6 Applying the WS-QoS framework

This chapter describes how to apply the implementation of the WS-QoS framework introduced in Chapter 5.

When implementing a WS-QoS compliant Web service three issues have to be taken into consideration: First, the service should implement a generic service interface (e.g. *tModel*), which already defines the use of an optional WS-QoS SOAP header to transport QoS information within service requests and responses. Second, the service has to implement a strategy to provide WS-QoS offers, which should be adjustable to changing situations of service utilization. Finally, to achieve the QoS level(s) associated with distinct offers, the selected offer and further QoS requirements have to be evaluated when receiving a request. Apart from developing new services one can also qualify existing service implementations as WS-QoS compliant by making just a few alterations to the code [44].

6.1 Implementing an abstract service description

To make the service available for dynamic look-up through the WSB, one should implement a specific *tModel*, which specifies a WS-QoS compliant Web service with the functionality of the service. The more common a *tModel* is, the more attractive it is for a client to search services implementing the *tModel*. Therefore, we encourage reusing an abstract service description rather than inventing a redundant interface.

If no such *tModel* exists, one has to specify a new one. The easiest way to create a *tModel* is to implement a dummy service featuring a WS-QoS SOAP header as described below and call up the WSDL file for this service. Clients will be able to generate a proxy object from this interface and use the proxy for various services only by setting the proxy's access point URI to the location of a concrete implementation. In any case, concrete services should use the same XML namespace as used in the service interface.

6.2 Managing WS-QoS offers

The QoS level provided by a service is part of its nature. WS-QoS offer definitions are therefore expected to be held in a WS-QoS extension to the service's WSDL file. This allows for a fast discovery of WS-QoS offers, since a service's description file is commonly referenced from its entry in a UDDI registry along with further information such as the service access point. The WS-API provides several ways to publish WS-QoS offers, depending on the personal strategies and requirements. It should be decided whether

- WS-QoS offers are held directly in WSDL or a further WS-QoS file is referenced from the WSDL's WS-QoS extension,
- the validity period of WS-QoS offers is updated manually or by the WS-QoS Offer Manager Object provided by the WS-QoS API,
- the service description referenced is a static WSDL file or XML generated by a server-side engine.

6.2.1 Including WS-QoS files from WSDL extension elements

To enable changes to QoS offers, offers should only be valid for a restricted period of time and then be renewed regularly. Frequent changes of a WSDL file are inappropriate, since most clients use the service description once to build a proxy object and then expect the service to keep its functionality stable. By defining an include element, an external WS-QoS file can be referenced from a WS-QoS extension, as the following code sample from a WSDL file MyService.wsdl referencing an external WS-QoS definition in a file MyOffer.wsqos shows:

```
<?XML version="1.0" encoding="utf-8"?>
<definitions XMLns="http://schemas.XMLsoap.org/wsdl/">
  ...
 <service name="MyService">
    <port name="MyServiceSoap" binding="s0:MyServiceSoap">
      <soap:address
     location="http://www.mydomain.com/MyServices/MyService.asmx"/>
    </port>
     <wsqos XMLns="http://www.wsqos.net/schemas/">
      <definition>
        <offers>
          <include url =
"http://www.mydomain.com/MyServices/MyOffer.wsqos"/>
        </offers>
      </definition>
     </wsqos>
 </service>
</definitions>
```

This way, changes are applied to the wsqos file rather than the WSDL file. Changes to WS-QoS offers can be made either manually or by employing the WS-QoS Offer Manager. Note that in some cases accessing *.wsqos files from a server has to be explicitly permitted in the server configuration.

6.2.2 Updating WS-QoS offers with the WS-QoS offer manager

Updating offers manually can become tiresome if service providers want to keep validity periods short to adjust the offers to changing situations. For the purpose of an automated offer update the WS-QoS API provides the class *WSQoSAPI.WSQoSOfferManager*. Its constructor receives three parameters:

As a first parameter, the type of the service is passed on. This type is used to retrieve custom attributes declared on the Web service class by means of reflection. These attributes hold information on paths to source files specifying QoS levels. While an *ImportOffers* attribute provides offers, which are merely to be copied into an assembly of up-to-date offers, an *ImportQoSDeclaration* attribute specifies a QoS level in the form of a *tQoSDefinition,* which has to be turned into a WS-QoS offer by applying a validity period.

The second parameter specifies the path of a file, which is to hold up-to-date WS-QoS offers. If the file is WSDL, it is extended by a WS-QoS extension. Should such an extension already exist, its content is overridden. This way one can either directly manipulate a WSDL file, update a WS-QoS file, which is referenced from the WSDL file, or change a WS-QoS file that is used as a source file for the WS-QoS extension to the WSDL generator engine.

The third parameter of the type *int* specifies the time period of an update interval. This interval is also used as the validity period of updated offers.

In the following example three files each hold the QoS level specification for one offer in a requirements element. These definitions are updated manually and changes will be applied on the next offer update. To allow for additional files to be added, all files are referenced from the following file AvailableOffers.wsqos:

```
<?XML version="1.0" encoding="utf-8"?>
<wsqos XMLns="http://www.wsqos.net/schemas/">
  <definition>
    <offers>
      <include url="C:/Inetpub/wwwroot/MyService/offer1.wsqos"/>
      <include url="C:/Inetpub/wwwroot/MyService/offer2.wsqos"/>
      <include url="C:/Inetpub/wwwroot/MyService/offer3.wsqos"/>
    </offers>
  </definition>
</wsqos>
```

The location of this file is declared in a *WSQoSAPI.ImportQoSDeclaration* attribute, which is applied to the service class. This custom attribute is evaluated by the *WSQoSAPI.WSQoSOfferManager* object to gather QoS level definitions, turn them into WS-QoS offers and then place them into a wsqos extension element of the WSDL file, of which the location is passed to the Offer Manager. Note that when using this mechanism, services need to be 'triggered' once by an initial service invocation to start the offer update process, before they can be discovered on the basis of up-to-date WS-QoS offers. The following code sample shows how the components introduced are defined on a Web service class:

```
[ImportQoSDeclaration(Location="C:/Inetpub/wwwroot/MyServices/AvailableOf
fers.wsqos")]
public class MyService : System.Web.Services.WebService
      private static WSQoSAPI.WSQoSOfferManager MyOfferManager =
new WSQoSAPI.WSQoSOfferManager( typeof(MyService),
"C:/Inetpub/wwwroot/MyServices/MyService.wsdl",
120000 );
      public MyService() { ... }
      // public Web methods ...
} // end class MyService
```

6.2.3 Using the WS-QoS extension to the .NET WSDL generator engine

A WSDL description can either be defined as a static file or it can be generated on demand. The ASP/.NET environment provides a tool wsdl.exe to generate a service description. The engine is called by adding the suffix ?WSDL to the services URL, e.g. http://www.mydomain.com/MyService/MyService.asmx?WSDL. To insert the WS-QoS extension to the generated WSDL one needs to register the *WSQoSAPI.WSQoSExtension* in the project's web.config file, as the following example shows:

```
<?XML version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <webServices>
      <serviceDescriptionFormatExtensionTypes>
       <add type="WSQoSAPI.WSQoSExtension, WSQoSAPI"/>
      </serviceDescriptionFormatExtensionTypes>
    </webServices>
    ...
  </system.web>
</configuration>
```

The *WSQoSAPI.ImportOffers* attribute has to be declared on the Web service class to specify a file holding up-to-date WS-QoS offers for the service. Note that in the following code sample, the Offer Manager now updates the WS-QoS file that is used as a source for the WSDL generator extension rather than a static WSDL file.

```
[ImportQoSDeclaration(Location="C:/Inetpub/wwwroot/MyServices/AvailableOf
fers.wsqos")]
[ImportOffers( Path="C:/Inetpub/wwwroot/MyServices/CurrentOffers.wsqos" )
]
public class MyService : System.Web.Services.WebService
{
      private static WSQoSAPI.WSQoSOfferManager MyOfferManager =
new WSQoSAPI.WSQoSOfferManager( typeof(MyService),
"C:/Inetpub/wwwroot/MyServices/CurrentOffers.wsqos", 120000);
      public MyService() { ... }
      // web methods ...
} // end class MyService
```

The *WSQoSAPI.ImportOffers* attribute is then evaluated by means of reflection every time the generator engine is called. The WS-QoS extension to the engine then ensures that offers defined in the specified file are included to the generated WSDL in a wsqos extension element. Again, services need to be 'triggered' once by an initial service invocation to start the offer update process. Note that for an automated update of offers the right to write the files holding these offers has to be granted to the server process, in our case the IIS.

6.3 Adaptation of client's QoS requirements

A WS-QoS compliant Web service needs to define a member of the type *WSQoSAPI.WSQoSSoapHeader* so that QoS information passed to the service by a WS-QoS compliant client can be taken into account by the service instance. This WS-QoS SOAP header is referenced in a SOAP header attribute on each Web method, as the following code extract shows:

```
public class MyService : System.Web.Services.WebService
{
      public WSQoSAPI.WSQoSSoapHeader MyWSQoSHeader;

      [SoapHeader("MyWSQoSHeader", Direction=SoapHeaderDirection.InOut)]
      [WebMethod()]
      public foo MyMethod(foo MyParam)
      {
         if (this.MyWSQoSHeader != null) // apply QoS level according to
requirements
         else                        // apply default (lowest) QoS level
         return magicOperation(MyParameter);
      }
}
```

The level of QoS applied could be decided on the basis of the selected offer as in the following example which differentiates processing time by setting distinct priorities for the worker thread serving a request:

```
// handle WSQoS requirements
if (this.MyWSQoSHeader != null)
{
      switch (this.MyWSQoSHeader.selectedOffer)
      {
        case "Posh" : System.Threading.Thread.CurrentThread.Priority =
            System.Threading.ThreadPriority.AboveNormal; break;
        case "Standard" : System.Threading.Thread.CurrentThread.Priority
        = System.Threading.ThreadPriority.BelowNormal; break;
        default : System.Threading.Thread.CurrentThread.Priority =
        System.Threading.ThreadPriority.Lowest; break;
      }
}
else System.Threading.Thread.CurrentThread.Priority =
     System.Threading.ThreadPriority.Lowest;
```

Yet, since we receive the whole WS-QoS SOAP header, we can also treat clients on a more differentiated approach considering their individual parameters, as the following example shows:

```
// handle WSQoS requirements
if (this.MyWSQoSHeader != null)
{
  WSQoSAPI.QoSInfo requirements = new
  WSQoSAPI.QoSInfo(this.MyWSQoSHeader.qosInfo);
  if (requirements != null)
  {
       if (requirements.ServerQoSMetrics != null)
       {
            double processingTime =
requirements.ServerQoSMetrics.ProcessingTime;
            if      (processingTime < 2) currentThread.Priority =
                        System.Threading.ThreadPriority.Highest;
            else if (processingTime < 4) currentThread.Priority =
                        System.Threading.ThreadPriority.AboveNormal;
            else if (processingTime < 6) currentThread.Priority =
                        System.Threading.ThreadPriority.Normal;
            else if (processingTime < 8) currentThread.Priority =
                        System.Threading.ThreadPriority.BelowNormal;
else  currentThread.Priority = System.Threading.ThreadPriority.Lowest;
       }
       else currentThread.Priority =
System.Threading.ThreadPriority.Lowest;
  }
  else      currentThread.Priority =
System.Threading.ThreadPriority.Lowest;
}
else  currentThread.Priority = System.Threading.ThreadPriority.Lowest;
```

A more complex strategy could consider individual requirements as long as they are conformant with certain threshold values associated with the selected offer.

6.4 Setting up a WSB

The full functionality of a WSB is implemented as part of the WS-QoS API. Therefore, implementing an own offer broker service is fairly easy: The first step is to create a new Web service on the local server. Instead of extending the class *System.Web.Services.WebService* it should rather inherit the abstract class *WSQoSAPI.WSQoSOfferBrokerWebService* provided by the WS-QoS API. The name space for the new Web service should be set to http://wsqos.org/ in order to comply with the abstract service description of a WS-QoS offer broker.

In order to get the WSB working, two functions have to be overridden by the concrete instance. *GetConfigFilePath()* should return a string containing the path of a configuration file. Note that the path should be an absolute path (i.e. NOT ./myfile.config !), since the working directory changes for local and remote service invocation. The file should hold information on the location of the UDDI registry to be used as well as the business-keys for trusted service providers as shown in the following sample configuration file:

```
<?XML version="1.0" encoding="utf-8" ?>
<config>
  <UDDIRegistryUrl>http://UDDI.ibm.com/ubr/inquiryapi</UDDIRegistryUrl>
  <TrustedProviders>
    <trustedProvider name="My Service Provider"
      UDDIBusinessKey="f1a66f68-39ca-4e3e-9786-7c2e11f3ed26" />
  </TrustedProviders>
</config>
```

Note that several trusted provider entries are possible. Restricting the scope of providers is important when working with a public UDDI registry to avoid the automated selection of fraud service providers.

The function *CreateCurrencyConverter()* returns a concrete instance of a *WSQoSAPI.CurrencyConverter*. One can either implement an own currency converter by extending the *WSQoSAPI.CurrencyConverter* abstract class or use the *WebServiceXCurrencyConverter* provided by the package WS-QoS Util. This object uses a Web service from the service provider WebserviceX.NET to obtain current exchange rates. The package can be found in the WebServicesXUtil.dll, which is part of the WS-QoS Stock Quote demo downloadable from the WS-QoS homepage [45].

6.5 Creating WS-QoS compliant client applications

Three steps are necessary to implement a WS-QoS compliant client application: First, the client has to become aware of its own QoS requirements. This task is taken care of by the WS-QoS Requirement Manager object provided by the WS-

QoS API. Furthermore, clients need to request either a local or a remote instance of a WSB to discover the service offering the most appropriate QoS level. Upon service invocation, the client application is expected to send its current QoS requirements in a WS-QoS SOAP header along with each service request.

6.5.1 Defining client side QoS requirements

A convenient possibility to define client side QoS requirements at the implementation time is to apply WS-QoS custom attributes to the client's service proxy class a.k.a. the Web reference. However, this static declaration of QoS requirements in the program code does not permit any alteration of these specifications at runtime. Yet for some parameters such as a maximal acceptable price as means to dynamically adjust threshold values at run-time is desirable. Therefore, requirements in external WS-QoS files can be included with an *ImportQoSDeclaration* attribute. These files are then monitored and any changes to the files are integrated into the current client requirement objects held by the clients WS-QoS Requirement Manager object. Static and dynamic specifications can be combined.

6.5.2 Creating a proxy class

First of all, a service proxy class should be generated by defining a Web reference from the *tModel* the service is going to implement. In our example, the Web reference is called GenericService:

```
// proxy object for the WS-QoS compliant service to be used
private GenericService.WSQoSAwareService MyService =
       new GenericService.WSQoSAwareService();
```

Since the QoS proxy introduced in Section 5.7 is used to perform packet marking based on the transport QoS level specified in the WS-QoS SOAP header, the QoS proxy is to be registered with the service proxy in the client application's constructor:

```
this.MyService.Proxy = new WebProxy("localhost", 7777);
```

6.5.3 Declaring custom attributes on the proxy class

WS-QoS elements can be declared with static custom attributes assigned to a service proxy class generated from a WSDL file. Editing of a generated proxy class is not recommended because changes will be lost in the case of updating. Elements for specific operations can be assigned to the corresponding Web method, as the following code samples shows.

```
[ServerQoSMetrics(ProcessingTime=0.5, Availability=0.98)]
[TransportQoSPriorities(Delay=8, PacketLoss=5)]
[ImportQoSDeclaration(Location="./maxprice.wsqos")]
public class MyWSQoSAwareService :
System.Web.Services.Protocols.SoapHttpClientProtocol {

      public WSQoSSoapHeader WSQoSSoapHeaderValue;
      public WSQoSAwareStockQuoteService() { ... }
[TransportQoSPriorities(Delay=5, Jitter=5)]
[CustomServerQoSMetric(Name="CPU_Share",
Ontology="http://mydomain.com/myontology.wsqos", Value="0.2")]
[System.Web.Services.Protocols.SoapHeaderAttribute( ... )]
[System.Web.Services.Protocols.SoapDocumentMethodAttribute( ... )]
      public foo MyOperation() {  ... }
}
```

Figure 43 gives an overview of the classes of the WS-QoS API representing elements of the WS-QoS XML schema. While the application of a (single) *Price* attribute is restricted to the scope of the whole service, the following custom attributes can be applied either directly to the service proxy class or on specific methods:

- ServerQoSMetrics and CustomServerQoSMetric,
- TransportQoSPriorities and CustomTransportQoSPriority,
- SecurityAndTransaction and ProtocolSupport,
- ContractAndMonitoring and ThirdParty

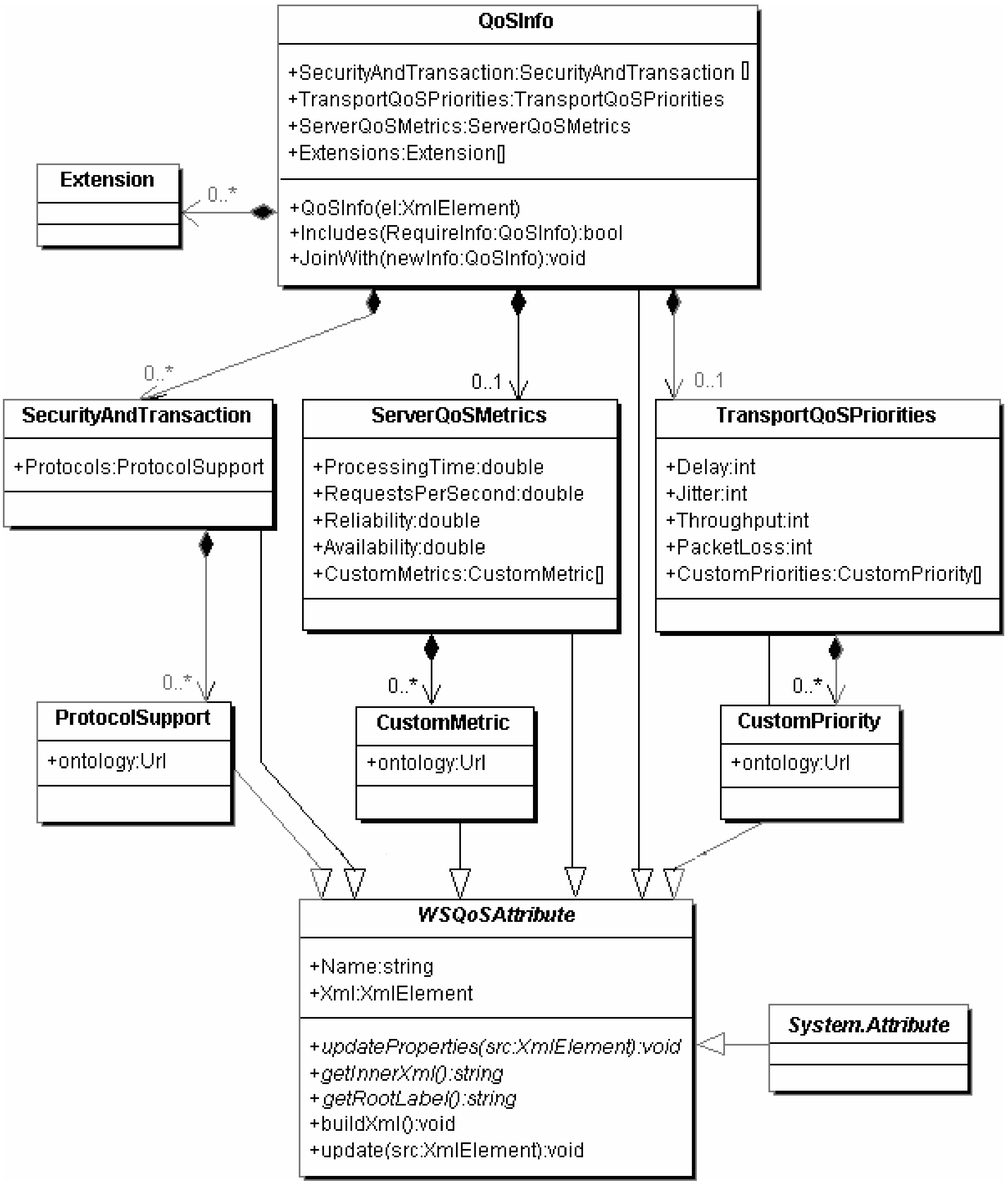


Figure 43. Classes of the WS-QoS API representing WS-QoS elements

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. WS-QoS custom attributes declared on the proxy class will be added to the *DefaultQoSInfo* element of the requirements object. If WS-QoS custom attributes are applied to a specific Web method within the proxy class, they will be added to an *OperationQoSInfo* element for this operation. Apart from holding the values for certain WS-QoS elements, these objects implement the functionality of whether one object provides an equal or better QoS level. Their function *Includes()*

is invoked by the WSB when testing an offer against a client requirement definition.

ImportQoSDeclaration attributes can be defined in order to reference XML WS-QoS files containing further requirements, which can be changed dynamically at runtime. When an import attribute is initiated, the import file is read and a corresponding QoS definition object is built by using the attributes' getInnerXML() method. Then a thread is initiated to perform regular checks on whether the import 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. The following code sample shows how static and dynamic WS-QoS declarations are combined (the file name of this code is *maxprice.wsqos*.

```
<?XML version="1.0" encoding="utf-8"?>
<wsqos XMLns="http://www.wsqos.net/schemas">
  <definition>
    <requirements name="Dynamic maximum price for my client application">
      <price currency="EUR">0.49</price>
    </requirements>
  </definition>
```

One can include this file by applying an *ImportQoSDeclaration* attribute to the service proxy class *MyWSQoSAwareService*. We also declare a selection of static QoS requirements on the class and one sample method:

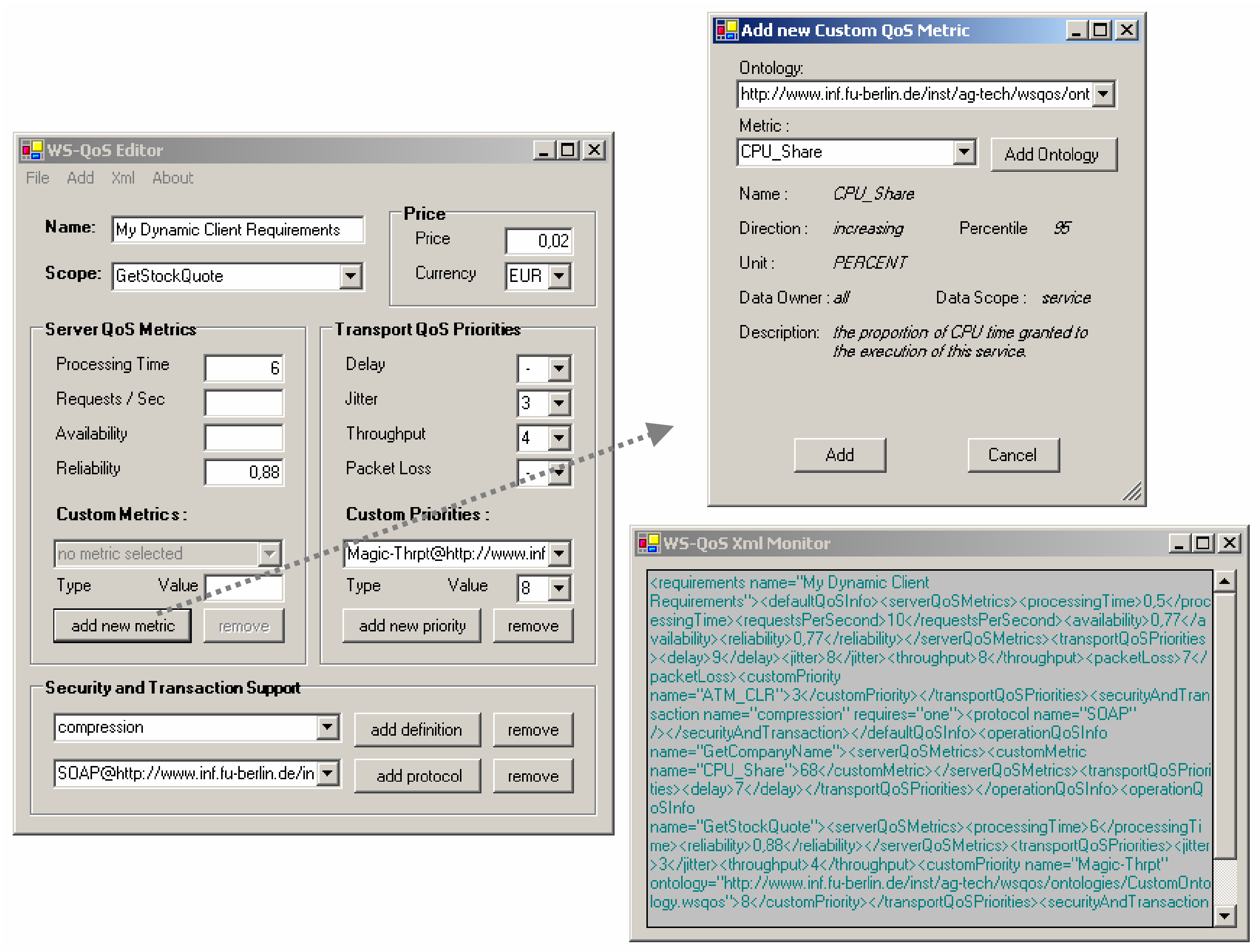
```
[ServerQoSMetrics(ProcessingTime=0.5, Availability=0.98)]
[TransportQoSPriorities(Delay=8, PacketLoss=5)]
[ImportQoSDeclaration(Location="./maxprice.wsqos")]
public class MyWSQoSAwareService :
System.Web.Services.Protocols.SoapHttpClientProtocol {


      public WSQoSSoapHeader WSQoSSoapHeaderValue;
      public WSQoSAwareStockQuoteService() { ... }
[TransportQoSPriorities(Delay=5, Jitter=5)]
[CustomServerQoSMetric(Name="CPU_Share",
Ontology="http://mydomain.com/myontology.wsqos", Value="0.2")]
[System.Web.Services.Protocols.SoapHeaderAttribute( ... )]
[System.Web.Services.Protocols.SoapDocumentMethodAttribute( ... )]
      public foo MyOperation() {  ... }
}
```

The WS-QoS requirement manager will now generate an equivalent WS-QoS XML as follows:

```
<?XML version="1.0" encoding="utf-8"?>
<wsqos XMLns="http://www.wsqos.net/schemas">
  <definition>
    <requirements name="My Requirements for StockQuote Client">
      <defaultQoSInfo>
        <serverQoSMetrics>
          <processingTime>0.5</processingTime>
          <availability>0.98</availability>
        </serverQoSMetrics>
        <transportQoSPriorities>
          <delay>8</delay>
          <packetLoss>5</packetLoss>
        </transportQoSPriorities>
      </defaultQoSInfo>
      <operationQoSInfo name="MyOperation">
        <serverQoSMetrics>
          <customMetric name="CPU_Share"
  ontology="http://mydomain.com/myontology.wsqos">0.2</customMetric>
        </serverQoSMetrics>
        <transportQoSPriorities>
          <delay>5</delay>
          <jitter>5</jitter>
        </transportQoSPriorities>
      </operationQoSInfo>
      <price currency="EUR">0.49</price>
    </requirements>
  </definition>
</wsqos>
```

6.5.4 Adjusting WS-QoS files with the WS-QoS editor

If one decides to use WS-QoS XML files to specify dynamic requirements there is no need to directly edit the XML code. The WS-QoS editor provides a comfortable GUI to edit WS-QoS specifications. Using the editor also ensures that the XML code is compliant with the WS-QoS XML schema. Figure 44 shows how a custom metric (CPU_Share in this case) is added to the Operation QoS info element for an operation *GetStockQuote()*.


WS-QoS Editor

File Add Xml About

Name: My Dynamic Client Requirements

Scope: GetStockQuote

Price: Price 0,02 · Currency EUR

Server QoS Metrics: Processing Time 6 · Requests / Sec · Availability · Reliability 0,88

Custom Metrics: no metric selected · Type · Value · add new metric · remove

Transport QoS Priorities: Delay - · Jitter 3 · Throughput 4 · Packet Loss -

Custom Priorities: Magic-Thrpt@http://www.inf · Type · Value 8 · add new priority · remove

Security and Transaction Support: compression · add definition · remove · SOAP@http://www.inf.fu-berlin.de/in · add protocol · remove

Add new Custom QoS Metric

Ontology: http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ont

Metric : CPU_Share · Add Ontology

Name : *CPU_Share*

Direction : *increasing* · Percentile *95*

Unit : *PERCENT*

Data Owner : *all* · Data Scope : *service*

Description: *the proportion of CPU time granted to the execution of this service.*

Add · Cancel

WS-QoS Xml Monitor

```
<requirements name="My Dynamic Client
Requirements"><defaultQoSInfo><serverQoSMetrics><processingTime>0,5</proc
essingTime><requestsPerSecond>10</requestsPerSecond><availability>0,77</a
vailability><reliability>0,77</reliability></serverQoSMetrics><transportQoSPriorities
><delay>9</delay><jitter>8</jitter><throughput>8</throughput><packetLoss>7</
packetLoss><customPriority
name="ATM_CLR">3</customPriority></transportQoSPriorities><securityAndTran
saction name="compression" requires="one"><protocol name="SOAP"
/></securityAndTransaction></defaultQoSInfo><operationQoSInfo
name="GetCompanyName"><serverQoSMetrics><customMetric
name="CPU_Share">68</customMetric></serverQoSMetrics><transportQoSPriori
ties><delay>7</delay></transportQoSPriorities></operationQoSInfo><operationQ
oSInfo
name="GetStockQuote"><serverQoSMetrics><processingTime>6</processingTi
me><reliability>0,88</reliability></serverQoSMetrics><transportQoSPriorities><jitter
>3</jitter><throughput>4</throughput><customPriority name="Magic-Thrpt"
ontology="http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ontologies/CustomOnto
logy.wsqos">8</customPriority></transportQoSPriorities><securityAndTransaction
```

Figure 44. Adding a custom server metric declaration with the WS-QoS Editor

6.5.5 The WS-QoS requirement manager

The WS-QoS Requirement Manager is used by the client application before the service selection and service invocation take place. Therefore, the following declaration should be made in the client application class:

```
// WS-QoS requirement manager used to survey current client requirements
private WSQoSAPI.WSQoSRequirementManager MyRequirementManager;
```

To get the Requirement Manager working, it has to be instantiated when the client application's constructor is called. A reference to the service proxy should be passed on as a parameter. Therefore, the following code has to be added to the client application's constructor:

```
// create requirement manager
this.MyRequirementManager = new
WSQoSAPI.WSQoSRequirementManager(this.MyService);
```

Upon instantiation, the Requirement Manager collects WS-QoS custom attributes that have been assigned to the proxy class and builds a WS-QoS requirement definition object. It then collects all *ImportQoSDeclaration* attributes, builds WS-QoS definition objects from the specified XML and sets their parent property to receive update messages in the case that a file has been changed. Finally, the newly built WS-QoS definition objects are added to those retrieved from the static attributes. From now on, current client QoS requirements can be retrieved from the WS-QoS Requirement Manager, e.g. for service selection, by requesting its *CurrentRequirements* property.

If desired, the current client QoS requirements can be recorded in a log file by calling the *StartLogging()* method on the Requirement Manager object. To start the logging of current requirements, the following code has to be added to the client application's constructor:

```
this.MyRequirementManager.StartLogging();
```

6.6 QoS-aware service selection with the WSB

The WSB holds up-to-date information on offers currently available for a group of services, which have been requested in recent time. The WSB categorizes offers by their interface that the services providing them implement. To keep an up-to-date list of all services implementing a given interface, the WSB consults one or more UDDI registries regularly. The WSB checks the availability of services regularly. Once an offer expires, the WSB deletes it from its registry. If the validity of the offer is extended, it will be re-detected during the next check.

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

6.6.1 Instantiation of the WSB

There are two implementations of the WSB. One is a local object running within the application. This ensures a highly performing service selection and detailed information on available offers, as needed for the WS-QoS Monitor described below. 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 could use a single private WSB running as a Web service within their network domain. This WSB could well be used by any other implementation of WS-QoS. Using the generic interface *WSQoSAPI.WSQoSOfferBroker* one can leave the choice of the implementation for later or even change it at runtime. The following code sample shows how the WSB should be declared in the client application class:

```
// UDDI registry key of the tModel the service implements
private string MyUDDITModelKey = "uuid: ... ";
// WS-QoS service broker used for QoS aware dynamic service discovery
private WSQoSAPI.WSQoSOfferBroker MyOfferBroker;
// best offer currently available
private WSQoSAPI.QoSOffer CurrentOffer = null;
```

Instead of specifying the tModel key directly in the code, a configuration file could be used to adjust to changes. Now, the WSB object has to be instantiated in the constructor of the client application, which is the easiest way to use a remote Web service broker. In this case one has to create a new *WSQoSAPI.RemoteOfferBroker* object:

```
// create offer broker
this.MyOfferBroker = new WSQoSAPI.RemoteOfferBroker();
```

The remote WSB object expects to load a configuration file named RemoteBroker.config in the application's working directory. This file is used to specify the URL of the WSB to be used. The file should show the following XML:

```
<?XML version="1.0" encoding="utf-8" ?>
<config>

      <BrokerServiceUrl>http://mydomain.com/MyWSQoSOfferBroker.asmx</Brok
      erServiceUrl>
</config>
```

Some applications, such as the WS-QoS Monitor introduced later, might prefer a local instance of the WSB in order to avoid the time consuming contacting of yet another Web service for offer selection or to retrieve more detailed information on available offers. Yet, this also means that all service providers and offers have to be watched while using a single WSB helps to restrict network traffic. The local WSB object makes use of the predefined static field *WSQoSAPI.CurrencyConverter.ActiveCurrencyConverter*. This abstract currency converter has to be initiated with an instance of a concrete converter to allow for the comparison of prices given in different currencies. One can either implement an own version or use the one provided with the *WebServicesXUtil.dll*. The following code shows how a currency converter is instantiated in the client application:

```
// create new currency converter
if (WSQoSAPI.CurrencyConverter.ActiveCurrencyConverter == null)
{
   WSQoSAPI.CurrencyConverter.ActiveCurrencyConverter =
new WebServicesXUtil.WebServiceXCurrencyConverter();
}
```

The local WSB object expects to load a configuration file named *LocalBroker.config* in the application's working directory. This file is used to specify the URL of the UDDI registry to be consulted as well as a list of trusted service providers whose services are considered for service selection. This way, automated selection of fraud services can be avoided. The configuration file should show the following XML:

```
<?XML version="1.0" encoding="utf-8" ?>
<config>
      <UDDIRegistryUrl>http://mydomain.com/UDDIpublic/inquire.asmx</UDDIR
egistryUrl>
  <TrustedProviders>
    <trustedProvider name="Provider A" UDDIBusinessKey=" ... " />
    <trustedProvider name="Provider B" UDDIBusinessKey=" ... " />
  </TrustedProviders>
</config>
```

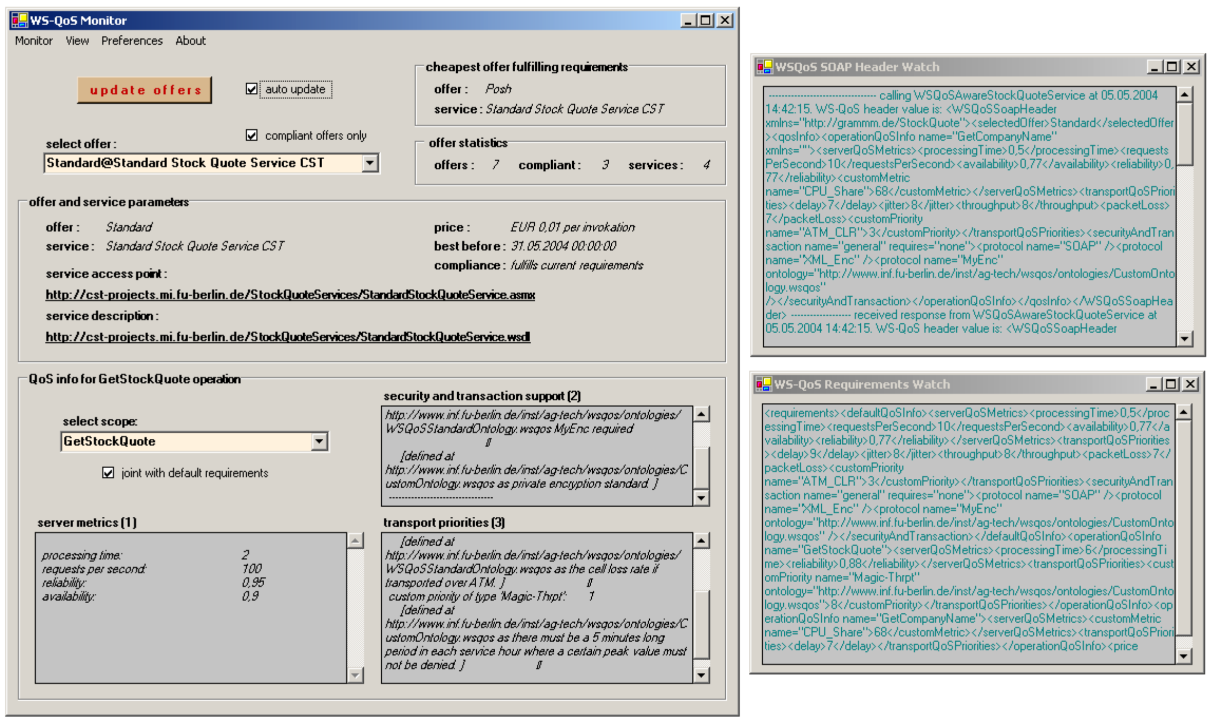
6.6.2 Updating the most appropriate WS-QoS offer

To update the most appropriate WS-QoS offer, a separate thread will be started to consult the WSB at intervals. The client's current QoS requirements and the *tModel* of the desired service type are passed to the WSB as parameters. The following lines of code show how the service access point of the service proxy is updated:

```
public void UpdateOffer()
{
      this.CurrentOffer = this.CurrentOfferBroker.GetBestOffer(
this.MyRequirementManager.CurrentRequirements,
this.MyUDDITModelKey);
// set new service access point
this.MyService.Url = this.CurrentOffer.Service.AccessPointUrl;
}
```

6.6.3 Comparing available offers with the WS-QoS monitor

Whenever one automates important decisions, the need to monitor the correctness of the *automation* arises. Therefore, we have developed the WS-QoS Monitor, which monitors and shows all available offers and the current client requirements. This way it makes possible to check on the compliance of offers. If no appropriate offer can be found, the overview of possible offers will help a user to see what requirements might be inappropriate and he or she could make the adjustments needed in order to make their application work again. Moreover, detailed information on services can be retrieved by following the links to the services' access points and description documents.



[✓] auto update

[✓] compliant offers only

select offer: Standard@Standard Stock Quote Service CST

cheapest offer fulfilling requirements
offer: Posh
service: Standard Stock Quote Service CST

offer statistics
offers: 7 compliant: 3 services: 4

offer and service parameters
offer: Standard
service: Standard Stock Quote Service CST
price: EUR 0,01 per invokation
best before: 31.05.2004 00:00:00
compliance: fulfills current requirements
service access point:
http://cst-projects.mi.fu-berlin.de/StockQuoteServices/StandardStockQuoteService.asmx
service description:
http://cst-projects.mi.fu-berlin.de/StockQuoteServices/StandardStockQuoteService.wsdl

QoS info for GetStockQuote operation

select scope: GetStockQuote

[✓] joint with default requirements

security and transaction support (2)

```
http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ontologies/
WSQoSStandardOntology.wsqos MyEnc required
                    //
    [defined at
http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ontologies/C
ustomOntology.wsqos as private encryption standard ]
 ------------------------------
```

server metrics (1)

```
processing time:                2
requests per second:            100
reliability:                    0,95
availability:                   0,9
```

transport priorities (3)

```
    [defined at
http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ontologies/
WSQoSStandardOntology.wsqos as the cell loss rate if
transported over ATM. ]                    //
 custom priority of type 'Magic-Thrpt':     1
    [defined at
http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ontologies/C
ustomOntology.wsqos as there must be a 5 minutes long
period in each service hour where a certain peak value must
not be denied. ]                  //
```

WSQoS SOAP Header Watch

```
-------------------------------- calling WSQoSAwareStockQuoteService at 05.05.2004
14:42:15. WS-QoS header value is: <WSQoSSoapHeader
xmlns="http://grammm.de/StockQuote"><selectedOffer>Standard</selectedOffer
><qosInfo><operationQoSInfo name="GetCompanyName"
xmlns=""><serverQoSMetrics><processingTime>0,5</processingTime><requests
PerSecond>10</requestsPerSecond><availability>0,77</availability><reliability>0,
77</reliability><customMetric
name="CPU_Share">68</customMetric></serverQoSMetrics><transportQoSPriori
ties><delay>7</delay><jitter>8</jitter><throughput>8</throughput><packetLoss>
7</packetLoss><customPriority
name="ATM_CLR">3</customPriority></transportQoSPriorities><securityAndTran
saction name="general" requires="none"><protocol name="SOAP" /><protocol
name="XML_Enc" /><protocol name="MyEnc"
ontology="http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ontologies/CustomOnto
logy.wsqos"
/></securityAndTransaction></operationQoSInfo></qosInfo></WSQoSSoapHea
der> ------------------ received response from WSQoSAwareStockQuoteService at
05.05.2004 14:42:15. WS-QoS header value is: <WSQoSSoapHeader
```

WS-QoS Requirements Watch

```
<requirements><defaultQoSInfo><serverQoSMetrics><processingTime>0,5</proc
essingTime><requestsPerSecond>10</requestsPerSecond><availability>0,77</a
vailability><reliability>0,77</reliability></serverQoSMetrics><transportQoSPriorities
><delay>9</delay><jitter>8</jitter><throughput>8</throughput><packetLoss>7</
packetLoss><customPriority
name="ATM_CLR">3</customPriority></transportQoSPriorities><securityAndTran
saction name="general" requires="none"><protocol name="SOAP" /><protocol
name="XML_Enc" /><protocol name="MyEnc"
ontology="http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ontologies/CustomOnto
logy.wsqos" /></securityAndTransaction></defaultQoSInfo><operationQoSInfo
name="GetStockQuote"><serverQoSMetrics><processingTime>6</processingTi
me><reliability>0,88</reliability></serverQoSMetrics><transportQoSPriorities><cust
omPriority name="Magic-Thrpt"
ontology="http://www.inf.fu-berlin.de/inst/ag-tech/wsqos/ontologies/CustomOnto
logy.wsqos">8</customPriority></transportQoSPriorities></operationQoSInfo><op
erationQoSInfo name="GetCompanyName"><serverQoSMetrics><customMetric
name="CPU_Share">68</customMetric></serverQoSMetrics><transportQoSPriori
ties><delay>7</delay></transportQoSPriorities></operationQoSInfo><price
```

Figure 45. An instance of the WS-QoS Monitor surveying the service selection for a WS-QoS compliant client

The WS-QoS Monitor, shown in Figure 45, allows users to register both a file containing current client requirements as well as a file containing a log of all WS-QoS SOAP headers exchanged between the client application and the service. One can monitor these files in the separate windows: Requirement Watch and WS-QoS SOAP Header Watch. If the files are changed, the corresponding views will be updated. Moreover, once requirements are registered, the detailed view of an offer informs the user whether the offer currently viewed is compliant with the client requirements or not.

The WS-QoS Monitor uses a local offer broker to show detailed information on available offers. Therefore, the configuration files provided with the program should be adjusted accordingly. If one uses a remote WSB, the monitor will request the URL of the UDDI registry used directly from the remote WSB to simulate its behavior.

6.7 Service invocation

Before calling an operation such as *MyOperation*, the specific QoS requirements for this operation have to be retrieved. The *GetJointOperationQoSInfo()* function is invoked on the *CurrentRequirements* property of the Requirement Manager. The function's name is passed on as a parameter. This way, we receive a joint definition of default values extended and overridden by definitions made in specific *operationQoSInfo*. The following code shows the joint requirements for the operation *MyOperation*.

```
<operationQoSInfo name="MyOperation">
   <serverQoSMetrics>
     <processingTime>0.5</processingTime>
     <availability>0.98</availability>
     <customMetric name="CPU_Share"
ontology="http://mydomain.com/myontology.wsqos">0.2</customMetric>
   </serverQoSMetrics>
   <transportQoSPriorities>
     <delay>5</delay>
     <packetLoss>5</packetLoss>
     <jitter>5</jitter>
     </transportQoSPriorities>
 </defaultQoSInfo>
```

The XML representation of this current QoS info object is placed in the WS-QoS SOAP header, along with the name of the WS-QoS offer currently selected. The SOAP header is finally associated with the service proxy through which the service invocation is initiated. The following code demonstrates the necessary steps to prepare a WS-QoS compliant service invocation:

```
// build wsqos header
this.MyWSQoSHeader.qosInfo = (XMLElement)this.MyRequirementManager.
CurrentRequirements.GetJointOperationQoSInfo("MyOperation").XML;
this.MyWSQoSHeader.selectedOffer = this.CurrentOffer.Name;
this.MyService.WSQoSSoapHeaderValue = this.MyWSQoSHeader;
// invoke service operation
foo f = this.MyService.MyOperation();
```

6.7.1 Logging of WS-QoS SOAP headers

WS-QoS SOAP headers communicated between client and server can be logged by declaring a *WSQoSAPI.WSQoSHeaderLogExtension* on the operation in question in the service proxy class as the following code sample shows. Note that after adding attributes to the proxy class an automated update of the Web reference would destroy the changes.

```
public class MyWSQoSAwareService :
System.Web.Services.Protocols.SoapHttpClientProtocol {


      public WSQoSSoapHeader WSQoSSoapHeaderValue;
      public WSQoSAwareStockQuoteService() { ... }
      [WSQoSAPI.WSQoSHeaderLogExtension(ServiceName="MyWSQoSAwareService"
)]
      [System.Web.Services.Protocols.SoapHeaderAttribute("WSQoSSoapHeader
Value",
Direction=System.Web.Services.Protocols.SoapHeaderDirection.InOut,
Required=false)]
[System.Web.Services.Protocols.SoapDocumentMethodAttribute( ... )]
      public foo MyOperation() {
            object[] results = this.Invoke("MyOperation", new object[]
{Symbol});
            return ((foo)(results[0]));
      }
}
```

To get the SOAP extension running, it has to be registered in the client application's *web.config* file as the following code extract shows:

```
<?XML version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
     <webServices>
      <soapExtensionReflectorTypes>
          <add type="WSQoSAPI.WSQoSReflector, WSQoSAPI"/>
      </soapExtensionReflectorTypes>
     </webServices>
     ...
  </system.web>
</configuration>
```

The client application's log file can be registered with the WS-QoS Monitor to be viewed in a separate window. The content of this WS-QoS SOAP Header Watch view is updated each time the client initiates a further service invocation. And by monitoring WS-QoS SOAP headers being exchanged the user will know that she has succeeded in building a WS-QoS compliant Web services project.

6.8 Conclusion

In this chapter, we described how to apply the implementation of our WS-QoS framework. Implementing a WS-QoS aware Web service application requires mainly three steps. First, the service should implement a generic service interface (e.g. *tModel*), which already defines the use of an optional WS-QoS SOAP header to transport QoS information within service requests and responses. Second, the service has to implement a strategy to provide WS-QoS offers, which should be adjustable to changing situations of service utilization. Finally, to achieve the QoS level(s) associated with distinct offers, the selected offer and further QoS requirements have to be evaluated when receiving a request. Apart from developing new services one can also qualify existing service implementations as WS-QoS compliant by making just a few alterations to the code.

In the next chapter, we demonstrate our performance measurements of to Web service performance in general and the performance of our implementation. Furthermore, we show the advantages for both clients and servers when the WS-QoS framework is applied.