4 The design of the WS-QoS architecture

The fundamental goal of the design of the WS-QoS architecture is QoS support during the whole communication process. Standard conformity, scalability, extensibility have to be supported as well as QoS mapping between different layers in terms of the Internet model.

This chapter describes the design of the WS-QoS framework. The framework is fully compatible to standard Web service protocols such as SOAP, WSDL, and UDDI. The WS-QoS framework specific elements are integrated into WSDL in a standard conform way.

Section 4.1 examines the requirements for QoS-aware Web service communication. Section 4.2 presents the WS-QoS XML schema which is applied to define both QoS requirements and offers. Section 4.3 focuses on the QoS-aware service discovery and selection based on the QoS requirements and offers. Section 4.4 discusses the design issues for QoS-aware service invocation. A discussion and evaluation of the architecture presented in the chapter can be found in 4.5.

4.1 Requirements for QoS-aware Web services

As stated in the previous chapter, QoS support is required in all stages of a Web service communication life cycle to enable true QoS support. A QoS-aware Web service communication process consists of three phases from the client's point of view. First of all is the QoS specification of QoS requirements. QoS-aware service discovery and selection regarding to the QoS requirements has to be supported in the second phase. The specified QoS is performed at service invocation in a third phase.

The standard Web service architecture depends on the invocation of operations by sending and receiving messages [25]. WSDL describes functional aspects of a Web service, but not non-functional aspects such as QoS. As more and more companies offer competing Web services, QoS aspects of Web services will become the decisive factor for the success of competitors. Specifications of both QoS aspects and parameters specific to Web service offers and requirements are essential in order to ensure their compatibility and therewith comparability.

UDDI is designed for publication and discovery of Web services. Yellow, white, and green page information of Web services such as contact information of a company or industry categories can be stored and viewed there. Since non-functional properties of a service such as price and performance parameters are subjects to frequent changes, repeated requests to a huge UDDI registry may prove to be a bottleneck for the performance of service lookups and selection. Therefore, a more sophisticated mechanism is required to improve service discovery and selection.

Many domains and layers in terms of the Internet Model participate in a Web service communication process. Figure 10 shows possible participating domains such as a client (device), an access point, a network, routers in the network, and a web server hosting Web services. The participating domains should provide QoS support actively, resulting in an overall performance gain. The layers are e.g. application layer and transport layer. If the Web services are also considered as a layer, it could be placed between the application and network layer, as depicted in Figure 2. In Chapter 7, we will demonstrate how our WS-QoS framework can be applied in order to support QoS through the different layers and domains.

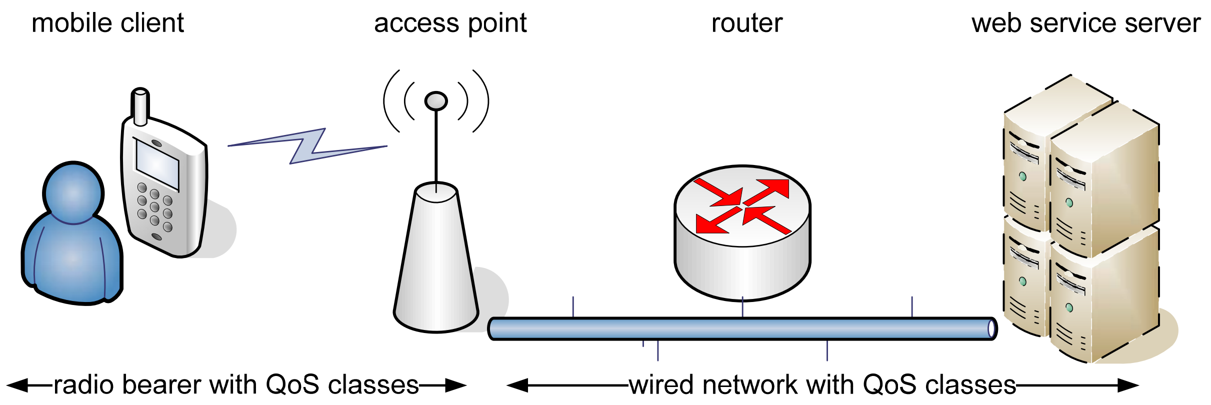

Figure 10. Participating domains

Our WS-QoS framework targets the following main requirements

- designing an architecture that allows both service clients and service providers to specify requests and offers with QoS properties and QoS classes,
- enabling an efficient service lookup and selection in order to accelerate the overall lookup process for service requestors,
- providing a flexible way for service providers to publish and update their service offers with different QoS aspects and parameters, as well as

- considering the QoS requirements regarding different layers and participating domains of a Web service communication process at runtime in order to achieve overall performance gains.

4.2 WS-QoS XML schema for QoS specification

For service providers and consumers it is essential to agree on a common language for defining QoS. Furthermore, an agreement ensures the compatibility and comparability of both offers and requirements, resulting in easy and prompt lookup and selection of services.

Service providers can specify the price for the service usage, supported protocols for management and monitoring as well as statements about different QoS levels of the offered operations of a service. Statements about the server performance, transport priorities, and references to offered security mechanisms can either be declared on operation or service level. A service consists of one or more operations.

The root element of a WS-QoS document is *wsqos*, which contains another element *definition*. As shown in Figure 11, a definition element can contain a *requirement*, *offers* or *ontology* element.

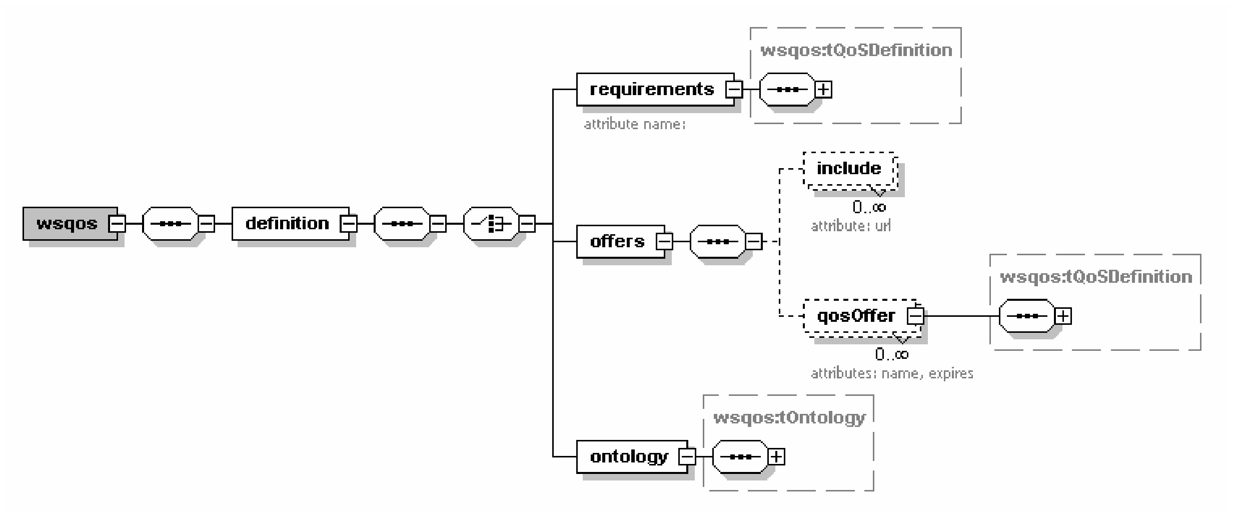

Figure 11. Structure of a wsqos element

The element *requirements* is of type *tQoSDefinition* and describes the QoS requirements of a service client. The values declared here represent the minimum requirements a service provider has to support.

QoS offers in form of a *qosOffer* element can be stated in an *offers* element. A QoS offer describes the minimum QoS level a service provider shall guarantee for the event that the client is willing to pay for the service. *qosOffer* is of type *tQoSDefinition,* ensuring that both offers and requirements have the same structure. This allows easy checking whether an offer fulfills a declaration of requirements.

The *qosOffers* element is extended with an *expire* attribute indicating the expire time of a service offer. The *include* element of the offers allows including QoS offers defined in other documents, and therefore allows for dynamically adjusting offers without changing the WSDL file. Furthermore, an offer can be referenced from multiple WSDL files and thus be reused for different services.

The *ontology* element defines QoS parameters and protocol references that are used in elements of type *tQoSDefinition*.

Standard QoS aspects such as *serverQoSMetrics*, *transportQoSPriorities*, or *securityAndTransaction* and their parameters are predefined. These can be enhanced by custom parameters, referring to a public WS-QoS ontology. Therefore, a *WSQoSOntology* element has been designed to hold references to QoS parameters and protocol (refer to section 4.2.2).

WS-QoS allows service providers to specify various QoS classes for the same service e.g. platinum, gold, and bronze, as shown in Table 2. Different QoS classes of Web services enable clients to choose the service that meets their best requirements. Classes of service may differ in any QoS aspects such as server performance, network performance, security including access rights, and consequently price [36].

Table 2. Example of different QoS classes

Class of service	**Platinum**	**Gold**	**Bronze**
Processing time	0.3ms	0.7ms	0.9ms
Throughput	200 request/s	150 request/s	100 request/s
…	…	…	…
Price per call	0.05€	0.03€	0.01€

WS-QoS also supports different QoS classes for operations of a service. The service provider can assign different QoS classes to an operation inside a service. Figure 12 shows an example of a service, which is composed of service operations with possibly different QoS classes.

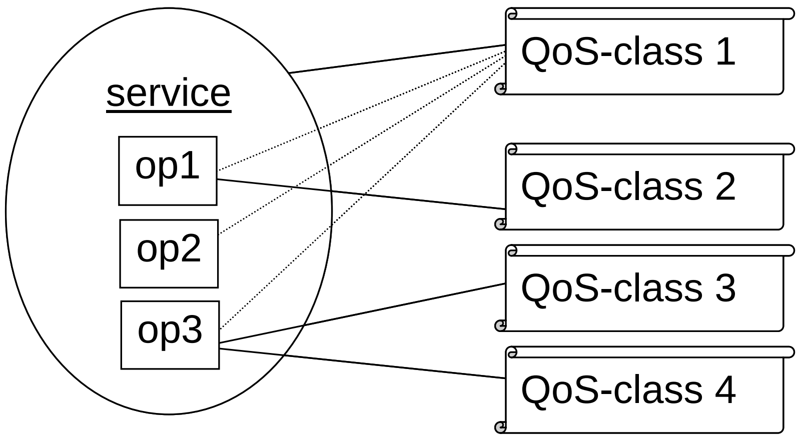

Figure 12. Assignment of QoS classes to a service or its operations

4.2.1 QoS Info

Central to the WS-QoS framework's XML Schema are elements of type *tQoSInfo*, shown in Figure 13. A *qosInfo* element holds information on the level of QoS regarding the server performance, transport and protocols. In a *serverQoSMetrics* element, values for the standard parameters such as processing time, requests per second, reliability, and availability can be declared. Moreover, custom server QoS metrics can be declared in a *customMetric* element as a child node of the *serverQoSMetrics* element, as shown in Figure 14.

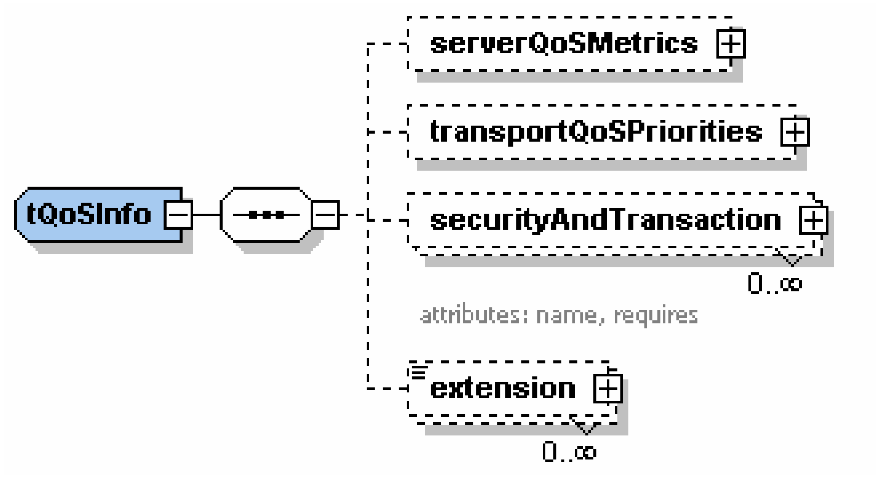

Figure 13. Structure of a qosInfo element

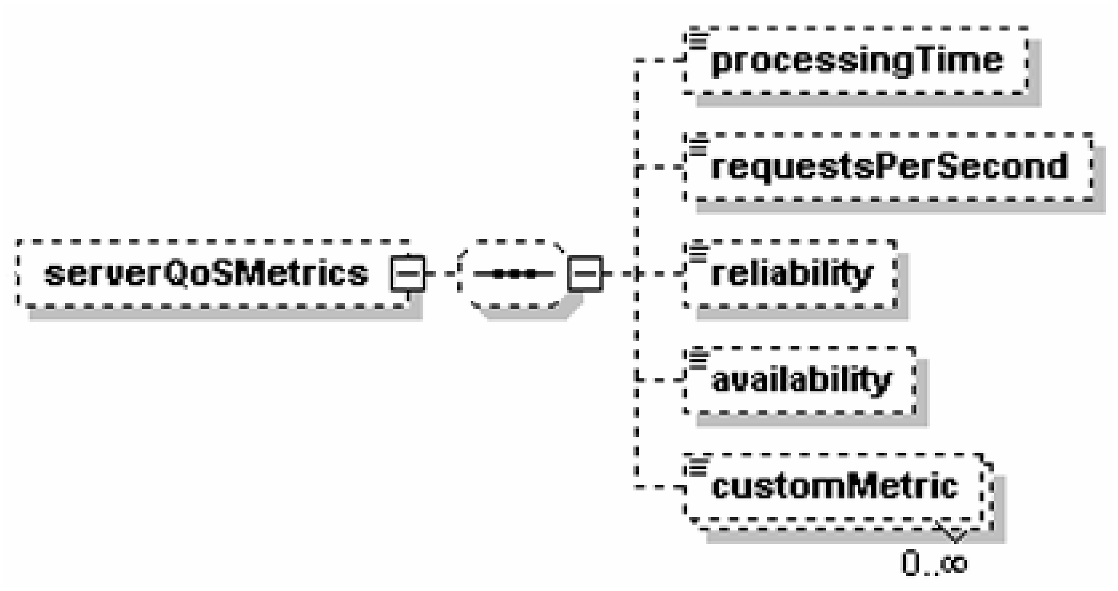

Figure 14. Structure of a serverQoSMetrics element

A *transportQoSPriorities* element, depicted in Figure 15, encapsulates information on priorities that can be declared for four standard transport parameters delay, jitter, throughput, and packet loss rate. Similar to the server QoS metrics, custom transport QoS priorities can be declared in a *customPriority* element added to a *transportQoSPriorities* element.

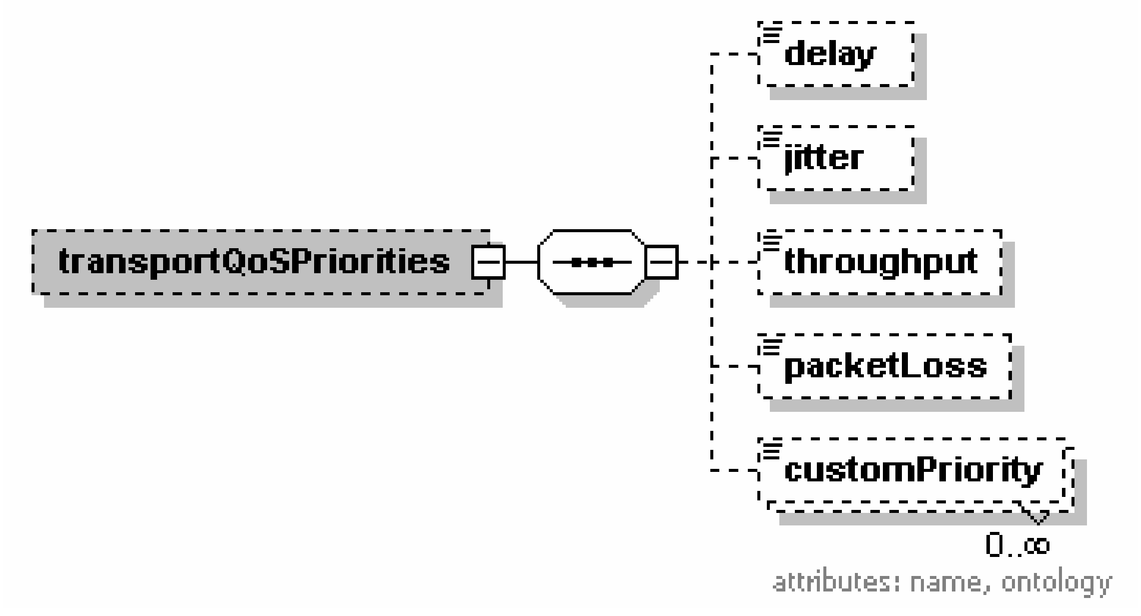

Figure 15. Structure of a transportQoSPriorities element

In most cases, neither the client nor the service provider knows in advance what kind of network technology will be used to exchange messages. Therefore, it is not appropriate to declare explicit values for metrics such as delay and jitter. In stead of absolute values priorities are declared for transport QoS. The priorities are mapped onto specific metrics of the underlying network at runtime helping to provide a distinct transport service level. (Refer to Section 4.4.1)

Security and transaction management for Web Services is realized by a variety of protocols. Most of them already have sophisticated mechanisms for negotiation of key and session information. Thus, security and transaction support at this level will be restricted to listing protocols needed for a successful service execution. The *securityAndTransaction* element of a *QoSinfo* element can hold several *protocol* elements, each referencing a specific protocol. A reference to a protocol in a *qosInfo* element can either require or offer compliance with a protocol in question. In the first case, another *qosInfo* element will not be compliant with the first specification if it does not at least offer using this protocol as well. In the later case, the protocol is offered in case the other party expects it, but interaction without the protocol is also allowed. The *qosInfo* element also allows the definition of custom QoS aspects in the *extensibilityElement*.

4.2.2 WS-QoS ontology

customMetrics, *customPriority* and *protocolSupport* statements all have an attribute *ontology*, which references a file containing a WS-QoS Ontology where the referenced types are defined respectively. Figure 16 shows the structure of such ontology. By using a combination of the ontology's URL and a parameter name, the reference will be unique.

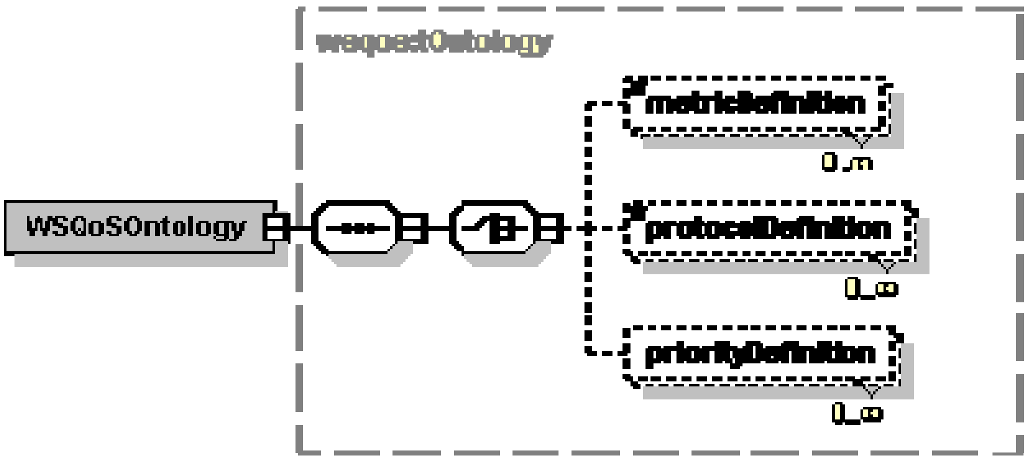

Figure 16. Structure of a QoSOntology element

A custom transport QoS priority is defined by a distinct name and a human readable definition of what metric the priority refers to in a *priorityDefinition* element. A custom server QoS metric defined in a *metricDefinition* element, as shown in Figure 17, also has a name and a human readable description of what is measured. It also includes information on the unit it is measured in and the scope of service invocations the metric is aggregated on, that is, whether the value is valid for the port on which the service is invoked, the whole service or even all services of the provider. Furthermore, it has to be stated whether the value is valid for all service executions or only for executions requested by the user. Finally, the direction of how values are to be compared is declared, which is essential for an automated comparison of whether an offer fulfills a set of requirements. Accordingly, in a *protocolDefinition* element, a protocol is defined by its name, a human readable description of the purposes of using this protocol and the URL of an overview document.

```
<xs:complexType name="tMetricDefinition" abstract="false">
      <xs:simpleContent>
         <xs:e4xtension base="xs:string">
           <xs:attribute name="name" type="xs:string" use="required"/>
           <xs:attribute name="direction" type="wsqos:tDirection"
use="required"/>
           <xs:attribute name="unit" type="wsqos:tUnit" use="required"/>
           <xs:attribute name="percentile" type="wsqos:tPercentage"
use="optional"/>
           <xs:attribute name="dataOwner" type="wsqos:tDataOwner"
use="required"/>
           <xs:attribute name="dataScope" type="wsqos:tDataScope"
use="required"/>
           <xs:attribute name="measurementIntervalLengthInSec"
type="xs:float" use="required"/>
           <xs:attribute name="description" type="xs:string"
use="optional"/>
      </xs:extension>
      </xs:simpleContent>
</xs:complexType>
```

Figure 17. tMetricDefinition

4.2.3 QoS definition

An element of type *tQoSDefinition,* as illustrated in Figure 18, holds one or more QoS info elements, plus specifications of the contract and management support and a specific price. QoS information can be defined for specific operations either in explicit *operationQoSInfo* elements or, for the scope of all operations of a service, in a *defaultQoSInfo* element. Both, the *defaultQoSInfo* and the *operationQoSInfo* elements are of the type *tQoSInfo*. The *contractAndMonitoring* node can hold references to protocols needed for service management, SLA, and/or QoS monitoring as well as entries of third parties that one side would be willing to trust. Finally, the *price* element relates the specified QoS level to the cost of service usage per invocation.

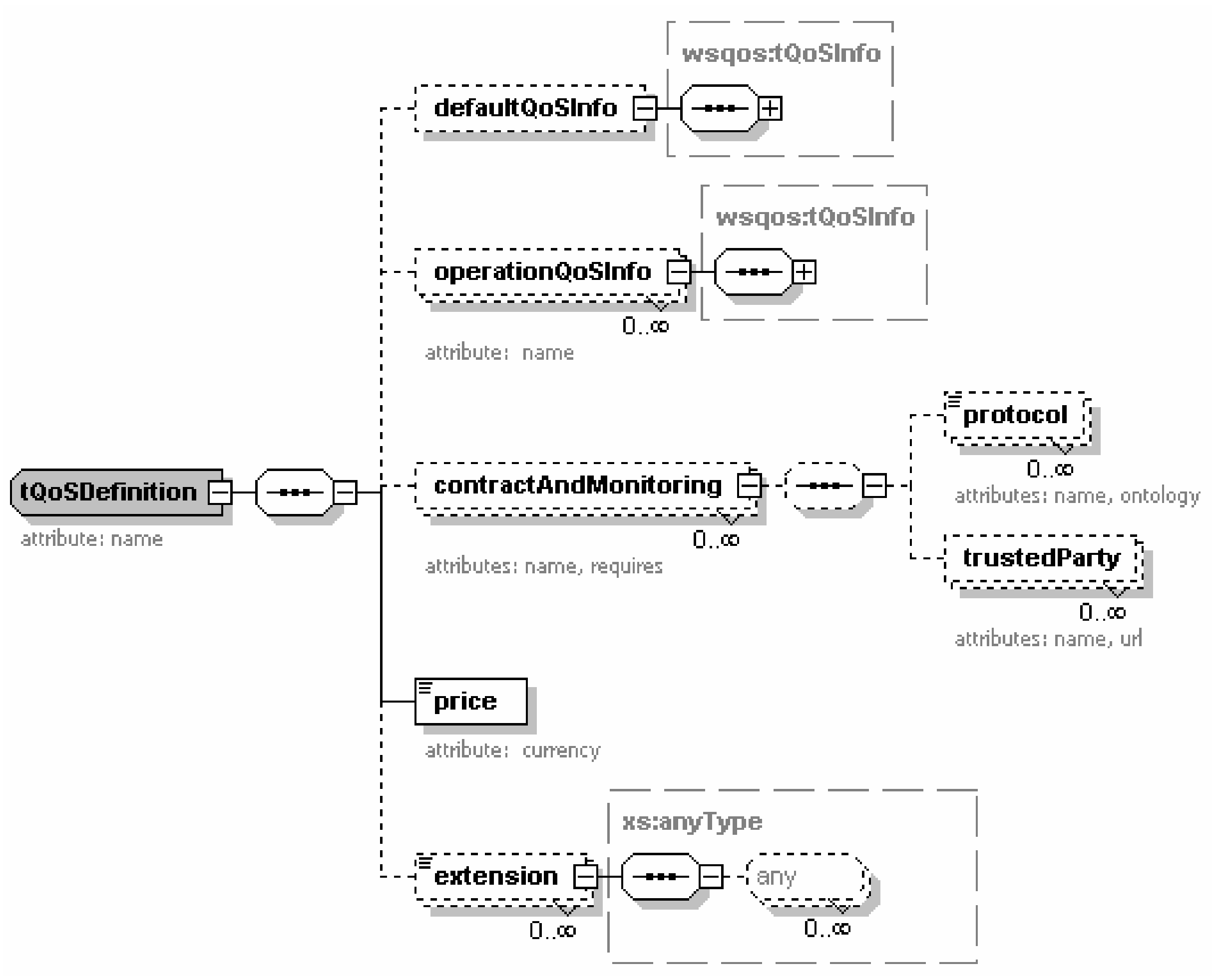

Figure 18. Structure of a tQosDefinition element

Elements of type *tQoSDefinition* are either instantiated as a *WSQoSRequirementDefinition* element expressing a client's QoS requirements or as a *qosOffer* element representing a minimal QoS level a service provider guarantees to provide for all requests.

4.2.4 WS-QoS offer definition

Offers for one service can be declared in a *WSQoSOfferDefinition* element as shown in Figure 19. The *qosOffer* element is extended by an attribute *expires,* denoting a point in time until which the offer will be valid.

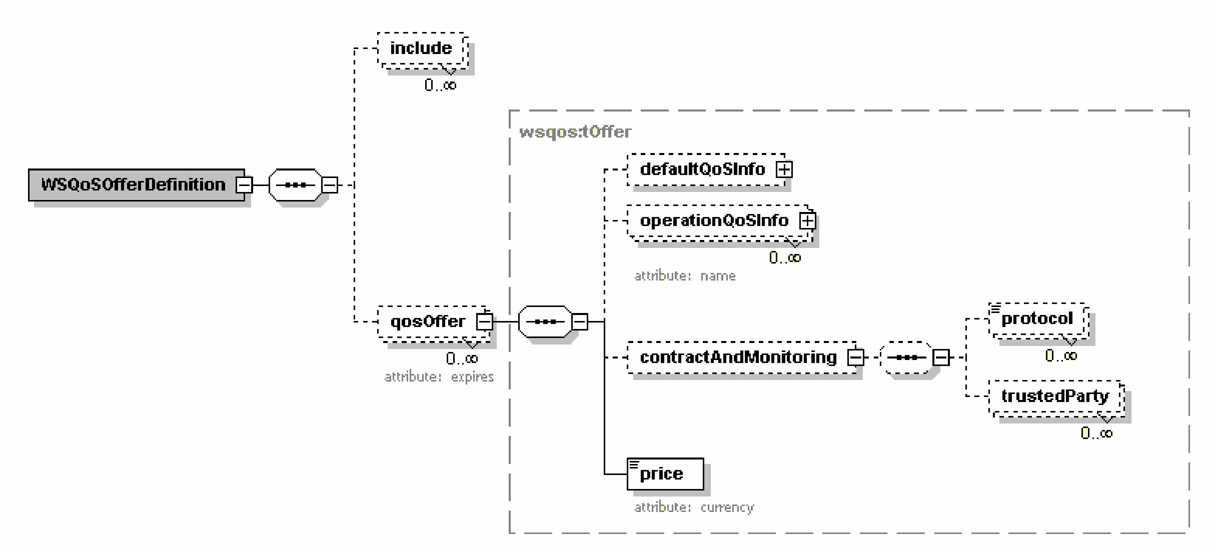

Figure 19. Structure of a WSQoSOfferDefinition element

The *WSQoSOfferDefinition* element is introduced into the Web service's WSDL file as an extension element of the service description's service node. Apart from defining offers in a *WSQoSOfferDefinition* element, offers in further WS-QoS files can be referenced in an *include* element, as shown in Figure 20. This allows for dynamically adjusting offers without changing the WSDL file. Furthermore, an offer can be referenced from multiple WSDL files and thus be reused for different services.

```
<wsqos xmlns="http://wsqos.org/">
     <definition>
      <offers>
       <include url = "http://lab3.pcpool/StockQuoteServices/
CurrentOffersForFastStockQuoteService.wsqos" />
      </offers>
     </definition>
</wsqos>
```

Figure 20. Further WS-QoS definition references in an include element

4.3 QoS-aware service discovery and selection

The UDDI registry defined by UDDI.org does not support QoS-aware service lookup. One mechanism to extend UDDI with QoS-awareness would be to augment UDDI with corresponding business logics. However, then UDDI would become proprietary; both client and server applications have to be re-implemented.

Due to these disadvantages, we have introduced a Web service broker (WSB), which is located outside UDDI. The WSB is WS-QoS aware. Its main task is to accelerate the client lookup process for appropriate services. Figure 21 depicts the participating roles service providers, clients, UDDI registries, and the WSB. Note that there are no service brokers defined in the standard Web service model [38].

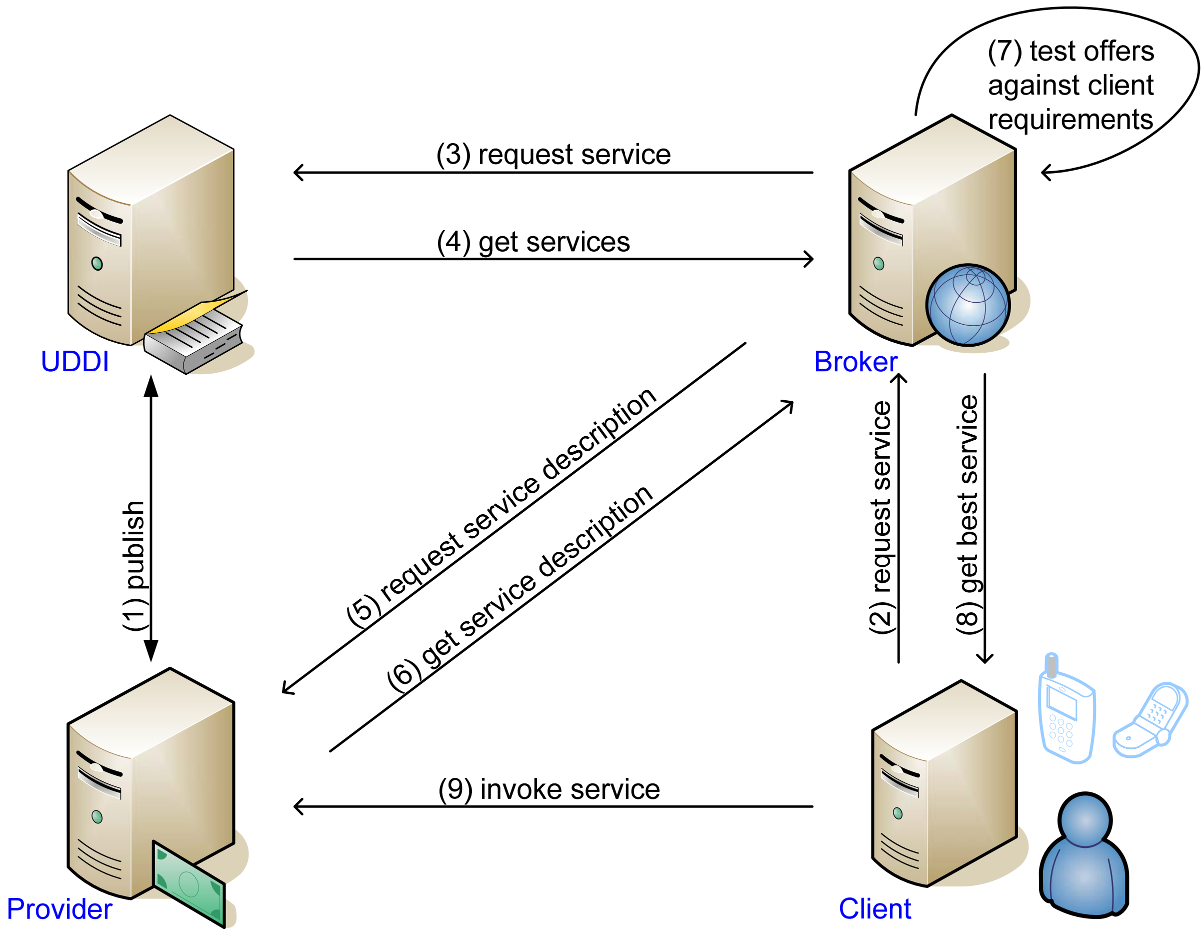

Figure 21. Interactions between the four participating roles

When the WSB does not have any information about a required service, the interactions between the roles are as follows.

1. Service providers publish their Web services to UDDI registries. Web services available in UDDI registries are identified uniquely by an interface key (*tModel*).

2. Clients ask the WSB for services that implement a certain interface and accomplish the QoS requirements.

3. If the WSB does not already hold up-to-date information on offers that accomplish clients' requirements, the WSB will request Web services according to the interface key from one or more UDDI registries. Note that we would prefer the model in which the WSB prefetches information of offers that clients could be interested in. This would accelerate the lookup phase significantly.

4. The UDDI registries return a list of services that implement the interface key.

5. The WSB asks the service providers for service descriptions, e.g. WSDL files.

6. The service providers return their service descriptions with QoS offers.

7. The WSB tests the offers against the clients' requirements, which the client sent in step 2.

8. The WSB returns the most appropriate service to the client.

9. The client directly invokes the service with the original QoS requirements. At this time, the QoS requirements regarding the network performance are actively mapped onto the underlying transport technology, as described in Section 4.4.1.

We assume that service brokers normally analyze the market and interesting service offers in advance. The WSB holds up-to-date information on offers currently available for a group of services. Therefore, a Web service client will contact the WSB for looking up a service instead of doing this with a UDDI registry. That means the interaction model from the client's point of view is reduced from 9 steps to 4 steps (step 2, 7, 8, and 9) as shown in Figure 22. The reduction of the interaction steps results in short lookup time. Section 7.2 presents our performance measurements proving this statement.

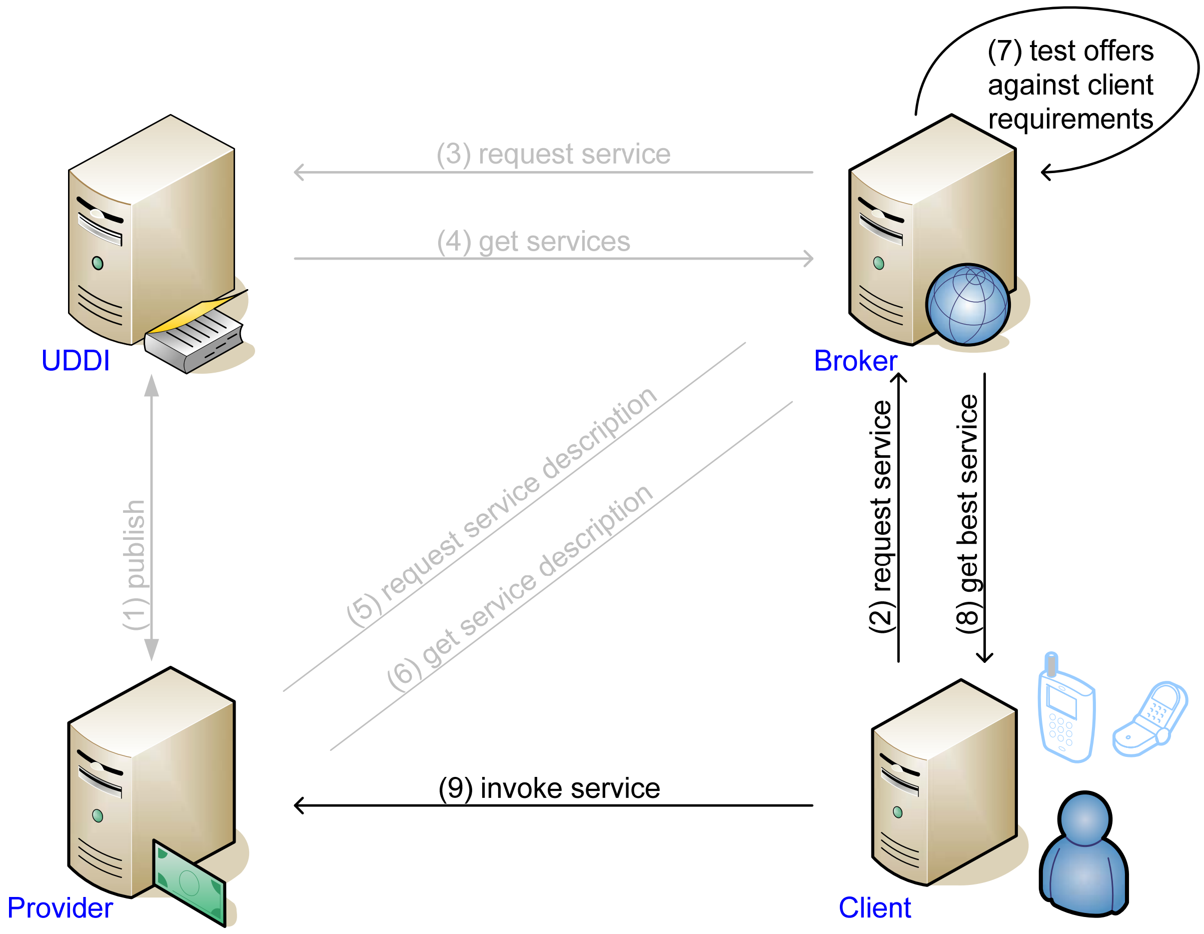

Figure 22. Reduced interactions between the four participating roles

There are two options to apply the WSB. One is to run it as a local object within the (client) application. This ensures a highly performing service selection and detailed information on available offers. The other possibility is to use it as a remote Web service to obtain the access point of the most appropriate service. This version is mainly intended as a (Web) service for multiple client applications that could use a single private WSB running within their network domain. This WSB could be used by any other WS-QoS compliant implementation, too.

4.4 QoS-aware service invocation

After defining QoS requirements and the selection of the appropriate service provider, the client invokes the service. Whereby, it is important that QoS-awareness is supported by all participating domains and layers along the whole communication path. QoS requirements specified by the clients are placed as a WS-QoS element in the SOAP header. Since the SOAP header is a part of SOAP messages it can be parsed, evaluated, modified, and mapped by WS-QoS aware components along the communication process enabling cross-layer communication. Figure 23 shows an example of WS-QoS requirements located in the SOAP header.

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <soap:Header>
  <WSQoSSoapHeader xmlns="http://www.wsqos.net/BookInformation">
   <selectedOffer>Quick</selectedOffer>
   <qosInfo>
    <operationQoSInfo xmlns="">
     <serverQoSMetrics>
      <processingTime>0.5</processingTime>
      <requestsPerSecond>10</requestsPerSecond>
      <availability>0.77</availability>
      <reliability>0.77</reliability>
     </serverQoSMetrics>
     <transportQoSPriorities>
      <delay>9</delay>
      <jitter>8</jitter>
      <throughput>8</throughput>
      <packetLoss>7</packetLoss>
     </transportQoSPriorities>
     <securityAndTransaction name="general" requires="none">
      <protocol name="SOAP" />
     </securityAndTransaction>
     <securityAndTransaction name="Compression" requires="one">
      <protocol name="ZIP" ontology=
"http://localhost/BookInformationServices/CustomOntology.wsqos" />
     </securityAndTransaction>
    </operationQoSInfo>
   </qosInfo>
  </WSQoSSoapHeader>
 </soap:Header>
 <soap:Body>
 <DynamicResultString xmlns="http://www.wsqos.net/BookInformation">
  <resultsize>0</resultsize>
  </DynamicResultString>
  </soap:Body>
</soap:Envelope>
```

Figure 23. QoS requirement in the SOAP header

In the following subsections, we discuss the QoS mapping between the application and transport layer, QoS support in web server, and load reduction through message load compression.

4.4.1 Mapping transport priorities to QoS-aware network technologies

Since different network technologies have different properties and QoS metrics respectively, we have decided to define priorities ranging from 1 to 10 in a *transportQoSPriorities* element whereby 1 represents the highest priority. The smaller the priority value the higher the requirement. A so called Adaptation Layer placed between the Web service layer and communication network layer is responsible for the mapping. As shown in Figure 24 the Adaptation Layer evaluates and maps the specified requirements from the higher layer onto a corresponding traffic class or properties, which are specific to a transport technology.

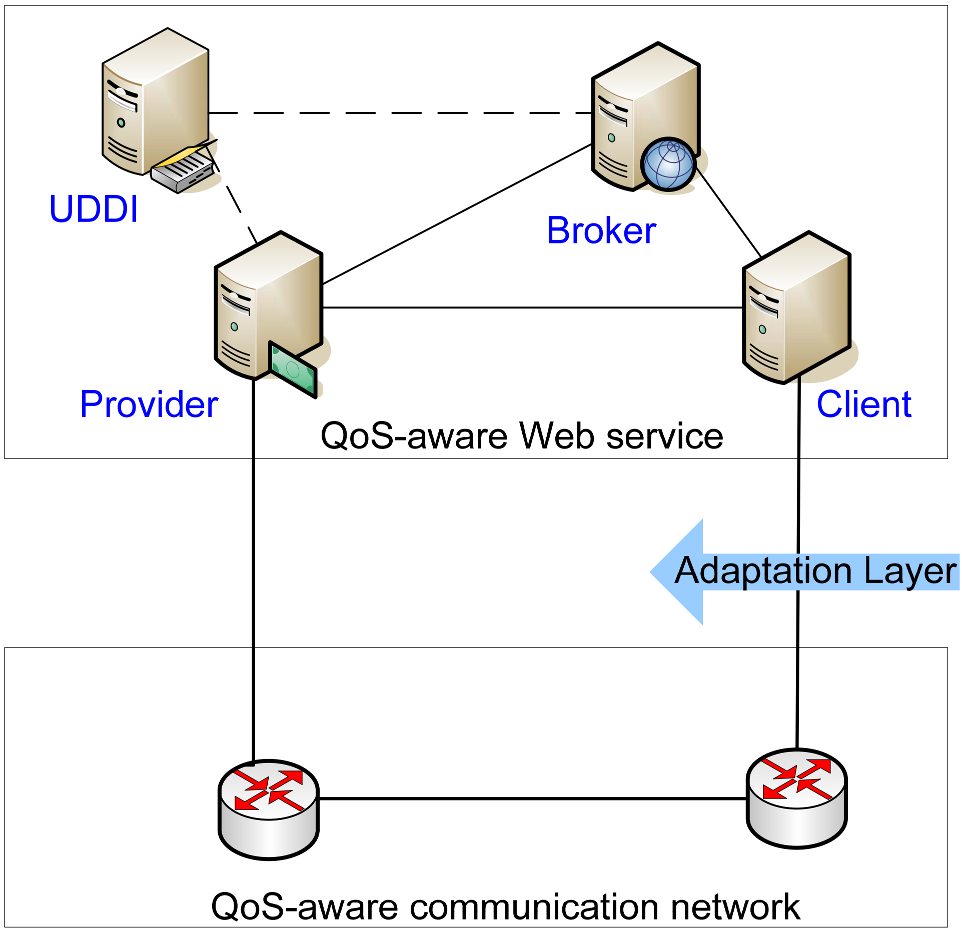

Figure 24. QoS mapping in the Adaptation Layer

Figure 25 shows a *transportQoSPriorities* entry in a WS-QoS operation information element defined in the WS-QoS XML schema for *MyOperation()*. In this case, the operation expects low data rate, high reliability, and low jitter. One could for example think of a low quality voice stream.

```
<?xml version="1.0" encoding="utf-8" ?>
<wsqos xmlns="http://wsqos.org/">
  ...
  <operationQoSInfo name=”MyOperation”>
    ...
    <transportQoSPriorities>
      <delay>5</delay>
      <jitter>3</jitter>
      <throughput>8</throughput>
      <packetLoss>3</packetLoss>
    </transportQoSPriorities>
    ...
  </operationQoSInfo>
  ...
</wsqos>
```

Figure 25. A WS-QoS transportQoSPriorities entry

Figure 26 shows an example of how mobile Web services could benefit from the QoS mapping [40]. The scenario consisting of a mobile client, a wireless network such as UMTS/3G or GPRS, an access point, a QoS-enabled wired network such as DiffServ and a WS-QoS-aware Web service server. The adaptation layer running on the mobile client translates the QoS requirements according to the transport QoS priorities to the corresponding UMTS QoS class and performs signaling with the UMTS system. Since both DiffServ and UMTS support QoS classes, the access point (AP) can now map the UMTS QoS class to a corresponding DiffServ class (DiffServ code point, DSCP). This task is performed without any knowledge of the WS-QoS framework. Optionally, if the AP supported WS-QoS, it could map the client’s requirement to a corresponding DSCP by evaluating the WS-QoS information located in the SOAP header. The advantage of this option is that the mapping would be fine-granular. However, the disadvantage is the performance loss due to processing the SOAP header. Since this header is only present in the first of several IP packets carrying a SOAP message, the AP has to perform per-flow management. This would cause scalability problems as experienced in the Integrated Services technology (IntServ). Therefore, the simple mapping of UMTS and DSCP classes should be preferred when the AP experiences high traffic load.

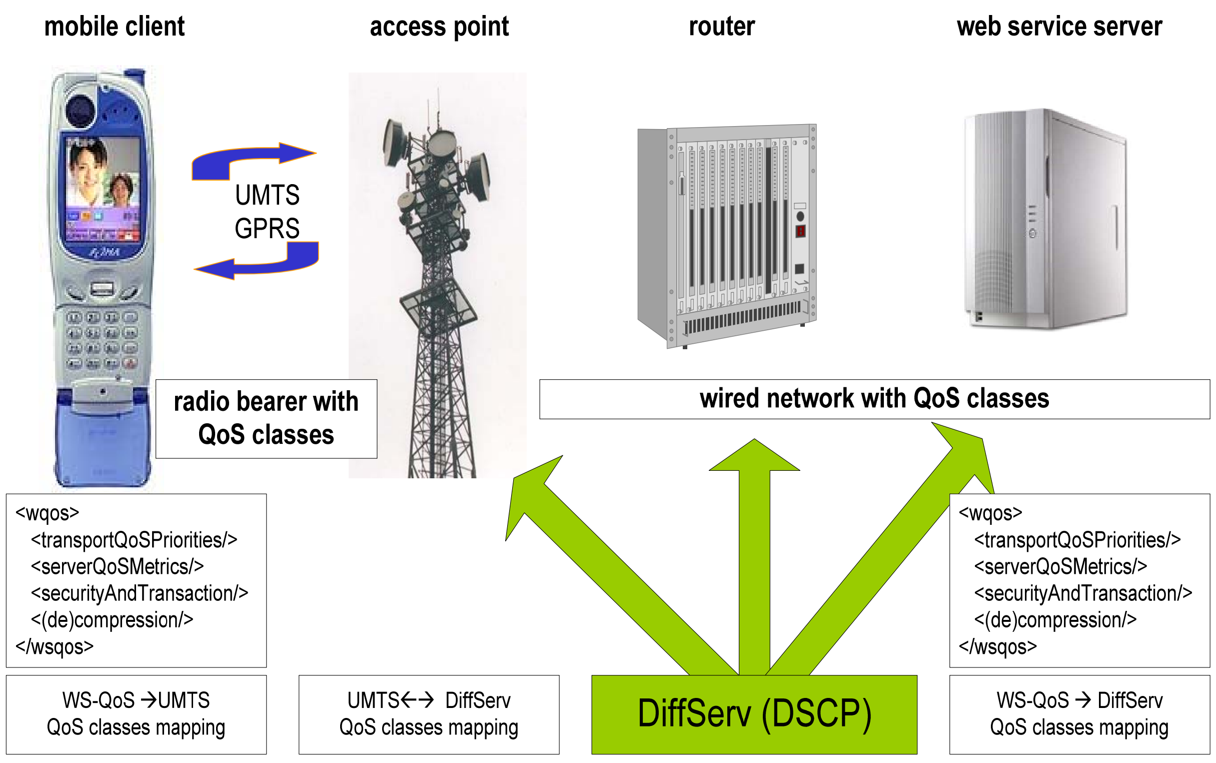
mobile client
access point
router
web service server
UMTS
GPRS
radio bearer with QoS classes
wired network with QoS classes
<wqos>
<transportQoSPriorities/>
<serverQoSMetrics/>
<securityAndTransaction/>
<(de)compression/>
</wsqos>
<wqos>
<transportQoSPriorities/>
<serverQoSMetrics/>
<securityAndTransaction/>
<(de)compression/>
</wsqos>
WS-QoS →UMTS QoS classes mapping
UMTS←→ DiffServ QoS classes mapping
DiffServ (DSCP)
WS-QoS → DiffServ QoS classes mapping

Figure 26. A mobile Web service scenario

The intermediate DiffServ-enabled routers treat the traffic depending on the DSCP. Upon receiving the client's request, the server processes the response. The server has to consider the client's requirements for the server performance defined in the *serverQoSMetrics* (refer to the next subsection). When the server sends the response, it will put the client's QoS requirements into the SOAP header again. A server side Adaptation Layer will then evaluate the QoS information and mark the DSCP in each IP packets accordingly. The intermediate routers will treat the IP packets according to the DSCP. The AP will map the DiffServ class to a corresponding UMTS class.

The concrete implementation of the Adaptation Layer for DiffServ will be discussed in Section 5.7.

4.4.2 Adaptive server performance

Service differentiation can take place on various levels at a web server, such as on the TCP [18], HTTP [18], end systems [17] or application level. One simple solution on the application level is to set the priority of the thread processing the request according to the clients' requirements. 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 the OS level [39]. Such an approach guarantees prioritized access to a limited scope of resources even in overloaded servers.

The *serverQoSMetrics* element of a WS-QoS definition shown in Figure 27 specifies server performance in terms of processing time, throughput, availability and reliability. This definition can be applied to service offers or requirements. In contrast to the values set for the *transportQoSPriorities* element absolute values are set here.

```
<?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 27. A WS-QoS serverQoSMetrics element.

4.4.2.1 Processing time

The processing time is defined as the time interval between the point when a request arrives at the server process and the point when the server process sends the response.

Servers become overloaded when one or several critical resources become scarce. Server overload affects the server processing time. Figure 28 schematically illustrates the processing time as functions of the request rate. It demonstrates how the processing time increases with the server load. The processing time is low as long as no server resource is over utilized. However, when the server resource bottleneck becomes over utilized, i.e., the bottleneck resource cannot keep up with the arrival rate of requests, the queue length to the resource bottleneck and thus the response time theoretically increases to infinity. This is depicted by the sudden increase of the response time [18].

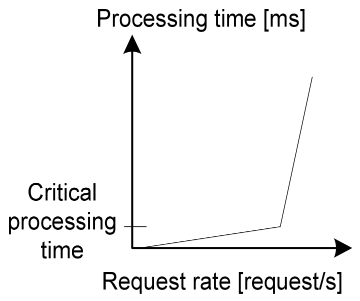

Figure 28. Impact of server load on processing time [18]

4.4.2.2 Throughput

Throughput means the amount of requests a server can process in certain time unit.

Sever overload affects either server throughput. Figure 29 depicts how the server throughput increases with the request rate until the request rate exceeds the capacity of the web server. At this point, the throughput decreases due to the additional and unproductive time the CPU spends on processing incoming connection requests that are dropped when the listen queue is full. Moreover, the high rate of network interrupts prevents the web server application from making fast progress, which contributes to the lower throughput. Lower server throughput leads to loss of revenue, while long delays cause user frustration and decrease task success and efficiency. Users' tolerance for delay is application dependent, but often a threshold of 10 seconds for web interaction is mentioned in the literature [18].

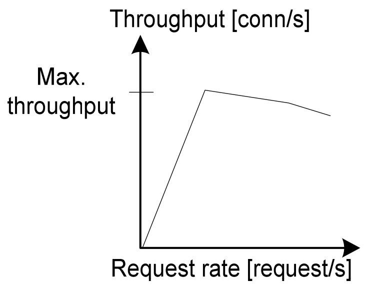

Figure 29. Impact of server load on throughput [18]

4.4.2.3 Reliability

Reliability refers to the general likelihood of an error occurring in a running server system. A perfectly reliable server will enjoy 100% availability, but when errors occur, availability can be influenced in different ways depending on the nature of the problems.

4.4.2.4 Availability

Availability means the probability that the system is operating properly when it is requested for use at a given time. The definition of availability is largely based on what types of downtimes, e.g. server overloading or malfunction, results in system unavailability.

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

It is well known that SOAP messages are verbose in comparison to binary protocols. The overhead of Web services stems mainly from the XML usage producing human readable text. Mani and Nagaranja have compared the XML's way to represent information with binary encoding. They quantified the overhead as 400% [41]. The growth of the Web service message size, which results in higher transmission time, creates a critical problem for delay sensitive applications, especially when XML data is transferred over the air and mobile devices are involved.

Although mobile devices are resource-constrained, the capability of mobile hardware in terms of CPU power and memory is increasing rapidly. But the improvement and increase of the battery life-time and the data rate for wireless transmission are still challenging issues in active research. Therefore, considering both aspects in mobile computing is essential.

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 a test case, a 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 (referred to as "ASP overhead" in Figure 30). The total amount of the request and response for transferring the same information value is about 1200 bytes.

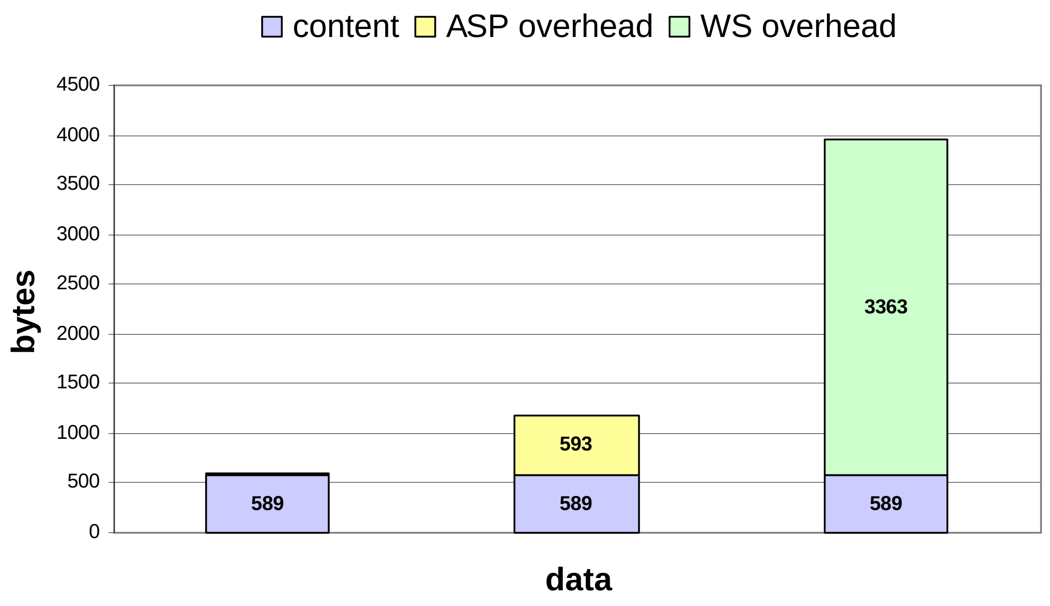

Figure 30. XML overhead for a simple request

Due to the verbosity of Web services, compressing the SOAP content before the wireless transmission is attractive. As shown in [42], compression is one way of dealing with the problem of large message sizes of Web services. Furthermore, compression is useful for poorly connected clients with resource-constrained devices despite the CPU time required for decompressing the responses. 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]. Furthermore, wireless transmission of a bit can require 1000 times more energy than a single 32-bit computation [43].

The energy consumed for (de)compression on servers is almost for free. Therefore, mobile clients should request data from the server in a format which facilitates low-energy decompression in order to reduce decompression energy [43].

In order to signal which compression algorithm is to be used the *securityAndTransaction* of the *operationQoSInfo* element in the WS-QoS XML schema is extended with two sub nodes compression and decompression. Server providers announce which compression algorithms they support. Clients define which compression algorithm a server has to use to compress responses. Algorithms are listed in the order of preference, so the most appropriate match can be found. Figure 31 shows example of the extension of the *securityAndTransaction* node with the new QoS parameters *compression* and *decompression*.

```
<?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 31. securityAndTransaction entries for compression and decompression algorithms.

Compression generally decreases server performance due to the additional CPU time required. A lightly loaded server can afford the extra cost of compressing responses. We will present measurements in Section 7.1 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 will 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 will present experiments that demonstrate that this approach works well. Both servers and clients with poor connectivity benefit during high server demand, while the server is protected from overload due to compression.

4.5 Conclusion

The WS-QoS XML schema is the core of the WS-QoS architecture. The WS-QoS XML schema enables the specification and thus the compatibility and comparability of QoS statements defined by both service clients and servers. All components participating in Web service communication such as WSB, Adaptation Layer, web servers apply the WS-QoS XML schema in order to provide QoS support in different layers and domains.

Three steps are defined in a Web service communication process from the client's point of view. They are the definition of requirements, service discovery and selection, and service invocation. The WS-QoS architecture ensures QoS-awareness during the whole Web service communication process resulting in a QoS-aware cross-layer communication.

In contrast to the classical ISO/OSI layered architecture that does not consider the inter-working of different layers, the cross-layer communication model has several advantages:

- Higher layers have knowledge about the parameters and routing algorithms of underlying network technologies
- Higher layers have knowledge about the current communication structures and their dynamics
- Resulting in higher layers can actively consume the QoS support of the low layers
- Lower layers have knowledge about the specific requirements from the higher layers
- All the knowledge can be merged in respect to QoS parameters of different aspects in order to support application-dependent requirements.