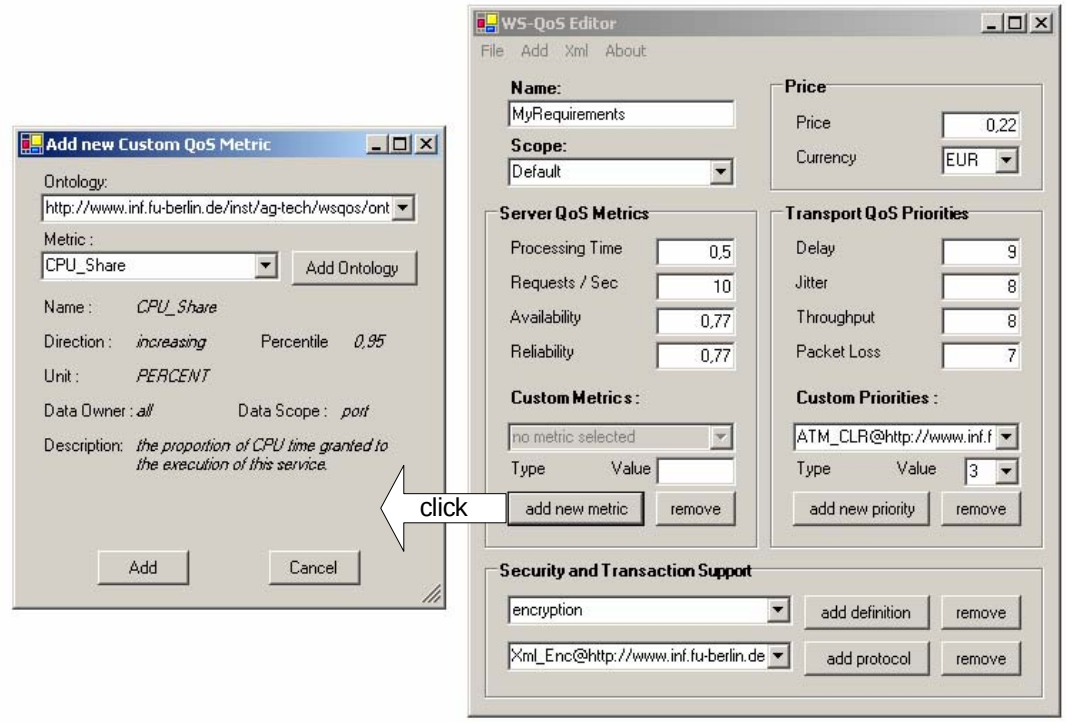

Figure 35. Defining custom QoS properties

5.4 WS-QoS requirement manager

On initialization, the WS-QoS Requirement Manager obtains a reference to the service proxy class to which either requirement attributes have been assigned or a reference to a *.wsqos* file is given. The Requirement Manager retrieves the QoS attributes either from the proxy class or from a *.wsqos* file. It then collects all *import* attributes, builds WS-QoS definition objects and sets their parent property to receive update messages in case that a *.wsqos* file has been changed. Finally, the newly built WS-QoS definition objects are added to those retrieved from the static attributes.

5.5 WS-QoS monitor

We have developed the WS-QoS Monitor, which examines all available offers and the current client requirements, making it possible to check the compliance of offers. If no appropriate offer can be found, the overview of possible offers will help users to evaluate what requirements might be inappropriate and users could then make adjustments needed in order to find a match.

Moreover, the WS-QoS Requirement Manager can be configured to log current requirements. Once this file is registered in the monitor, requirements can be viewed in the requirement watch window or directly in the offer window of the

GUI. Finally, the package *WS-QoS Util* provides a SOAP extension attribute, which can be assigned to the proxy's web methods. The SOAP extension will log WS-QoS SOAP headers of all service requests and responses. One can register this file in the monitor as well use it to survey WS-QoS SOAP headers in the SOAP header watch window of the GUI. Figure 36 shows the main window with two watch windows in the lower right corner and a QoS-aware client application to the upper right [47].

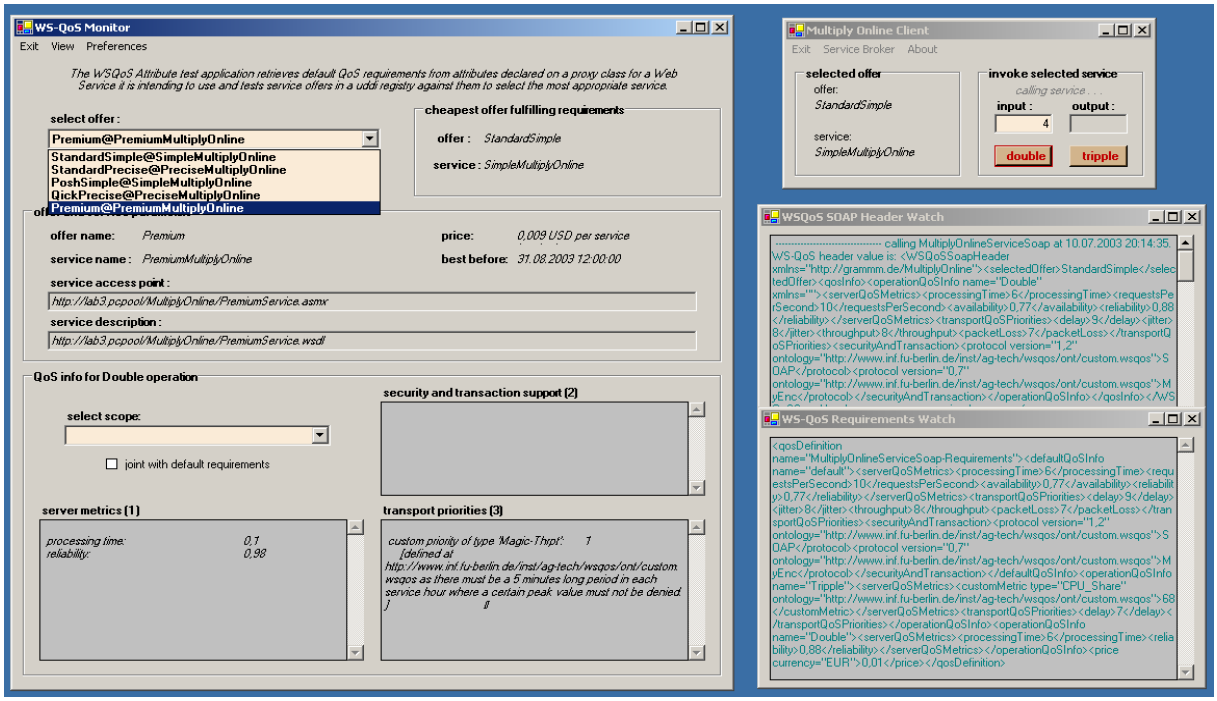

Figure 36. WS-QoS Monitor surveying the service selection of a sample client

5.6 Web service broker

The WS-QoS Broker (WSB) holds up-to-date information on offers currently available for a group of services, which have been requested recently. Offers are grouped by the interface (*tModel*) that the services providing them implement. The first time a client requests a service implementing a certain interface the WSB will consult one or more UDDI registries. The WSDL files for these services are then checked for WS-QoS extensions and available offers are built. From this time the WSB will consult the newly created offer list in order to find the best match for clients and their requirements, allowing an accelerated lookup process.

To keep offer lists up-to-date, the WSB inquires the UDDI periodically in order to find new offers. Once an offer expires, the WSB removes it from its local cache. If the validity of the offer is extended, the WSB will be re-detected it during the next check.

When a client application inquires the WSB for the cheapest available offer, it sends its QoS requirements as a part of the request. In the order of their price, the WSB then tests the available offers whether they fulfill the client's

requirements. The first compliant offer is returned to the client. It is worth noting that one can implement an own strategy for defining the QoS parameters and the selection of the appropriate services. We just give here an idea of how the selection could be done.

There are two implementations of the WSB. One is a local object running within the application. This ensures a good performance of the service selection and detailed information on available offers, as needed for the WS-QoS Monitor. The other implementation uses a remote Web service to obtain the access point of the most appropriate service. This version is mainly intended as a light process for multiple client applications that use a single private WSB that runs as a Web service within a network domain. The WSB could well be used by any other WS-QoS compliant implementation.

5.7 QoS mapping

QoS-aware service invocation is the third step of Web service communication from the client's point of view. The client has defined its QoS requirements and got the most appropriate offer from the WSB. It is now going to invoke the service.

The WS-QoS parameters are placed in the SOAP header. We have implemented QoS Proxies to map the QoS requirements regarding the network QoS support for the DiffServ network, which supports QoS classes. We have implemented a QoS Proxy for each client side and server side, called QoSProxyC for the client side and QoSProxyS for the server side. The QoSProxies are located between the Web service and the network layer.

The QoSProxies classify the IP packets by marking the DSCP in the IP packets. The routers in a DiffServ domain forward the IP packets according to the DSCP and network policies.

5.7.1 QoSProxy for the client side

As Figure 37 shows the QosProxyC is placed between the client application and its network interface. When the client sends SOAP messages the QoSProxyC picks the QoS requirements, analyses them, and marks the DSCP values of the outgoing IP packets. the QoS requirements are located in the SOAP header. When responses arrive, the QosProxyC just passes them to the client application.

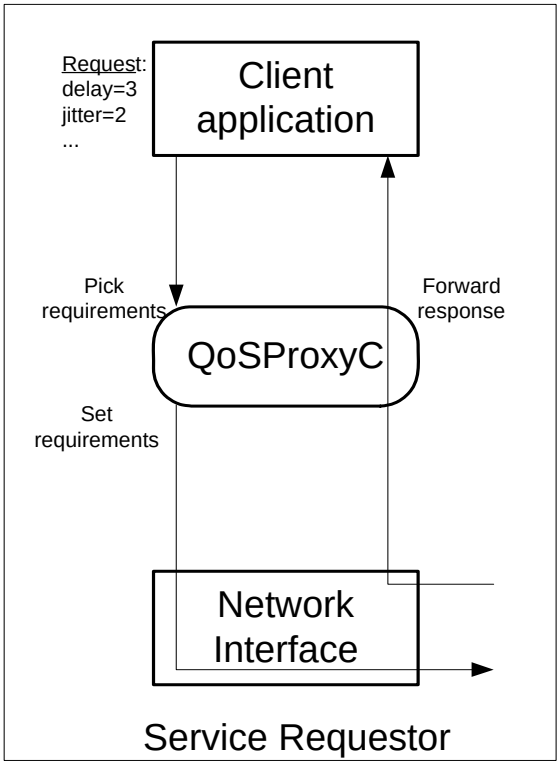

Figure 37. QoSProxy on the client side

Placing the QoSProxyC between the application and network layer has several practical advantages:

- The application may not always possess sufficient system privileges to perform the required actions, since manipulating IP packets requires normally more system rights.
- Application developers need not have in depth knowledge of the underlying network technologies and need not include any program code of the low level network API.
- The system stability is not affected since the high-level application does not call any low-level APIs.
- The error handling and maintenance of applications are improved.
- The definition of QoS requirements is independent of the underlying transport technology and network interface, respectively.

5.7.2 QoSProxy for the server side

When a client request (in form of packets) arrives at the server, the Web service will process it, and then send the response back to the client. In the normal case, the response would not obtain any QoS support, and the IP packets would be sent with the “Best Effort” model. In this case, the QoS support would be given only in one direction: from the client to the server. It would not make sense when the network (in the DiffServ domain) supported the packets only on their way from

the client to the server, while the packets transmitted from the server to the client would be under the mercy of the current network status.

When packets sent by the client will obtain a specific DiffServ support, packets sent by the server should obtain the same DiffServ support in order to preserve the same level of network support. Therefore, we have implemented QoSProxyS for the web server side for the QoS mapping. As we know, the client will define its requirements in the SOAP message it sends to the server. The server should consequently define QoS requirements in the response SOAP message returning to the client.

Actually, there are two approaches for setting the values of these transport parameters on the server side:

- The server uses the same values defined by the client in the request message
- The server defines its own values irrespective of the values present in the received SOAP message

Each approach has its characteristics. When the server receives a SOAP message in which the client has defined a specific support level which does not currently suit the server or which the server cannot ensure at the moment, the server will then set its own QoS parameters. However, this may not be practical because the packets sent by the server should obtain the same support level as the received packets have obtained. When the packets will be sent with a lower support level, this may endanger the fulfillment of the client requirements. They may get lost or dropped, and this would contradict the goals of the whole approach. Therefore, it seems to be more reasonable to use the same parameter values as those set by the client.

As on the client side, we have placed the QoSProxyS between the server application and its network interface which has several advantages:

- Disburden the server from additional work
- QoSProxyS is independent from web servers hosting Web services

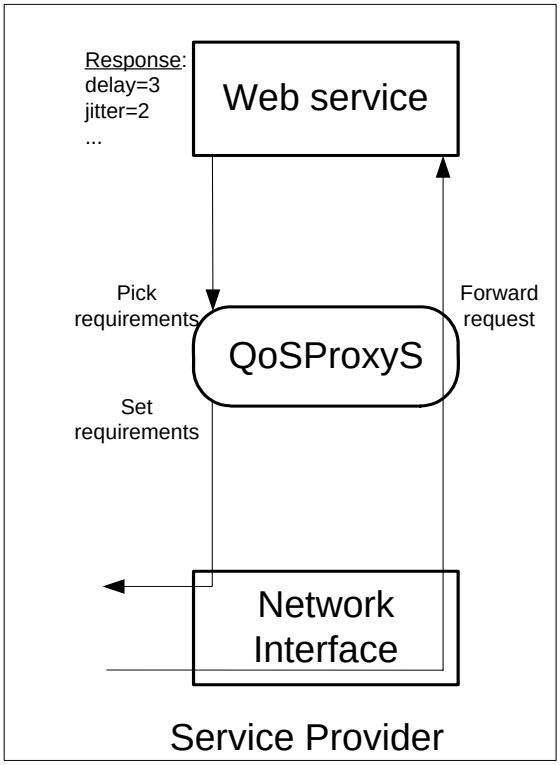

Figure 38. QoSProxy on the server side

As Figure 38 depicts, the QoSProxyS does not process the incoming messages. When the SOAP messages are sent, the QoSProxyS marks the DSCP of the IP packets due to the QoS requirements located in the SOAP header.

The main difference between the QoSProxyC and QoSProxyS stems from the different implementations. Since a server has to serve different clients simultaneously, the QoSProxyS has to maintain flows to different clients resulting in a more complex implementation. Details about how to maintaining different flows on the server machine are given in [48].

For the marking of the DSCP, both the client and the server QoS proxy applies a mapping table which is introduced in the following subsection.

5.7.3 Mapping table

QoS requirements for the network are mapped to the DSCP by applying a mapping table shown in Table 3. The standard metrics for the network are delay, jitter, throughput, and packet loss rate. The priorities set for that metrics are dimensionless. But metrics are measured in:

- Delay [s]
- Jitter [s]
- Throughput [bit/s]
- PacketLoss [%]

The priority ranges from 1 to 10. The smaller the priority value the higher the priority. E.g. delay=1 means the lowest possible delay is required.

Table 3. Mapping table for DiffServ

Priority	1	2	3	4	5	6	7	8	9	10
DSCP (decimal)	46	46	10	12	18	20	26	28	34	36
DiffServ Service class	PS	PS	AF1	AF1	AF2	AF2	AF3	AF3	AF4	AF4

Since Premium Service (PS) should provide a special support to the application, and especially in delay and jitter (because PS means low delay and low jitter values), it is reasonable to assign PS to the first 2 priority values (i.e. to priorities 1 and 2).

We assume that the priorities should be "homogenous" which means that an application would not set a very high value for one metric and very low values for another metrics.

There are four classes of assured forwarding (AF): AF1, AF2, AF3, and AF4, whereby AF1 has a higher priority and AF4 has the lowest one.

One can specify the minimum bandwidth (referred to as min_BW) and the maximum bandwidth (referred to as max_BW) in the QoSProxy. The min_BW and max_BW are required to subdivide the maximal allowed bandwidth to each application into ten segments, which is the number of possible bandwidth priority values (from 1 to 10). Accordingly, a throughput priority of 1 corresponds to max_BW, and throughput priority of 10 corresponds to min_BW. We divide a segment with a length of max_BW into 10 pieces on a "bandwidth axis", and each segment will correspond to a throughput priority value.

Mathematically, we have a linear equation between a priority value and the real bandwidth amount. We can calculate the increasing factor of this equation with the help of Equation 1:

$$factor = \frac{\max\_BW - \min\_BW}{9}$$

Equation 1. Factor calculation

The real bandwidth value that will be assigned to an application can be obtained with the help of Equation 2:

$$BW = \left[\left(10 - priority\_BW\right) * factor + \min\_BW\right]$$

Equation 2. Bandwidth calculation

(Whereas priority_BW is the throughput priority value set by the application).

Figure 39 shows an example for such a mapping process:

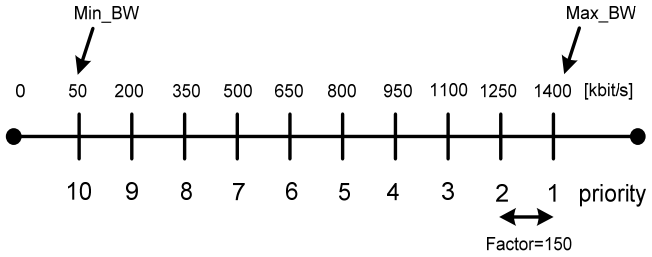

Figure 39. Example showing the different BW values

In the example, the user has specified the two following values:

- min_BW = 50 kbit/s
- max_BW = 1400 kbit/s

By using Equation 1 we get: *factor* = 150

Taking a throughput priority of 4 as an example, and by using Equation 2 we get: *assigned bandwidth* = $\left[(10-4)*150+50\right]$ kbit/s = 950 kbit/s

When multiple applications on the same host are running with Premium Service support, then all assigned bandwidth values will be added together and the sum must not be greater than total_BW. When an application requires more bandwidth than available, two possibilities exist to handle it: either the QoS proxy application will reject the request and return an error message to the requesting application, or the data will be sent with best effort model (i.e. with no DiffServ support). We use the first method in our application since the user should be informed when her requirements can't be satisfied.

5.7.4 QoS proxies

Figure 40 depicts the participating components and the data flows during the interaction between a Web service client and the service provider at runtime. The QoS requirements regarding the network performance specified by each the client and the server will be declared in the SOAP headers, which will be parsed by the corresponding QoS proxies on both the client and the server side. Based on the QoS parameters, the proxies mark the DiffServ specific DSCP in the IP packets. DiffServ routers in the DiffServ domain will treat the traffic between client and server depending on the DSCP value set in the packet headers.

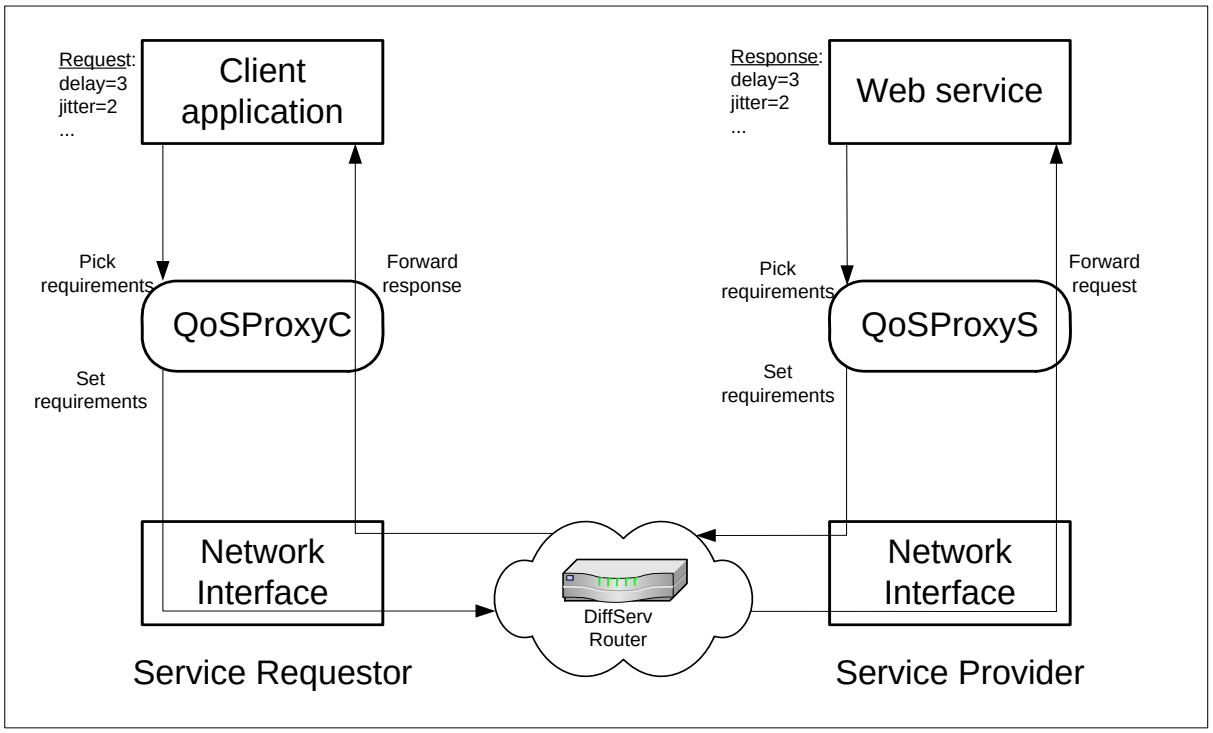

Figure 40. Proxy applications on both sides

In Section 7.3, we will present of performance measurement of the QoS proxies. The measurement results will prove the advantages of the QoS proxies. The QoS mapping of traffic parameters is one of many QoS aspects during Web service communication. In the following subsections, we introduce how to apply the WS-QoS framework in order to enable the adaptive server performance and adaptive message load compression for mobile devices.

5.8 Adaptive server performance

The *serverQoSMetrics* element of a WS-QoS definition shown in Figure 14 in Section 4.2.1 specifies server performance in terms of processing time, throughput, availability, and reliability.

Figure 41 shows an example of the definition of the server performance. The processing time should be 5ms, the throughput of the server should be 30 requests per second, the reliability of the server should be 99%, and the availability of the server should be 98%.

```
<?xml version="1.0" encoding="utf-8" ?>
<wsqos xmlns="http://wsqos.org/">
  ...
  <operationQoSInfo name="MyOperation">
    ...
    <serverQoSMetrics>
      <processingTime>5</processingTime>
      <requestsPerSecond>30</requestsPerSecond>
      <reliability>0.99</reliability>
      <availability>0.98</availability>
    </serverQoSMetrics>
    ...
  </operationQoSInfo>
  ...
    </wsqos>
```

Figure 41. A WS-QoS serverQoSMetrics element

A service offer defines a distinct level of service performance. Request differentiation can take place on various levels. In the current implementation, we perform request differentiation on application level. Response times are influenced by setting the priority of the thread processing the request according to the clients' requirements.

While we pursue a simple prioritization mechanism, more elaborate approaches of request differentiation could be applied in order to distinguish levels of server performance. The availability of different resources provided by a web server can be differentiated on OS level [17][18][39]. Such an approaches guarantee prioritized access to a limited scope of resources even in overloaded servers.

5.9 End-to-end QoS support for adaptive message load compression

It is easy to extend our WS-QoS framework in order to support mobile devices that access Web services over the air. Due to our experiments introduced in Section 7.1, compression of data is attractive for mobile users. Wireless transmission of a bit can require 1000 times more energy than a single 32-bit computation [43]. Compression and decompression on mobile devices need not be performed by the same algorithm. Energy consumption can be reduced up to 30% by choosing the lowest-energy compressor and decompressor on a mobile device [43].

In order to signal which compression algorithm is to be used we extend the *securityAndTransaction* node of the *operationQoSInfo* element in our WS-QoS

XML schema with two sub nodes *compression* and *decompression*, as shown in Figure 42. Since *compression* and *decompression* are custom defined QoS aspects, we have specified ontology references for them, also shown in Figure 42.

```
<?xml version="1.0" encoding="utf-8" ?>
<wsqos xmlns="http://wsqos.org/">
  ...
  <operationQoSInfo name="MyOperation">
    ...
    <securityAndTransaction name="compression" requires="one">
        <protocol name="zlib"
ontology="http://www.mydomain.com/compression.wsqos" />
        <protocol name="bzip2"
ontology="http://www.mydomain.com/compression.wsqos" />
    </securityAndTransaction>
    <securityAndTransaction name="decompression" requires="one">
       <protocol name="bzip2"
ontology="http://www.mydomain.com/compression.wsqos" />
        <protocol name="zlib"
ontology="http://www.mydomain.com/compression.wsqos" />
    </securityAndTransaction>
    ...
  </operationQoSInfo>
  ...
    </wsqos>
```

Figure 42. securityAndTransaction entries for compression and decompression algorithms

Server providers announce which (de) compression algorithms they support. Clients define which compression algorithm a service provider has to use to compress responses. Algorithms are listed in the order of preference, so the most appropriate match can be found.

5.10 Conclusion and discussion

This chapter presented the implementation of the WS-QoS architecture. The QoS-aware Web services communication consists of three steps from the client's point of view. After introducing a scenario of QoS-aware Web service selection, we first introduced the WS-QoS XML schema, which is the core of the whole architecture. All components participating in a WS-QoS aware communication apply this schema. The second step for the client is the QoS-aware service discovery and selection. Users can apply the WS-QoS Editor, Requirement Manager, and Monitor to edit, manage and monitor their QoS requirements. WSB is responsible for selecting the most appropriate offer available regarding to

client's requirements. When having the most appropriate service provider, the client invoked the service directly at the service provider in the third step. At this time, the specified transport QoS metrics are mapped by the QoS proxies on both the client and the server side. Web server will consider metrics defined in the *serverQoSMetrics* element to performance request differentiation. Finally, we introduced the notion of adaptive message load compression, which is useful for mobile devices that access Web services over the air.

In the next chapter, we introduce how to apply the WS-QoS framework.