Patterns for Measuring Performance-Related QoS Properties in Distributed Systems

Ernst Oberortner
Distributed Systems Group, Information Systems Institute
Vienna University of Technology, Austria
e.oberortner@infosys.tuwien.ac.at

Uwe Zdun
Software Architecture Group, Institute for Distributed and Multimedia Systems
University of Vienna, Austria
uwe.zdun@univie.ac.at

Schahram Dustdar
Distributed Systems Group, Information Systems Institute
Vienna University of Technology, Austria
dustdar@infosys.tuwien.ac.at

In distributed systems, clients can access a server's objects via a network. Service Level Agreements (SLA) can exist, which specify – among other things – performance-related Quality of Service (QoS) properties between the client and the server, such as round-trip time, processing time, or availability. For a provider, i.e., the server's host, serious financial consequences or other penalties can follow in case of not fulfilling the SLAs. The consumer, i.e., the client, wants to evaluate that the provider complies with the guaranteed SLAs. Designing and developing a QoS-aware distributed system means facing many design challenges, such as where and how to measure the performance-related QoS properties. This paper presents design practices and patterns for measuring such QoS properties by extending existing patterns. The pattern's implementations are exemplified in a web service-oriented distributed system. The focus of the pattern lies on the QoS measuring impact on the client's or server's performance, the extend of separation of concerns, the property of reusability, the preciseness of the measured QoS properties, and the vulnerability to forgery by the server or the client. This paper's patterns should help software architects and developers in building an efficient solution for measuring performance-related QoS properties in a distributed system.

1 Introduction

In a distributed system, service level agreements between service provider and consumer can exist. An SLA is a contract that contains – among other things – agreements on performance-related properties when the consumer accesses the providers services over a network. SLAs assure that the consumers get the service they paid for and that the service provider fulfils the SLA guarantees. If the server provider can not meet the goals, then serious financial consequences or other penalties can follow. Service providers need to know what they can promise within the SLAs and what their IT infrastructure can deliver. On the other hand, the consumer wants to observe and validate that the server provider does not violate the guaranteed SLAs [19, 11].

In this work we concentrate on SLA clauses that contain mainly performance-related Quality of Service (QoS) measurements for validations. During the development of an appropriate QoS monitoring infrastructure with the distributed system, many challenging design problems have to be faced. For example, where to measure the SLA's QoS properties, because they can be measured in various layers on the client-side or the server-side, as well as in intermediary components of the network communication. We concentrate on the following design challenges of measuring performance-related QoS properties: (1) Does the QoS measuring impact the system's performance? (2) Does the QoS measuring provide precise QoS measurements? (3) Does the QoS measuring provide a good separation of concerns so that it does not modify the client's or server's implementation? (4) Can the QoS measurements be forged in case they have to be reported? and (5) Is the QoS measuring solution reusable for new clients or servers?

We document design practices and patterns of measuring performance-related QoS properties, such as round-trip time, network latency, or processing time [12, 17, 16, 21]. The paper evaluates the presented patterns against the challenging design problems and gives advice in the decision making for building QoS-aware distributed systems. The background of this work are established patterns, presented in the Gang of Four (GoF) book [6], the Pattern-Oriented Software Architecture (POSA) series [5, 18, 4], and the Remoting Patterns book [22]. The patterns described herein are meant for software architects and developers who have to design and develop distributed systems and decide how to measure performance-related QoS properties within the distributed system.

The presented patterns do not take into account how to store or evaluate the measurements. The main focus of the patterns lies on distributed systems where clients invoke the server's remote objects in an RPC request/response style. The patterns can be applied in synchronous invocations and have to be slightly modified to be usable in asynchronous invocations. Patterns for message-oriented distributed systems, such as presented in the Enterprise Integration Patterns book [7], can be used and extended to implement the QoS-aware communication between the clients and the remote objects. Although, measuring QoS in multicast distributed systems is outside the scope of this work.

This paper is organized as follows: The next section, Section 2, explains the relevant existing patterns and performance-related QoS properties which build the basis of the presented patterns within this paper. In Section 3 we present the patterns for measuring the performance-related QoS properteis. To get a better understanding of the patterns, Section 4 exemplifies their implementations in a web service-oriented distributed system. Next, Section 5 summarizes the main findings of the patterns with respect to the design challenges. We conclude with Section 6.

2 Background

This section gives some needed background information on basic patterns in distributed systems that build the basis of the presented patterns within this paper. First, the basic patterns and their relationships are described. The second part of this section explains the background on performance-related QoS properties used in this paper's patterns.

2.1 Basic Patterns in Distributed Systems

In Figure 1 we illustrate the typical activities within a distributed system when a client invokes some server's remote object. Mostly, a middleware manages the communication between the client and the server, hiding the heterogenity of the underlying platforms and providing transparency of the distributed communications. The middleware can access the network services offered by the operating system for accessing and transmitting requests to the server's remote objects over the network [22].

For accessing the client's middleware, the REQUESTOR pattern can be used [22]. The REQUESTOR invokes the remote object's operation using the underlying middleware. Also, the client's application can access the middleware following the CLIENT PROXY pattern [22] to provide a good separation of concerns and to attach additional information to the client's requests. The CLIENT PROXY invokes the

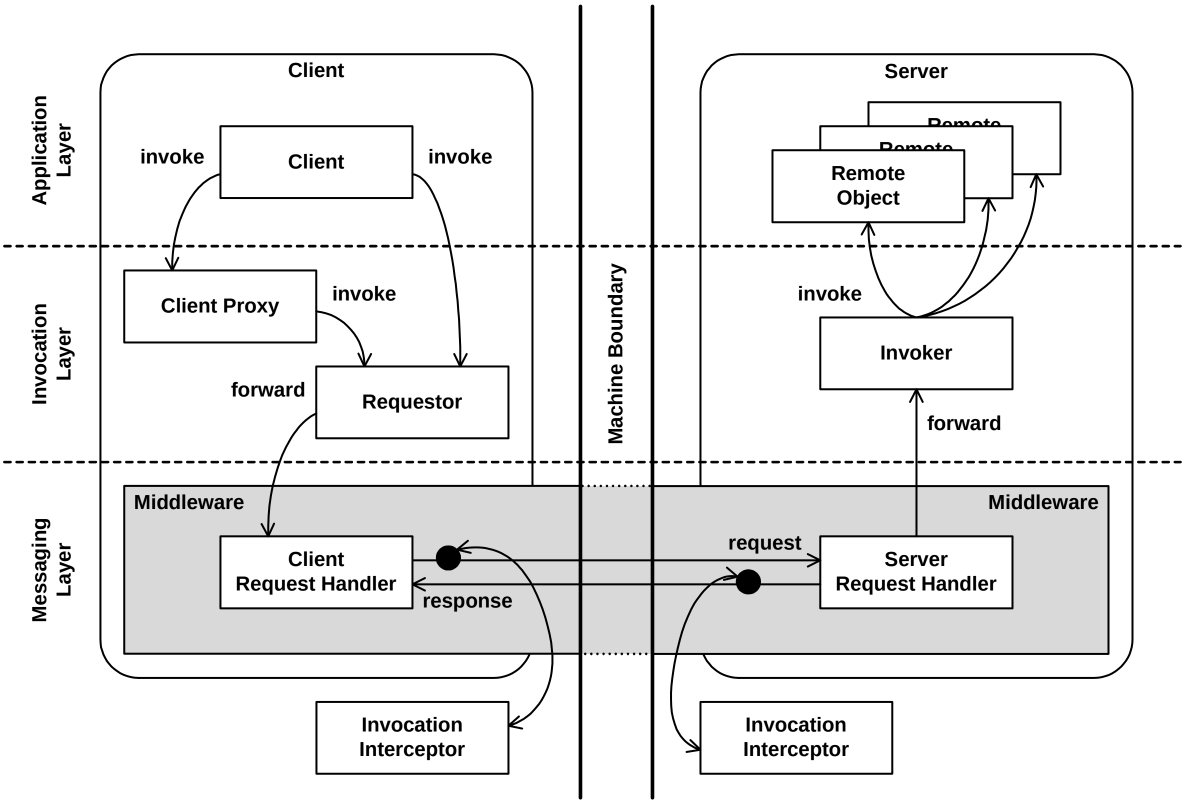


Figure 1: An overview of existing patterns in distributed systems

middleware using the REQUESTOR pattern. The implementation of the client's middleware can follow the CLIENT REQUEST HANDLER pattern [22], to send the requests over the network to the server and to handle the server's response.

The implementation of the server's middleware can follow the SERVER REQUEST HANDLER pattern [22]. A SERVER REQUEST HANDLER receives the incoming requests, performs additional processing, and forwards the requests to the INVOKER of the remote objects. The INVOKER [22] receives the requests from the SERVER REQUEST HANDLER, can perform additional processing again, and dispatches the request to the corresponding remote object. After the remote object processed the incoming request it sends the response back to the INVOKER, which performs some additional processing, and forwards the response to the SERVER REQUEST HANDLER. The SERVER REQUEST HANDLER can perform again some additional processing and forwards the response to the requestor.

The INVOCATION INTERCEPTOR pattern [22], which is based on the INTERCEPTOR pattern [18], provides hooks in the invocation path to perform additionally required actions, such as logging or securing the invocation data. Mostly, the client's or server's middleware provides functionalities for placing INVOCATION INTERCEPTORS into the invocation path. Hence, an INVOCATION INTERCEPTOR can process and manipulate the available invocation data, which depends on the INVOCATION INTERCEPTOR's place in the invocation path. The middleware can provide the feature of attaching and changing an INVOCATION INTERCEPTOR dynamically during the runtime of the system, such as by using an API or configuration files. As a consequence, the INVOCATION INTERCEPTOR implies a higher complexity of the middleware's implementation. An INVOCATION INTERCEPTOR can attach the context-specific information to the INVOCATION CONTEXT [22] of the invocation data. In this paper, we assume the usage of the INVOCATION CONTEXT pattern for storing the performance-related QoS measurements during remote object invocations.

Client and server interactions can take place within a local area network (LAN) or over a wide area network (WAN), such as the Internet. If a client wants to invoke a remote object that is not located in the same LAN, the client request must be sent over a WAN to the corresponding remote object's LAN. In this case, inside the LAN a proxy server can be used, whose implementation follows the well-known PROXY pattern. Client and server can make use of the different PROXY patterns, such as such as CLIENT PROXY, VIRTUAL PROXY, and FIREWALL PROXY [4].

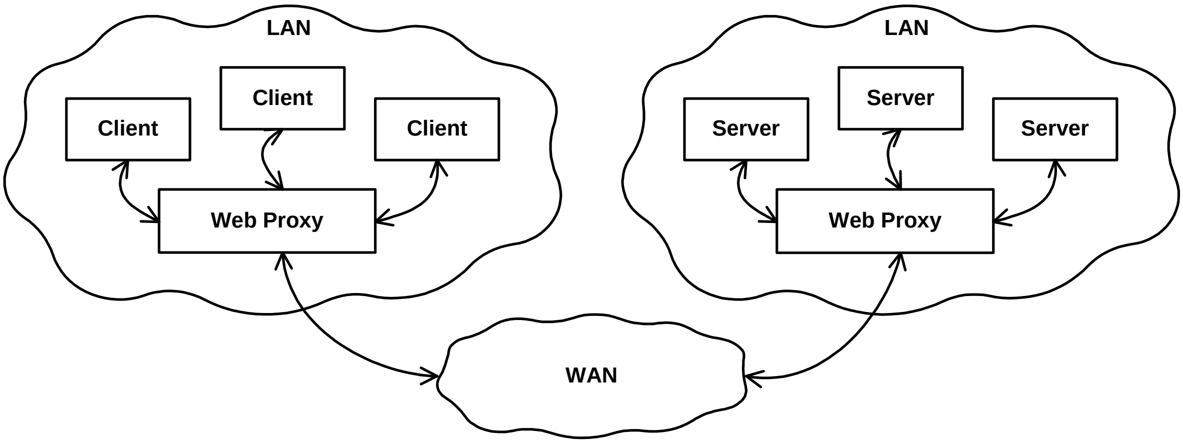


Figure 2: Using the WEB PROXY pattern

Figure 2 illustrates the usage of the PROXY pattern for implementing a web proxy. In this scenario, every component – client, server, and web proxy – features some middleware that manages the network access. For accessing a remote object over a WAN, the client-side WEB PROXY receives the requests from the clients within the LAN. It applies additional processing to the client's request, marshals it, and sends it into the WAN. A server-side WEB PROXY receives requests over a WAN, unmarshals them, applies additional processing, and forwards it to the appropriate remote object in the same LAN. After the remote object's processing, the server-side WEB PROXY receives the response, marshals it, applies additional processing, and sends it back to the client-side requestor. The client-side WEB PROXY receives the server-side response, applies additional processing and forwards the response to the appropriate client.

2.2 Performance-Related QoS Properties

Figure 3 shows some basics of some existing remoting middlewares, such as in web services frameworks like Apache CXF[1] or Apache Axis2[2]. The invocation data, between the client and the server, flows in the middleware through so-called chains, following the above described patterns. Client and server have both an incoming and an outgoing chain – `IN Chain` and `OUT Chain` in Figure 3 – which are responsible for processing the incoming requests and outgoing responses, respectively. Chains consist of multiple phases, making it possible to specify precisely where to hook INVOCATION INTERCEPTORS into the invocation path.

Many performance-related QoS properties have been reported in the literature that can be measured and monitored in a distributed system [12, 17, 16, 21]. This paper focuses on the following performance-related QoS properties:

- The *Round-Trip Time* is a client-side QoS property and it measures the elapsed time between the sending of the client's request and receiving the server's response.
- The *Marshaling Time* can be measured on the client and server-side. It is a measurement of the elapsed time for serializing the invocation data into the transport format of the underlying network.

[1] `http://cxf.apache.org/`
[2] `http://ws.apache.org/axis2/`

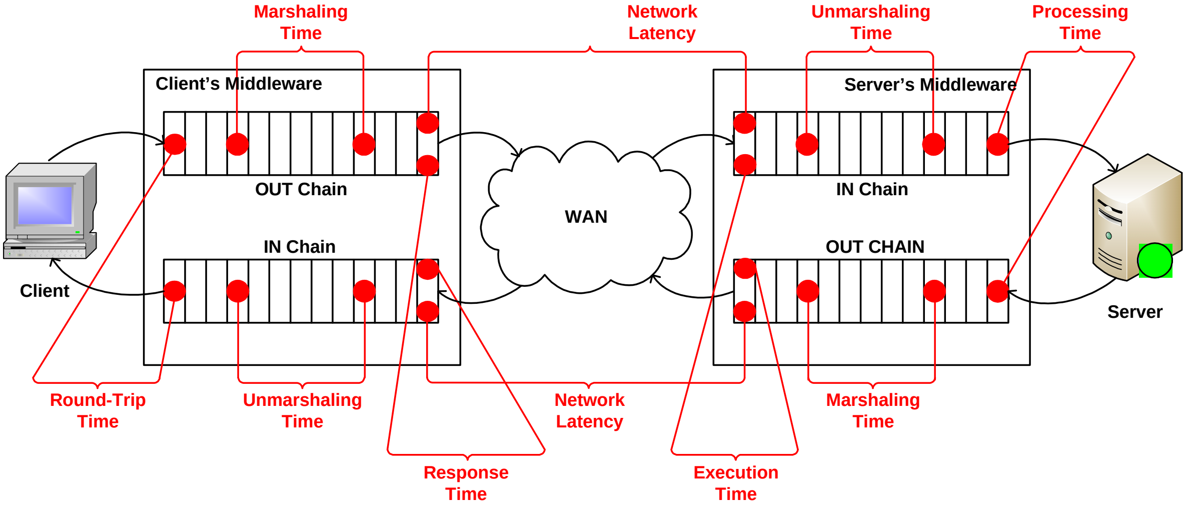


Figure 3: Measuring points of performance-related QoS concerns

- The required time for transmitting the marshaled invocation data over the network is called the *Network Latency*. It requires measuring points on the client- and the server-side. The *Network Latency* can be measured during the transmission of the client's request and during the transmission of the server's response.

- The *Response Time* is a client-side QoS property and measures the elapsed time between transmitting the serialized invocation data to the server and the reception of the server's response.

- On the server-side the *Processing Time* is the elapsed time for processing an incoming request. It does not take the marshaling and unmarshaling time into account.

- The *Unmarshaling Time* can be measured on the client-side and server-side. On the server-side it measures the elapsed time of de-serializing the incoming invocation data of the client's request to be compatible to the overlying layers. Similar, on the client-side it is a measure of the required time of de-serializing the invocation data of the server's response.

- The *Execution Time* is a server-side QoS property. It is a measure about the complete required time of a client's request, i.e., unmarshaling, processing, and marshaling.

3 Patterns for Measuring Performance-Related QoS Properties

In a distributed system, the client invokes the the server's remote objects, the server receives the incoming requests, processes them, and returns the response. During this process, the negotiated performance-related QoS properties within an SLA have to be measured. This section explains patterns for measuring performance-related QoS properties, by listing the the forces and consequences of each pattern for use on the client- and on the server-side.

Pattern: QOS INLINE

An SLA contains negotiated performance-related QoS properties where only the elapsed time of a remote object invocation is relevant to the client, i.e., the round-trip time. For the server it is relevant to measure the elapsed time of processing a client's requests, i.e., the processing time. The server may be interested to find some possible bottle-necks within the remote object's behaviour as well.

How can the client's and the remote object's implementation be instrumented for measuring performance-related QoS properties?

Consider the typical scenario of measuring performance-related QoS properties in distributed systems. The client invokes some server's remote object via a underlying middleware (see Section 2.1). The middleware transmits the client's request to the remote object using a network. The remote object's middleware receives the request, the remote object processes the request, and returns the response back to the client. The client's and the remote object's implementation have to be instrumented to measure the SLA's performance-related QoS properties.

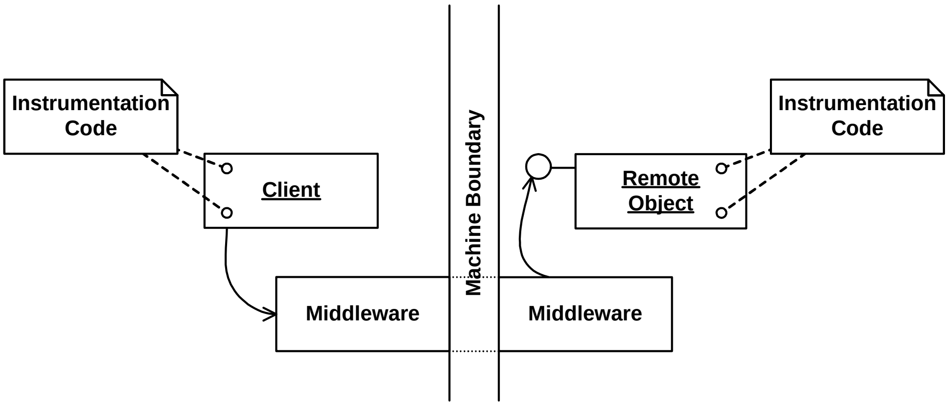


Figure 4: The QOS INLINE pattern

Instrument the client' and the remote object' implementation with local measuring points by placing them directly into their implementation.

Figure 4 shows the QOS INLINE pattern. The client invokes a remote object using a middleware and wants to measure the elapsed time of the remote object invocation. Hence, the client's implementation can be instrumented for measuring the round-trip time. On the server-side, the remote object receives the client's request and has to measure the processing time. Again, the remote object's implementation can be instrumented directly with local measuring points.

On the client-side, the round-trip time can be measured precisely by calculating the time difference between sending the request and receiving the response. The client-side implementation of the QOS INLINE pattern is simple and affects the client's behaviour only slightly.

On the server-side, the processing time of the client's request can be measured precisely because multiple measurement points can be placed at arbitratry places in the remote object's implementation. Hence, it is also possible to find bottle-necks within the processing of the client's request. Dependent on the number of measuring points, the implementation of the server-side QOS INLINE pattern does have insignificant affect on the remote object's performance.

The QOS INLINE pattern does not provide a good separation of concerns because the measuring points are placed into the source code directly. Also, the QOS INLINE pattern is not a reusable solution because existing clients and remote objects have to be instrumented and redeployed individually. In case the client and the server are measuring the negotiated performance-related QoS properties independently, both can fake the QoS measurements.

A general consequence of the QOS INLINE pattern is that not many performance-related QoS properties can be measured. At the client-side, it is easy to measure the round-trip time, but, difficult to measure performance-related QoS properties that have to be measured in some underlying network layers, such as the network latency. It is easy to measure the processing time at the server-side, and the round-trip time at the client side. But on both sides it is difficult to measure performance-related QoS properties that have to be measured in some underlying layers, such as the network latency. Assuming a small number of measuring points, separate tools, such as packet sniffers, can be utilized to measure the performance-related QoS properties that are not measurable with the QOS INLINE pattern.

Known Uses:

- Because of the simplicity of the QOS INLINE pattern, every source code can be extended with time measurements. The QOS INLINE pattern can not be applied in distributed systems only, also in local function calls or object method invocations. Using the QOS INLINE pattern is advisable if the QoS measurements are relevant in the client's and remote object's source code.
- In [10], the authors extend the client's and remote object's implementation with local measuring points to measure performance-related QoS properties in web service-oriented distributed systems.

Pattern: QOS WRAPPER

The negotiated SLAs between client and server include performance-related QoS properties with respect to the elapsed times of remote object invocations. The client and remote object have to be instrumented for measuring the negotiated performance-related QoS properties. The client' and the remote object's implementation should be instrumented with a reusable solution that provides a good separation of concerns.

Which solution is reusable and provides a good separation of concerns for instrumenting the client and the remote objects for measuring performance-related QoS properties?

Measuring performance-related QoS properties during remote object invocations can be done by instrumenting the client's or remote object's implementation directly. But, this solution does not provide a good separation of concerns and reusability. It is not possible to attach the measuring of the performance-related QoS properties to existing clients and remote objects without redeployment. For improvement, the performance-related QoS properties have to be measured separated from the client's and remote object's implementation.

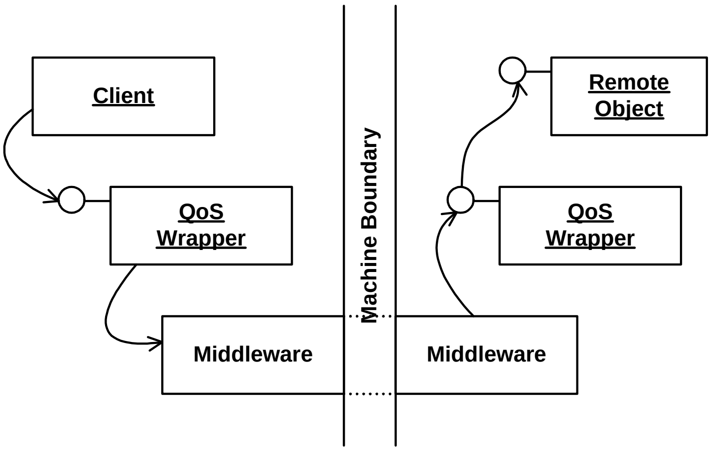


Figure 5: The QOS WRAPPER pattern

Instrument the client's and remote object's implementations with local QOS WRAPPERS that are responsible for measuring the performance-related QoS properties. Let the clients invoke the remote objects using a client-side QOS WRAPPER. Extend the remote objects with a server-side QOS WRAPPER that receives the client's requests.

Figure 5 illustrates the QOS WRAPPER pattern. The client invokes a remote object using a client-side QOS WRAPPER that offers the client the remote object's interfaces, takes over the remote object invocation, and the measuring of the performance-related QoS properties. At the server-side, the QOS WRAPPER processes the incoming requests by invoking the corresponding remote object, measures the server-side performance-related QoS properties separated from the remote object's implementation, and returns back the remote object's response to the requesting client.

Every client and remote object can be instrumented with a local QOS WRAPPER, providing a uniform measuring of the performance-related QoS properties and a reusable solution. Also, a QOS WRAPPER provides a good separation of concerns because it measures the performance-related QoS properties separated from the client's and remote object's implementation.

In case the client and the server are measuring the negotiated performance-related QoS properties independently, both can fake the QoS measurements. On the client-side, the remote object invocations are insignificantly lengthened because the client invokes the remote object not directly, but via the QOS WRAPPER. Hence, a client-side QOS WRAPPER provides precise QoS measurements. A server-side QOS WRAPPER can insignificantly lengthen the remote object invocations as well, but, it measures the QoS properties precisely.

A general consequence of the QOS WRAPPER pattern is that not many performance-related QoS properties can be measured. The client's QOS WRAPPER can measure the round-trip time and the server's QOS WRAPPER the processing time. But on both sides it is difficult to measure performance-related QoS properties that have to be measured in some underlying layers, such as the network latency. Separate tools, such as packet sniffers, can be utilized for measuring the performance-related QoS properties.

The client-side QOS WRAPPER can be implemented following the CLIENT PROXY pattern [4], whereas the server-side QOS WRAPPER can be implemented following the INVOKER [22] pattern.

Known Uses:

- Afek et al. [1] implemented a framework for QoS-aware remote object invocations over an ATM network. The authors extended the Java RMI interface by providing an API to the clients. Following the QOS WRAPPER pattern, the client-side API ensures QoS by providing a good separation of concerns. A server-side QOS WRAPPER server acquires and arranges the service with the desired QoS.
- Mani and Nagarajan [10] illustrate an example of a QOS WRAPPER for measuring the performance-related QoS of web services. The implementation of the measurements follows the QOS INLINE pattern by putting time calculations within the automatically generated QOS WRAPPER. In web service-oriented distributed systems, clients invoke the remote web services via a stub, which can be automatically generated using a `wsdl2java` tool. The automatically generated stubs can be extended with the required QoS measurements following the QOS WRAPPER.

Pattern: QOS INTERCEPTOR

Clients and remote objects have to be instrumented for measuring performance-related QoS properties with a good separated, reusable and precise solution. Because of SLA negotiations it is required to measure as many as possible performance-related QoS properties that can be the reasons for long-running remote object invocations and SLA violations. Access to the middleware's implementation is given, allowing its instrumentation for measuring the required performance-related QoS properties also on the lower layers of the invocations or the wire.

How should the middleware be instrumented for measuring performance-related QoS properties of remote object invocations? How should the middleware be designed and implemented to be dynamically configureable by the middleware users?

Let's consider that the client and the server's remote object have access to their underlying middleware and can instrument it individually and dynamically for measuring the performance-related QoS properties. The desired solution requires that the middleware offers facilities of attaching the implementation of the required QoS measuring dynamically. The middleware's instrumentation should be reusable, enhancing the deployment of new clients and remote objects. Furthermore, it should be possible to measure all relevant performance-related QoS properties (see Section 2.2) precisely, having a good separation of concerns.

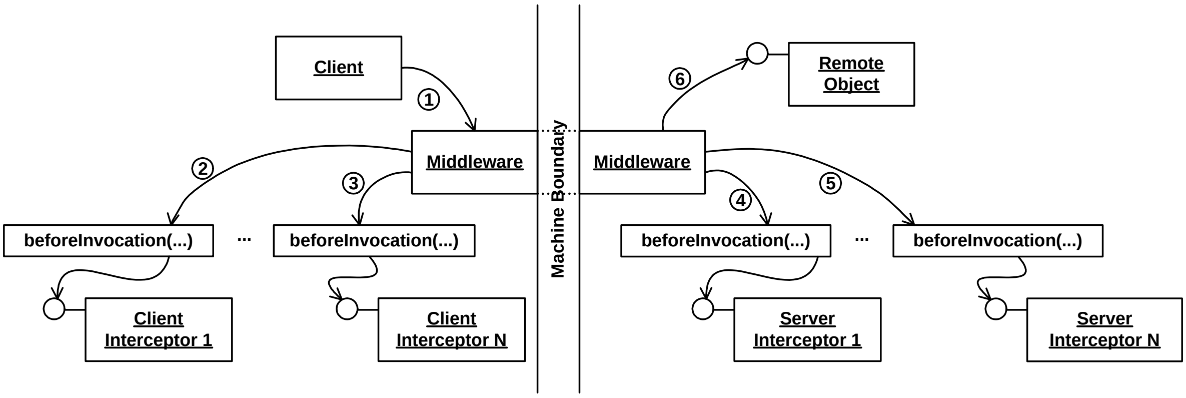


Figure 6: The QOS INTERCEPTOR pattern

Hook QOS INTERCEPTORS into the invocation path that are responsible for measuring the performance-related QoS properties. Provide possibilities in the middleware for attaching a QOS INTERCEPTOR dynamically, such as APIs or configuration files.

Figure 6 demonstrates the QOS INTERCEPTOR pattern, which can be used on the client- and the server-side. The only requirement is that the middleware provides the feature of attaching new QOS INTERCEPTORS. In this case, it is possible to attach and replace QOS INTERCEPTORS dynamically. Multiple QOS INTERCEPTORS can be placed in the invocation path, where each of them is responsible for measuring different performance-related QoS properties. Hence, it is possible to find bottle-necks of long-running remote object invocations. Having access to the middleware's implementation results in more precise measurements of performance-related QoS properties.

The QOS INTERCEPTOR has the benefit that the client's and remote object's implementations do not have to be instrumented for measuring the performance-related QoS properties. The client's and remote object's middleware are instrumented to hook QOS INTERCEPTORS into the invocation path. A QOS INTERCEPTOR provides a good separation of concerns because the measuring is separated from the client's and remote object's implementation. Because a QOS INTERCEPTOR is hooked in the client's or remote object's local middleware, a precise measuring of almost all performance-related QoS properties can be achieved. In addition, a QOS INTERCEPTOR is reusable because existing QOS INTERCEPTORS can be attached dynamically into the middleware of existing clients and remote objects.

Placing multiple QOS INTERCEPTORS into the invocation path can result in long-running remote object invocations, affecting the client's or remote object's performance. Furthermore, the middleware's complexity increases by providing hooks or interfaces for attaching and changing QOS INTERCEPTORS in the invocation path dynamically. In case the client and the server are measuring the negotiated performance-related QoS properties independently, both can fake the QoS measurements.

The QOS INTERCEPTOR pattern is an extension of the INVOCATION INTERCEPTOR pattern [22].

Known Uses:

- Many middleware infrastructures for remote object invocation offer the facilities for adding required QOS INTERCEPTORS into the invocation path, such as OpenORB[3], .NET Remoting[4], Apache Axis2, or Apache CXF. A difference between existing technologies lies in the naming of the QOS INTERCEPTOR. For example, in OpenORB and Apache CXF they are called *interceptors*, in Apache Axis2 *handlers*, and in the .NET Remoting framework the *RealProxy* has to be extended to intercept the remote object invocations.
- To provide a good separation of concerns, the authors of the QoS CORBA Component Model (QOSCCM) [14] use the QOS INTERCEPTOR pattern to easily adapt an application for measuring performance-related QoS properties.

Pattern: QOS REMOTE PROXY

In distributed systems, the client and the remote object do not have to be necessarily located in the same local area network (LAN). In this case, the client invokes the remote object via a wide area network (WAN), such as the Internet.

How to introduce a good separated, reusable, and uniform infrastructure for measuring the performance-related QoS properties in the case when client and remote objects are not located in the same LAN?

In many cases, the server hosts the remote object and is not located in the client's LAN. Hence, the client has to access the remote object via a WAN. Client and server want to measure performance-related QoS properties. The desired solution should be uniform for each client and remote object, enhancing the deployment of new clients and remote objects. Also, a good separated and reusable QoS measurement infrastructure is desired.

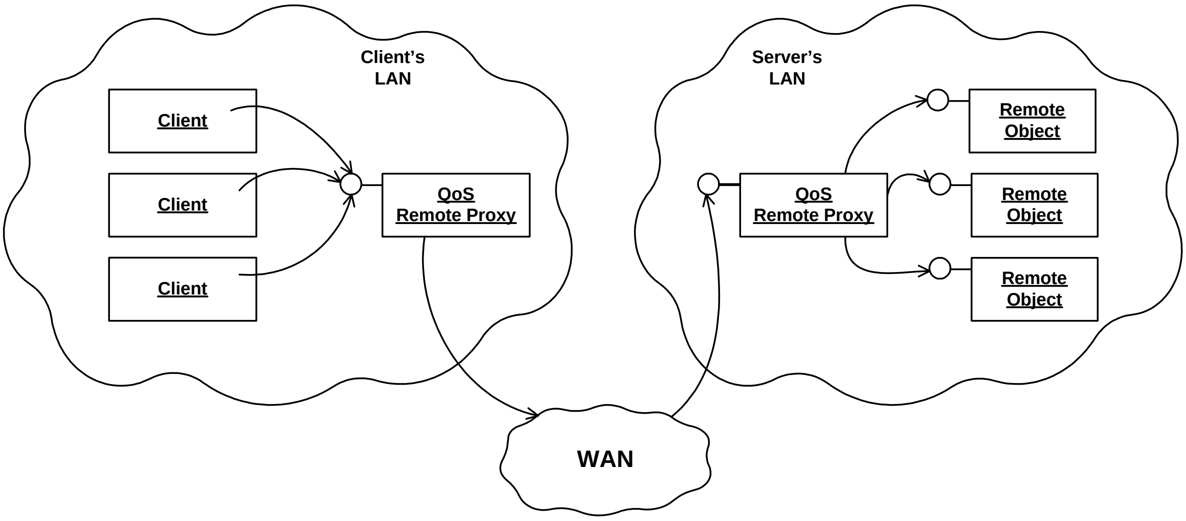


Figure 7: The QOS REMOTE PROXY pattern

Implement and setup a QOS REMOTE PROXY in the client's and remote object's LAN that takes over the responsibility of measuring the performance-related QoS properties. In the client's LAN, configure each client to invoke the remote objects via the LAN's QOS REMOTE PROXY. In the server's LAN, make each remote object only be accessible via a QOS REMOTE PROXY.

[3] `http://openorb.sourceforge.net/`

[4] `http://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.71).aspx`

Figure 7 shows an infrastructure where the client and the remote object are not located in the same LAN. As shown, the QOS REMOTE PROXY pattern can be applied in both LANs. The client's QOS REMOTE PROXY receives the client's request, performs the required QoS measurements, and forwards the request to the remote object's LAN. A server-side QOS REMOTE PROXY receives the client's requests (directly or via the client's QOS REMOTE PROXY), performs the required QoS measurements, and forwards the request to the appropriate remote object. After the remote object processed the request, it sends the response back to the server-side QOS REMOTE PROXY that measures the required performance-related QoS properties and forwards the response to the requestor. The client-side QOS REMOTE PROXY receives the response (from the remote object directly or from the server-sid QOS REMOTE PROXY), performs QoS measuring, and forwards the response to the appropriate client.

In the client's and the server's LAN, a QOS REMOTE PROXY provides a good separation of concerns because the measuring of the performance-related QoS properties is separated. Also, there is no impact on the client's and server's performance. In addition a QOS REMOTE PROXY is a reusable solution. Each new client can be configured to invoke the remote object via the client's LAN QOS REMOTE PROXY. Also, it is possible to configure each remote object that is only accessible via the server's LAN QOS REMOTE PROXY.

At minimum one extra hop in the client's and server's LAN is needed because of accessing the QOS REMOTE PROXY instead of accessing the WAN or the remote object directly. Hence, the measurements of the performance-related QoS properties at the QOS REMOTE PROXY differ from the client's and remote object's local QoS measurements. In case the client and the server are measuring the negotiated performance-related QoS properties independently, both can fake the QoS measurements.

A client-side QOS REMOTE PROXY can affect the client's performance slightly. But, a QOS REMOTE PROXY can impact the performance of the client's LAN because each client has to invoke the remote object via the QOS REMOTE PROXY. On the server-side, a QOS REMOTE PROXY does not affect the performance of the remote object directly, but, it can have an impact on the server's LAN. A QOS REMOTE PROXY inside the server's LAN can be implemented as a load-balancer, gateway, reverse proxy, dispatcher, as well as a firewall following the appropriate patterns [4].

The QOS REMOTE PROXY does not necessarily require that the client and the remote object are located in different LANs. In a case where client and remote object are located in the same LAN, the setup of one QOS REMOTE PROXY inside the LAN is adequate.

Known Uses:

- Wang et al. [20] introduce a QoS-Adaptation proxy that receives the clients' requests, performs the QoS measurements, and forwards the clients' requests to their destinations. The clients' applications remain unchanged while the proxy performs the necessary adaptations and QoS measurements.
- The Corba IIOP specifications [13] introduce the VisiBroker [3] environment that uses the QOS REMOTE PROXY pattern for measuring the performance-related QoS properties.
- The Apache TCPMon[5] tool can be instrumented to serve as a proxy between the clients and the server's remote objects. An implementation of the QOS REMOTE PROXY is to extend this tool for measuring performance-related QoS properties.

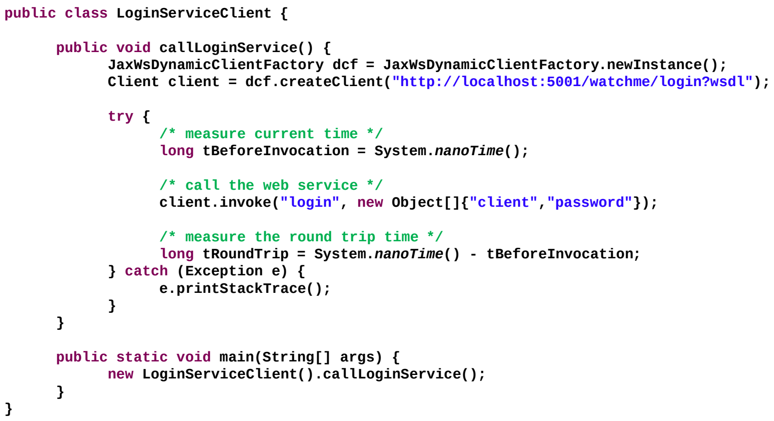
[5] http://ws.apache.org/commons/tcpmon/

4 Selected Example: Measuring performance-related QoS properties of web services

This section exemplifies the presented patterns for measuring performance-related QoS properties within a web service-oriented distributed system [15, 2]. We implemented the clients and remote objects using the Apache CXF web service framework. In this example we implemented a web service which offers the functionality to login into a remote system. The service's operation receives a username and a password from the client and checks if the client is authorized to enter. In the following, we present the client-side implementation of the presented patterns.

Pattern: QOS INLINE

Figure 8 shows a code excerpt of a client that invokes a web service and measures the round-trip time following the QOS INLINE pattern. We used the Apache CXF's feature of implementing a dynamic client where we do not have to use the `wsdl2java` tool for generating the web service's stub explicitly.



```
public class LoginServiceClient {

    public void callLoginService() {
        JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
        Client client = dcf.createClient("http://localhost:5001/watchme/login?wsdl");

        try {
            /* measure current time */
            long tBeforeInvocation = System.nanoTime();

            /* call the web service */
            client.invoke("login", new Object[]{"client","password"});

            /* measure the round trip time */
            long tRoundTrip = System.nanoTime() - tBeforeInvocation;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new LoginServiceClient().callLoginService();
    }
}
```

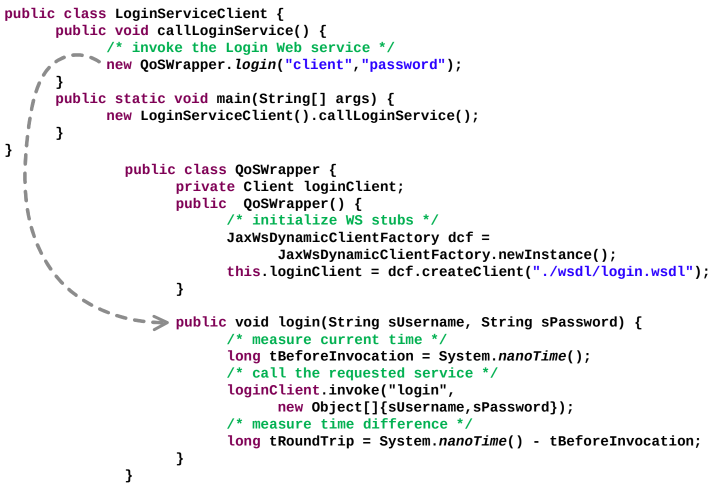
Figure 8: Measuring the round-trip time following the QOS INLINE pattern

The client offers a `callLoginService` method to invoke the web service's `Login` operation. First, we have to instantiate the `JaxWsDynamicClientFactory`, following the FACTORY pattern [6]. Then, the client is created by using the previously instantiated FACTORY. The client puts two QoS measuring points around the actual web service invocation – `client.invoke(...)` – to measure the round-trip time of the web service invocation.

Pattern: QOS WRAPPER

In Figure 9 we illustrates a web service client that measures the round-trip of the web service invocation following the QOS WRAPPER pattern. Instead of placing measuring points for the round-trip time in the client's implementation directly, the client invokes the web service via a local QOS WRAPPER. The implemented QOS WRAPPER offers the same interface to the client as the remote object. In this example, the QOS WRAPPER takes over the responsibility of measuring of the round-trip time of a web service invocation.

Instead of invoking the web service directly, the client calls the `invoke` method of the `QoSWrapper`. Within the `invoke` method, the QOS WRAPPER measure the elapsed time of the web service invocation, i.e., the round-trip time.

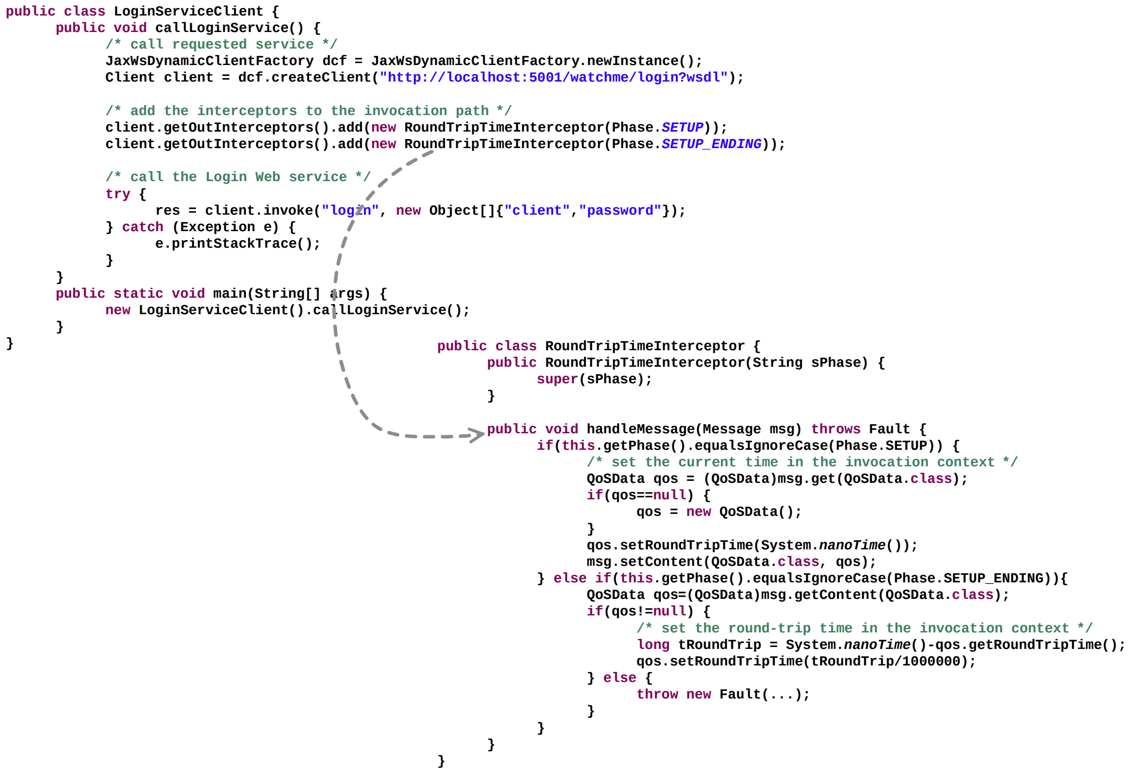


```
public class LoginServiceClient {
      public void callLoginService() {
            /* invoke the Login Web service */
            new QoSWrapper.login("client","password");
      }
      public static void main(String[] args) {
            new LoginServiceClient().callLoginService();
      }
}
              public class QoSWrapper {
                    private Client loginClient;
                    public  QoSWrapper() {
                          /* initialize WS stubs */
                          JaxWsDynamicClientFactory dcf =
                                JaxWsDynamicClientFactory.newInstance();
                          this.loginClient = dcf.createClient("./wsdl/login.wsdl");
                    }
                    public void login(String sUsername, String sPassword) {
                          /* measure current time */
                          long tBeforeInvocation = System.nanoTime();
                          /* call the requested service */
                          loginClient.invoke("login",
                                new Object[]{sUsername,sPassword});
                          /* measure time difference */
                          long tRoundTrip = System.nanoTime() - tBeforeInvocation;
                    }
              }
```

Figure 9: Measuring the round-trip time following the QOS WRAPPER pattern

Pattern: QOS INTERCEPTOR

The QOS INTERCEPTOR pattern can be implemented easily using the Apache Axis, Apache CXF web services framework or in object-oriented RPC middlewares, such as CORBA, .NET Remoting, and Windows Communication Foundation.



```
public class LoginServiceClient {
      public void callLoginService() {
            /* call requested service */
            JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
            Client client = dcf.createClient("http://localhost:5001/watchme/login?wsdl");

            /* add the interceptors to the invocation path */
            client.getOutInterceptors().add(new RoundTripTimeInterceptor(Phase.SETUP));
            client.getOutInterceptors().add(new RoundTripTimeInterceptor(Phase.SETUP_ENDING));

            /* call the Login Web service */
            try {
                  res = client.invoke("login", new Object[]{"client","password"});
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
      public static void main(String[] args) {
            new LoginServiceClient().callLoginService();
      }
}
                                          public class RoundTripTimeInterceptor {
                                                public RoundTripTimeInterceptor(String sPhase) {
                                                      super(sPhase);
                                                }

                                                public void handleMessage(Message msg) throws Fault {
                                                      if(this.getPhase().equalsIgnoreCase(Phase.SETUP)) {
                                                            /* set the current time in the invocation context */
                                                            QoSData qos = (QoSData)msg.get(QoSData.class);
                                                            if(qos==null) {
                                                                  qos = new QoSData();
                                                            }
                                                            qos.setRoundTripTime(System.nanoTime());
                                                            msg.setContent(QoSData.class, qos);
                                                      } else if(this.getPhase().equalsIgnoreCase(Phase.SETUP_ENDING)){
                                                            QoSData qos=(QoSData)msg.getContent(QoSData.class);
                                                            if(qos!=null) {
                                                                  /* set the round-trip time in the invocation context */
                                                                  long tRoundTrip = System.nanoTime()-qos.getRoundTripTime();
                                                                  qos.setRoundTripTime(tRoundTrip/1000000);
                                                            } else {
                                                                  throw new Fault(...);
                                                            }
                                                      }
                                                }
                                          }
```

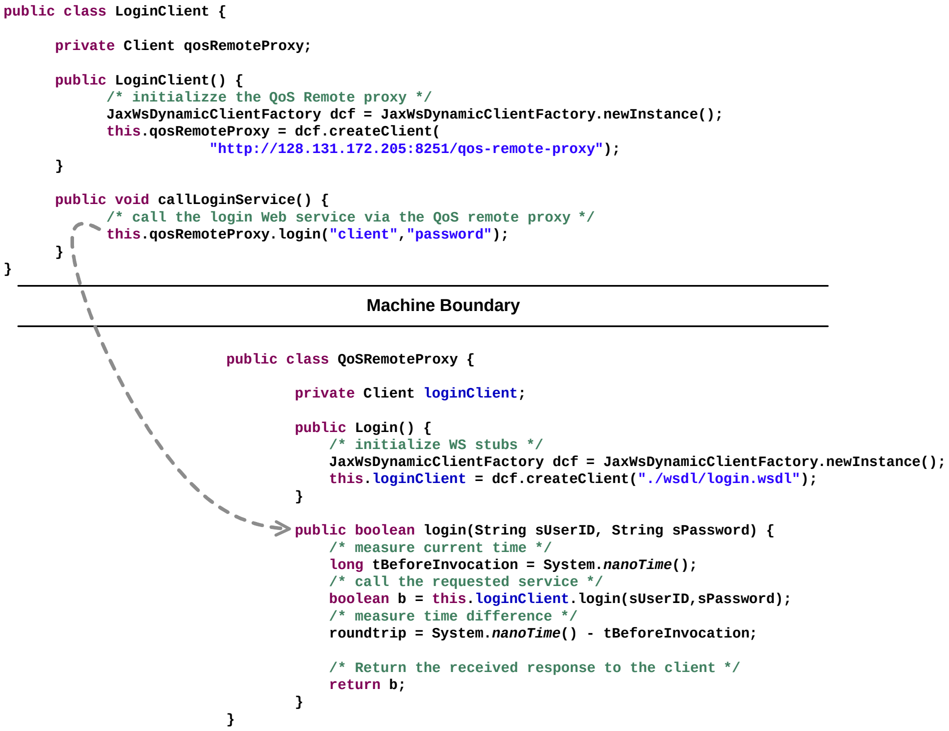
Figure 10: Measuring the round-trip time following the QOS INTERCEPTOR pattern

Figure 10 shows an excerpt of the client's implementation and the implemented QOS INTERCEPTOR for measuring the round-trip time of a web service invocation. First, the client initializes the generated stubs of the web service, creates objects of the interceptors, and defines where to place them into the invocation path. In our example, the `RoundTripTimeInterceptor` measures the round-trip time between the `SETUP` and `SETUP_ENDING` phases of the client's `OUT` chain. The Apache CXF web service framework provides facilities for attaching the interceptors to the invocation path by calling the `getOutInterceptors().add()` method.

The `handleMessage` method of the `RoundTripTimeInterceptor` contains the business logic of the QOS INTERCEPTOR. In the `SETUP` phase, the interceptor puts the current time into the INVOCATION CONTEXT – `QoSData` – of the message. In the `SETUP_ENDING` phase, the interceptor calculates the time difference – the round-trip time – and puts it again into the INVOCATION CONTEXT.

Pattern: QOS REMOTE PROXY

The QOS REMOTE PROXY offers interfaces to the clients to invoke remote objects and takes over the responsibility of measuring the performance-related QoS properties. In comparison to the previously shown QOS WRAPPER example, the client does invoke the web service via the QOS REMOTE PROXY over the LAN and not directly.



```
public class LoginClient {

      private Client qosRemoteProxy;

      public LoginClient() {
            /* initializze the QoS Remote proxy */
            JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
            this.qosRemoteProxy = dcf.createClient(
                        "http://128.131.172.205:8251/qos-remote-proxy");
      }

      public void callLoginService() {
            /* call the login Web service via the QoS remote proxy */
            this.qosRemoteProxy.login("client","password");
      }
}
```

```
public class QoSRemoteProxy {

        private Client loginClient;

        public Login() {
            /* initialize WS stubs */
            JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
            this.loginClient = dcf.createClient("./wsdl/login.wsdl");
        }

        public boolean login(String sUserID, String sPassword) {
            /* measure current time */
            long tBeforeInvocation = System.nanoTime();
            /* call the requested service */
            boolean b = this.loginClient.login(sUserID,sPassword);
            /* measure time difference */
            roundtrip = System.nanoTime() - tBeforeInvocation;

            /* Return the received response to the client */
            return b;
        }
}
```

Figure 11: Measuring the round-trip time following the QOS REMOTE PROXY pattern

We illustrate our Apache CXF implementation of a QOS REMOTE PROXY in Figure 11. The client invokes the `login` method of the QOS REMOTE PROXY instead of calling the web service's `login` operation directly. As illustrated, the QOS REMOTE PROXY performs the measuring of the performance-related QoS properties. In our example, the implemented QOS REMOTE PROXY measures the round-trip time of the web service invocation.

5 Discussion

This section provides information about possible ways of the patterns' implementations and lists possibilities of future work to store and evaluate the QoS measurements.

Aspect-oriented Implementation of the Patterns

A possible way to implement some of the presented patterns and to provide a good separation of concerns, is to follow the aspect-oriented programming (AOP) paradigm. An aspect is a construct that contains the separated concern's implementation and a description of how to weave it into the code [9, 8].

To improve the separation of concerns within the QOS INLINE pattern, its implementation can follow the AOP paradigm. Implementing an aspect-oriented QOS INLINE solution results in a QOS WRAPPER. The aspects of measuring performance-related QoS properties are separated from the client's or remote object's implementation, resulting in a good separation of concerns. Furthermore, the measuring aspects can be reused and attached to new deployed clients and remote objects.

The QOS INTERCEPTOR pattern can be implemented following the AOP paradigm. Such a solution is interesting if the middleware does not provide hooks for placing a QOS INTERCEPTOR into the invocation path.

In the case where the QOS REMOTE PROXY has additional responsibilities to measuring performance-related QoS properties, its implementation can follow the AOP paradigm. Hence, it is possible to separate the QOS REMOTE PROXY'S QoS measuring from its business logic.

Automatic Generation of the Patterns

It is possible to generate the presented patterns and their components for measuring the performance-related QoS properties automatically. In general, all reusable parts can be generated automatically.

Following the QOS INLINE pattern, it is difficult to generate the measuring points into existing clients or remote objects. Only clients and remote objects that have to be newly deployed can be generated automatically including the measuring points. The client's or the remote object's implementation has to be developed manually. Following the AOP paradigm, it is possible to generate the required aspects for measuring the performance-related QoS properties automatically.

A QOS WRAPPER can be generated automatically and the generic parts of the client's and remote object's implementations for accessing the client or remote object via the QOS WRAPPER.

QOS INTERCEPTORS can be generated and attached to the client's and remote object's middleware automatically for measuring the performance-related QoS properties. Existing clients and remote objects can be extended easily.

It is possible to generate a QOS REMOTE PROXY automatically. New clients can be generated and configured to access the remote objects only via the generated QOS REMOTE PROXY. Also, it is possible to generate the remote objects automatically and to configure them that they are only accessible via a QOS REMOTE PROXY. Existing clients and remote objects have to be re-configuried or re-deployed.

Storing and Evaluating the QoS Measurements

This paper's patterns cover the aspects of measuring performance-related QoS properties in distributed systems. A further important aspect is how to store and evaluate the QoS measurements. Many possibilities of storing the QoS measurements exist, such as using local log files or databases. It is also possible to forward the measurements to a QoS monitor by using, for example communication channels [7]. The QoS monitor can be either centralized or de-centralized, and can evaluate the QoS measurements immediately or at later stages.

The implementation of the QOS INTERCEPTORS can follow the INVOCATION CONTEXT pattern for storing an invocation's QoS measurements. The QOS INTERCEPTOR pattern provides possibilities to forge the QoS measurements by simply changing the stored QoS measurements in the INVOCATION CONTEXT or for whatever reasons. The available information of the INVOCATION CONTEXT depends on the interceptor's location in the invocation path. To overcome this problem, a centralized QoS monitor can be developed which receives the INVOCATION CONTEXT when a QOS INTERCEPTOR is executed.

Evaluating performance-related QoS properties brings the necessity to avoid violations of the negotiated QoS concerns within the SLAs. A monitoring component is required that measures the QoS properties in a predictive and pro-active way to give warnings to the system users to avoid possible future violations. Still a huge research challenge is to implement systems that react to the warnings and avoid violations automatically.

A possibility of storing and evaluating performance-related QoS properties in distributed systems is to use an event-based system. A Complex Event Processing (CEP) engine receives QoS events from the clients and services, stores them, and evaluates them. A CEP evaluates the QoS events using predefined QoS rules that are checked against the received events during the runtime of the system.

6 Conclusion

The contribution of this paper are four patterns that focus on measuring performance-related QoS properties in distributed systems. The patterns are extensions of well-known existing patterns, presented in the Gang of Four (GoF) book [6], the Pattern-Oriented Software Architecture (POSA) series [5, 18, 4], and in the Remoting Patterns book [22]. We highlighted the patterns concerning their impact on the client's or service's performance, their reusability, the extent of separation of concerns, the preciseness of the QoS measurements, and the vulnerability to forgery of the QoS measurements.

The paper gave background information on the existing patterns as well as on the pattern's relevant performance-related QoS properties. We exemplified the patterns for measuring client-side performance-related QoS properties in web service-oriented distributed systems. As a future work we intend to describe and define patterns on storing and evaluating the gathered QoS measurements. This paper's patterns should help software architects and developers in designing architectures for measuring performance-related QoS properties in a distributed system.

Acknowledgment

We would like to thank our shepherd Andy Carlson for his constructive and supporting help during the shepherding process to improve the quality of the patterns and the paper itself.

Also, this work was supported by the European Union FP7 project COMPAS, grant no. 215175.

References

[1] Y. Afek, M. Merritt, and G. Stupp. Remote Object Oriented Programming with Quality of Service or Java's RMI over ATM, 1996.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, October 2003.

[3] Borland. VisiBroker – A Robust CORBA Environment for Distributed Processing, 2009.

[4] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley, 2007.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[7] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[10] A. Mani and A. Nagarajan. Understanding quality of service for Web services – Improving the performance of your Web services, 2002. `http://www.ibm.com/developerworks/library/ws-quality.html`, *last accessed: May 2010.*

[11] E. Oberortner, U. Zdun, and S. Dustdar. Tailoring a model-driven Quality-of-Service DSL for Various Stakeholders. In *MISE '09: Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 20–25, Washington, DC, USA, 2009. IEEE Computer Society.

[12] L. O'Brien, P. Merson, and L. Bass. Quality Attributes for Service-Oriented Architectures. In *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[13] O. M. G. (OMG. Common Object Request Broker Architecture/Internet Inter-ORB Protocol (CORBA/IIOP), 2008.

[14] O. M. G. (OMG. Quality Of Service For CCM (QOSCCM), 2008.

[15] M. P. Papazoglou. *Web Services: Principles and Technology*. Pearson, Prentice Hall, 2008.

[16] S. Ran. A Model for Web Services Discovery with QoS. *SIGecom Exch.*, 4(1):1–10, 2003.

[17] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 205–212, Washington, DC, USA, 2006. IEEE Computer Society.

[18] D. C. Schmidt, H. Rohnert, M. Stal, and D. Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[19] The Service Level Agreement Zone. SLA Information Zone, 2007. `http://www.sla-zone.co.uk/` *(last accessed: 09/2010).*

[20] Q. Wang, Q. Ye, and L. Cheng. An Inter-Application and Inter-Client Priority-Based QoS Proxy Architecture for Heterogeneous Networks. In *ISCC '05: Proceedings of the 10th IEEE Symposium on Computers and Communications*, pages 819–824, Washington, DC, USA, 2005. IEEE Computer Society.

[21] W. D. Yu, R. B. Radhakrishna, S. Pingali, and V. Kolluri. Modeling the Measurements of QoS Requirements in Web Service Systems. *Simulation*, 83(1):75–91, 2007.

[22] U. Zdun, M. Kircher, and M. Völter. Remoting Patterns. *IEEE Internet Computing*, 8:60–68, 2004.

Pattern	Forging the QoS Measurements	Impact on Performance	Precise QoS Measurements	Separation of Concerns	Reusability
QOS INLINE	Server and client can forge the QoS measurements in case they are measuring them independently of each other	Minimal performance impact	Precise measurement of a small number of QoS properties, such as round-trip time or processing time.	No, because modification of client's or remote object's implementation necessary	Bad, because of direct placement of QoS measuring points into client's or remote object's implementation
QOS WRAPPER	Server and client can forge the QoS measurements in case they are measuring them independently of each other	Insignificant lengthening of the remote object invocations	Minimal variation of a small number of QoS properties, such as round-trip time or processing time.	Yes, because of separated QoS measuring from client's or remote object's implementation	Good, because existing QOS WRAPPER can be reused for new clients and remote objects.
QOS INTERCEPTOR	In a combined solution, client and server can forge the QoS measurements by using the INVOCATION CONTEXT pattern	Insignificant lengthening of the remote object invocations	Minimal variation of a large number of QoS properties, such as network latency of marshaling time	Yes, because of separated QoS measuring from client's or remote object's implementation	Good, because existing QOS INTERCEPTOR can be reused for new clients and remote objects.
QOS REMOTE PROXY	Server and client can forge the QoS measurements in case they are measuring them independently of each other	One additional hop in the LAN is required that lengthens the remote object invocations.	An extra hop in the LAN is required to access the QOS REMOTE PROXY leading the imprecise QoS measurements	Yes, because of separated QoS measuring from client's or remote object's implementation	Good, because existing QOS REMOTE PROXY can be reused for new clients and remote objects.

Table 1: Summarization of the presented patterns