Patterns for Measuring Performance-related QoS Properties in Service-oriented Systems

ERNST OBERORTNER, Distributed Systems Group, Information Systems Institute, Vienna University of Technology, Austria, e.oberortner@infosys.tuwien.ac.at
UWE ZDUN, Software Architecture, Faculty of Computer Science, Vienna University, 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 service-oriented systems, clients can access services 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 service provider serious financial consequences or other penalties can follow in case of not fulfilling the SLAs. The service consumer wants to evaluate that the provider complies with the guaranteed SLAs. Designing and developing a QoS-aware service-oriented 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 and utilizing existing patterns. The focus of the patterns lies on the QoS measuring impact on the client's or service's performance, the extend of separation of concerns, the property of reusability, and the preciseness of the measured QoS properties. The patterns help to build efficient solutions to measure performance-related QoS properties in a service-oriented system.

Categories and Subject Descriptors: D.2.11 [**Software Architectures**] Patterns

General Terms: Measurement, Performance, Reliability

1. INTRODUCTION

In a service-oriented system, service level agreements (SLA) between service provider and consumer can exist. An SLA is a contract over a period of time that contains — among other things — agreements on performance-related properties when the consumer invokes the provider's services over a network. SLAs assure that the consumers get the service they pay for and that the service provider fulfills the SLA guarantees. If the server provider can not meet the goals, then serious financial consequences and a diminished reputation 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, a service consumer wants to observe and validate that the service provider does not violate the guaranteed SLAs during the SLA's validity [The Service Level Agreement Zone 2007; Oberortner et al. 2009].

In this work we concentrate on SLA clauses that contain agreements on performance-related Quality of Service (QoS) measurements. During the development of an appropriate QoS monitoring infrastructure, many challenging design decisions must be taken. We concentrate on the design decision about where to measure the SLA's performance-related QoS properties. QoS properties 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 have discovered in a thorough literature review

Author's address: E. Oberortner, Argentinierstrasse 8/184-1, 1040 Vienna, Austria, email: e.oberortner@infosys.tuwien.ac.at
A preliminary version of this paper was presented in a writers' workshop at the 17th Conference on Pattern Languages of Programs (PLoP). PLoP'10, October 16-18, Reno, Nevada, USA. ACM 978-1-4503-0107-7

(such as[Zeng et al. 2008; Keller and Ludwig 2003; Sahai et al. 2002; Michlmayr et al. 2010; Hauck and Reiser 2000; Aurrecoechea et al. 1998]) the following requirements on measuring performance-related QoS properties: The QoS measuring solution should (1) not impact the overall system's performance, (2) deliver precise QoS measurements and should not falsify other QoS measurements, (3) provide separation of concerns in order to not modify the client's or service's implementation, and (4) be reusable for new clients and services.

We document design practices and patterns of measuring performance-related QoS properties, such as round-trip time, network latency, or processing time [O'Brien et al. 2007a; Rosenberg et al. 2006; Ran 2003; Yu et al. 2007]. 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 [Gamma et al. 1995], the Pattern-Oriented Software Architecture (POSA) series [Buschmann et al. 1996; Schmidt et al. 2000; Buschmann et al. 2007], and the Remoting Patterns book [Voelter et al. 2004]. 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 main focus of the patterns lies on service-oriented systems where clients invoke services that reside on servers. The patterns can be applied in synchronous invocations and must be slightly modified in order to be usable in asynchronous invocations. Patterns for message-oriented distributed systems, such as presented in the Enterprise Integration Patterns book [Hohpe and Woolf 2003], can be utilized and extended to implement a QoS-aware communication between the clients and the services. The presented patterns do not take into account how to store or evaluate the measurements.

This paper is organized as follows: In the next section, Section 2, we explain background information about existing patterns in distributed systems, SLAs, and performance-related QoS properties. In Section 3 we present the patterns, its forces and consequences, for measuring the performance-related QoS properties in service-oriented systems. For a better understanding, we exemplify the patterns' implementation in Section 4. In Section 5 we discuss the patterns regarding model-driven development (MDD) to generate the patterns automatically. We conclude the paper in 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 [Voelter et al. 2004].

For accessing the client's middleware, the REQUESTOR pattern can be used [Voelter et al. 2004]. 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 [Voelter et al. 2004] to provide a good separation of concerns and to attach additional information to the client's requests. The CLIENT PROXY invokes the middleware using the REQUESTOR pattern. The implementation of the client's middleware can follow the CLIENT REQUEST HANDLER pattern [Voelter et al. 2004], 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 [Voelter et al. 2004]. A SERVER REQUEST HANDLER receives the incoming requests, performs additional processing, and forwards the requests to the INVOKER of the remote objects. The INVOKER [Voelter et al. 2004] receives the requests from

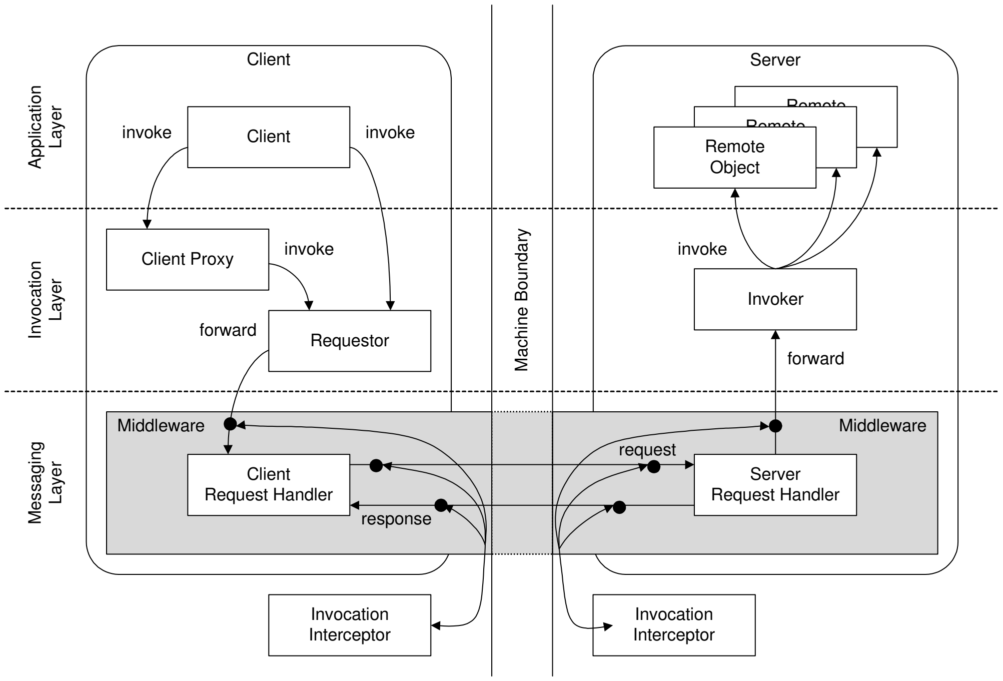


Fig. 1. An overview of existing patterns in distributed systems

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 [Voelter et al. 2004], which is based on the INTERCEPTOR pattern [Schmidt et al. 2000], 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 [Voelter et al. 2004] 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 [Buschmann et al. 2007].

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

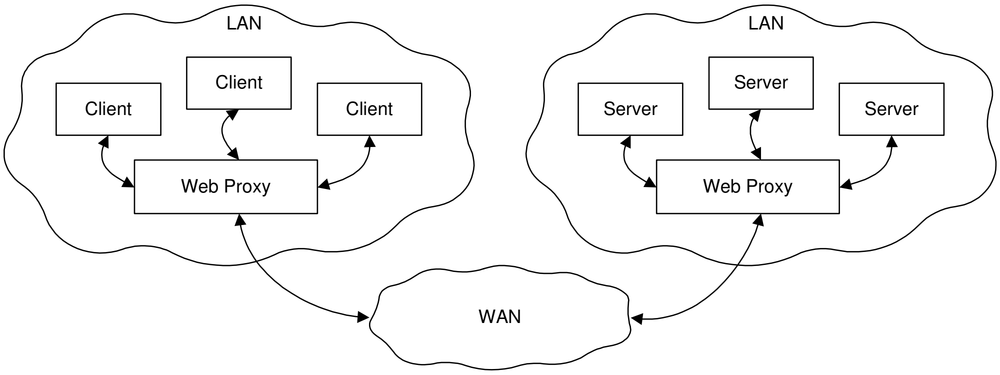


Fig. 2. Using the WEB PROXY pattern

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 Service Level Agreements (SLA)

SLAs are contracts between service providers and service consumers which assure that service consumers get the service they paid for and that the service fulfils the SLAs requirements. In our work, we consider SLA clauses of agreements about the services' performance-related QoS properties (see Section 2.3). An SLA describes technical and nontechnical characteristics of a service, including the services' performance-related QoS properties [Lamanna et al. 2003; Sahai et al. 2001; jie Jin et al. 2002].

Typically, an SLA is defined over a specific time period [Frølund and Koistinen 1998; Lamanna et al. 2003; jie Jin et al. 2002]. For a service provider it could result in serious financial consequences in case of violating the SLAs [Oberortner et al. 2009]. It is therefor important to monitor the negotiated SLAs' performance-related QoS properties and to enforce the services' quality during the SLA's validity [O'Brien et al. 2007a]. A service consumer wants to know if the service provider delivered the service quality as negotiated in the SLA. This is mainly of importance after the SLA's validity.

2.3 Performance-related QoS Properties

A service's performance-related QoS attributes are non-functional properties of the service's performance [O'Brien et al. 2007b]. Service consumers invoke services over the Internet, making it challenging to deliver the service's quality because of the Internet's dynamic and unpredictable nature [Mani and Nagarajan 2002].

Reported in the literature are many performance-related QoS properties that can be measured and monitored in a service-oriented system [O'Brien et al. 2007a; Rosenberg et al. 2006; Ran 2003; Yu et al. 2007]. Within the service consumer's network, clients invoke services that reside in the service provider's network and that are hosted on servers. A server can host multiple services. In Figure 3 we show the basics of some existing remoting middlewares, such as in web services frameworks like Apache CXF [The Apache Software Foundation c] or Apache Axis2 [The Apache Software Foundation b]. The invocation data, between the client and the service, flows in the middleware through so-called chains. A client and a server have an incoming and an outgoing chain – `IN Chain` and `OUT Chain` in Figure 3 – which are responsible for processing the incoming requests and the outgoing responses, respectively. Chains consist of multiple phases, making it possible to specify precisely where to hook INVOCATION INTERCEPTORS into the invocation path.

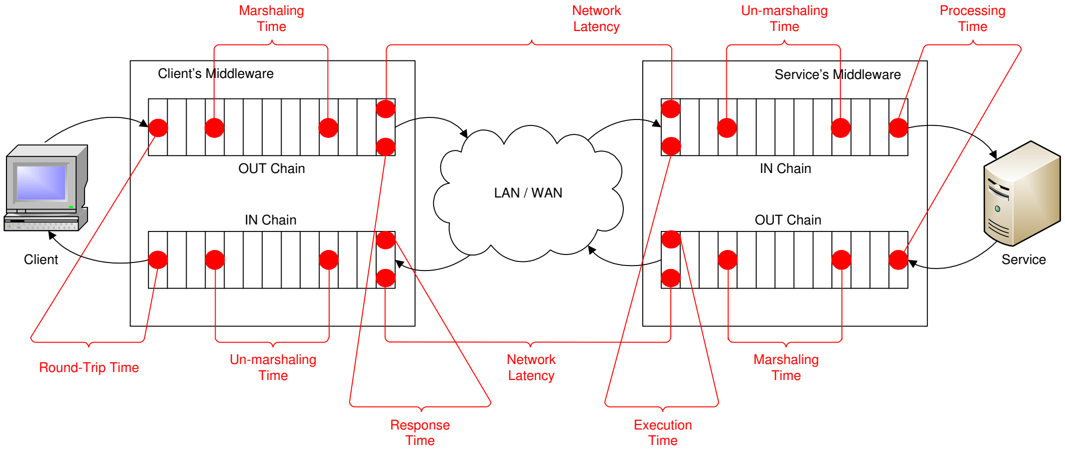


Fig. 3. Measuring points of performance-related QoS concerns

In this work, we differentiate between negotiable and network-specific performance-related QoS properties. Negotiable QoS properties deal with the elapsed time of processing a service request and how long it takes that the service consumer receives a response. Mostly, SLAs do not contain agreements about network-specific QoS properties. For example, a service consumer does not want to know how long it takes to marshal or un-marshal the transmitted data. But, measuring network-specific QoS properties and can be useful to identify, for example, bottle-necks of long-running service invocations.

2.3.0.1 *Negotiable performance-related QoS properties*

—*Round-Trip Time*
For a service consumer, it is important to know how long it takes to receive the requested data or results from the service. The elapsed time between the sending of the client's request and receiving the service's response is referred to the service's round-trip time.

—*Processing Time*
On the server-side the processing time is the elapsed time for processing the client's incoming request. It does not take into account the performance-related QoS properties of processing the incoming requests and outgoing response in the underlying middleware.

—*Response Time*
The response time is a client-side QoS property and measures the elapsed time between transmitting the marshaled invocation data to the server and the reception of the server's response. In the terminology of SLA, the response time often refers to the elapsed time of responding to a problem in case the service is down.

2.3.0.2 *Network-specific performance-related QoS properties*

—*Marshaling Time*
The invocation data must be marshaled for its transmission over the underlying network. At the client-side, the marshaling time measures the elapsed time of marshaling the outgoing invocation data of the service request. At the server-side, the marshaling time measures the elapsed time of marshaling the invocation data of the service's response.

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

—*Network Latency*
The required time for transmitting the marshaled invocation data over the network is called network latency. It requires measuring points at the client- and the server-side. Network latency can be measured during the sending of the client's request and its reception at the server-side. Also, network latency is the elapsed time of transmitting the marshaled service's response from the service to the client over the network.

—*Un-marshaling Time*
The marshaled invocation data must be un-marshaled to be process-able in the overlying layers. At the server-side, the un-marshaling time measures the elapsed time of un-marshaling the incoming invocation data of the service request. At the client-side, the un-marshaling time measures the elapsed time of un-marshaling the invocation data of the service's response.

3. PATTERNS FOR MEASURING QOS IN SERVICE-ORIENTED SYSTEMS

In this section we present four patterns about how to instrument the clients' and the services' implementation to measure the performance-related QoS properties of service invocations. This can be done in various ways and layers of the network communication, such as in the application layer, the network layer, or by extending some used middleware.

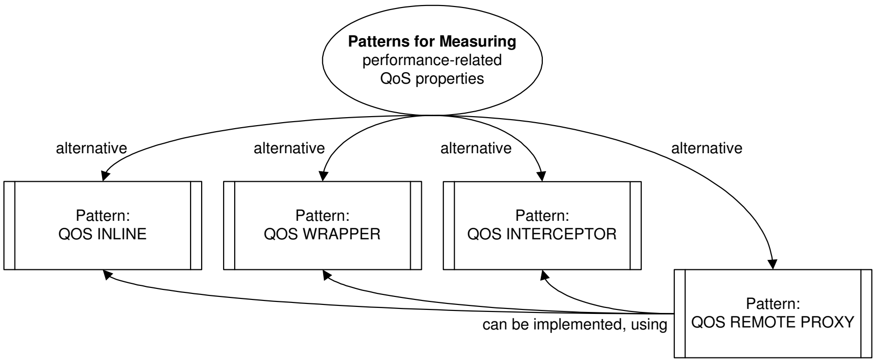


Fig. 4. Requirements on measuring and relationships between the patterns

In Figure 4 we illustrate the relationship between the patterns. The QOS INLINE, QOS WRAPPER, and QOS INTERCEPTOR patterns are implemented locally at the clients or services. The QOS REMOTE PROXY pattern is a centralized measuring solution within the service consumer's or service provider's network.

In the following we explain and analyse each pattern. Measuring the performance-related QoS properties should deliver precise measurements, produce a minimal performance overhead, and should not decrease the system's scalability. The implementation of the measuring pattern should be de-coupled from the services' or clients' implementation in order to provide separation of concerns and reusability.

3.1 Pattern: QOS INLINE

An SLA contains negotiated performance-related QoS properties where only the elapsed time of a service 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 the client's request, i.e., the processing time. The server may be interested to find some possible bottle-necks within the service's behaviour as well.

❖ ❖ ❖

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

Consider a typical scenario of measuring performance-related QoS properties in a service-oriented system. The client invokes some service via an underlying middleware (see Section 2.1). The middleware transmits the client's request to the service over a network. The service's middleware receives the request, the service processes the request, and returns the response back to the client. Now, the client's and the service's implementation have to be instrumented to measure the SLA's performance-related QoS properties.

Therefore,
Instrument the client's and the service's implementation with local measuring points and place them directly into their implementation.

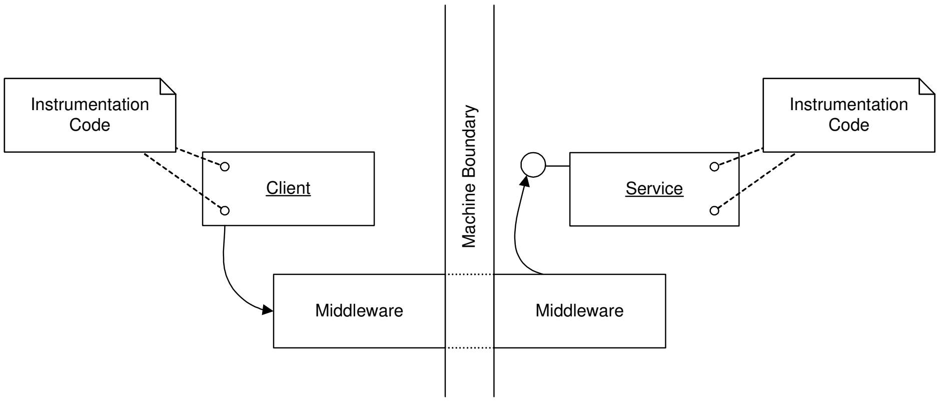


Fig. 5. The QOS INLINE pattern

Figure 5 shows the QOS INLINE pattern. The client invokes the service via a middleware and wants to measure the elapsed time of the service invocation. The client's implementation can be instrumented for measuring the round-trip time. On the server-side, the service receives the client's request, processes it, and measures the processing time. The service'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 elapsed time between sending the service request and receiving the response. The QOS INLINE pattern does not affect the measurements of other performance-related QoS measurements. The client-side implementation of the QOS INLINE pattern affects the client's performance only slightly. On the server-side, the QOS INLINE pattern can measure the processing time precisely without influencing other performance-related QoS measurements. Multiple measurement points can be placed at arbitratry places in the service's implementation, making it possible to find, for example, bottle-necks within the processing of the client's request. Dependent on the number of measuring points, QOS INLINE pattern does not have significant affect on the service's performance.

❖ ❖ ❖

The QOS INLINE pattern does not provide a good separation of concerns because the measuring points are placed into the implementation directly. Also, the QOS INLINE pattern is not a reusable solution because existing clients and services must be instrumented and redeployed individually.

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.

Every source code can be extended with time measurements using the QOS INLINE pattern. The QOS INLINE pattern can not be applied in service-oriented 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 or service's application layer.

Known Uses:

—Mani and Nagarajan [Mani and Nagarajan 2002] explain QoS in service-oriented systems by using the QOS INLINE pattern to measure the elapsed time of a service invocation, i.e., the round-trip time.

3.2 Pattern: QOS WRAPPER

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

❖ ❖ ❖

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

As proposed by the QOS INLINE pattern, measuring performance-related QoS properties during service invocations can be done by placing measuring points into the client's or service's implementation directly. But, this solution does not provide separation of concerns and reusability. It is not possible to attach the measuring of the performance-related QoS properties to existing clients and services without redeployment. For improvement, the performance-related QoS properties have to be measured separated from the client's and service's implementation.

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

Figure 6 illustrates the QOS WRAPPER pattern. The client invokes a service using a client-side QOS WRAPPER that offers the client the service's operations, takes over the service invocation, and measures the performance-related QoS properties. At the server-side, the QOS WRAPPER processes the incoming requests for the service, measures the server-side performance-related QoS properties separated from the service's implementation, and returns back the service's response to the requesting client.

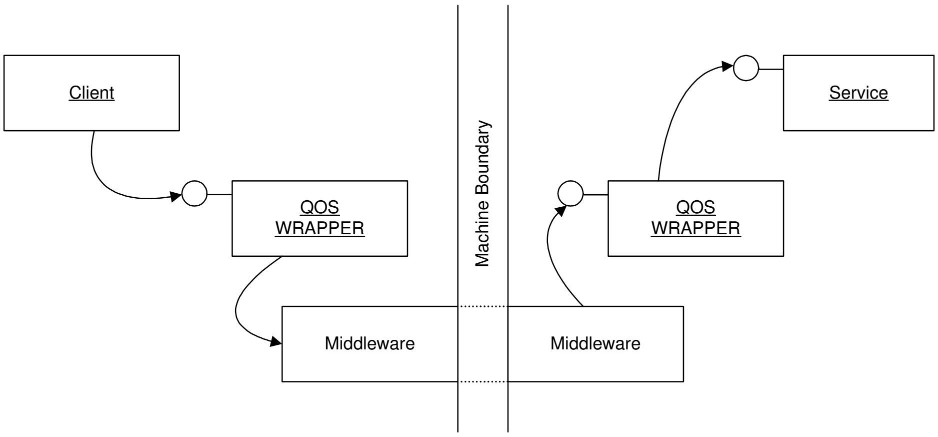


Fig. 6. The QOS WRAPPER pattern

Every client and service can be instrumented with a local QOS WRAPPER, providing a uniform measuring of the performance-related QoS properties and a reusable solution. A QOS WRAPPER provides separation of concerns because it measures the performance-related QoS properties separated from the client's or the service's implementation. Furthermore, a QOS WRAPPER does not impact the overall system's performance because it measures the performance-related QoS properties locally in each client or service.

❖ ❖ ❖

On the client-side, the service invocations are insignificantly lengthened because the client does not invoke the service not directly, but via the QOS WRAPPER. A client-side QOS WRAPPER provides precise QoS measurements. A server-side QOS WRAPPER can insignificantly lengthen the service invocations as well, but, it measures the QoS properties precisely. The client's QOS WRAPPER can measure the round-trip time and the server's QOS WRAPPER the processing time. The QOS WRAPPER pattern is not able to measure network-specific performance-related QoS properties. But, separate tools, such as packet sniffers, can be utilized to measure network-specific performance-related QoS properties.

The client-side QOS WRAPPER can be implemented following the CLIENT PROXY pattern [Buschmann et al. 2007], whereas the server-side QOS WRAPPER can be implemented following the INVOKER [Voelter et al. 2004] pattern.

Known Uses:

—Afek et al. [Afek et al. 1996] 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.

—Loyall et al. [Loyall et al. 1998] introduce Quality Objects (QuO) that are responsible for checking contractually agreed QoS properties. A QuO is a QOS WRAPPER because it is located at the client's or service's machine, but, is separated from the client's or service's implementation.

—Wohlstadter et al. [Wohlstadter et al. 2004] build a QOS WRAPPER that wraps the Apache Axis middleware for measuring QoS.

—The Application Resource Measurement (ARM) API [SAS] is a QOS WRAPPER to measure performance-related QoS properties.

—Rosenberg et al. [Rosenberg et al. 2006] utilize a QOS WRAPPER in order to measure the services' performance-related QoS properties.

3.3 Pattern: QOS INTERCEPTOR

Clients and services must be instrumented to performance-related QoS properties with a reusable, precise, from the implementation separated solution. Access to the middleware is provided to measure negotiable and network-specific performance-related QoS properties in order to detect, for example, bottle-necks in long-running service invocations.

❖ ❖ ❖

How can the middleware be instrumented to measure performance-related QoS properties of service invocations?

In service-oriented systems, the client's middleware transmits the invocation data of the service request to the service over a network. The service's middleware is responsible for receiving incoming data and to transmit the service's response to the client. The client's middleware processes the incoming data and forwards it to the client's application. Let's consider that access to clients' and the services' underlying middleware is provided. The middleware must be instrumented to measure negotiable and network-specific performance-related QoS properties, providing reusability, precise QoS measurements, and separation of concerns.

Therefore,
Hook QOS INTERCEPTORS into the middleware that intercept the message flow between the client and the service. Let the QOS INTERCEPTORS measure the performance-related QoS properties of service invocations.

Figure 7 demonstrates the QOS INTERCEPTOR pattern that utilizes the INVOCATION INTERCEPTOR pattern [Voelter et al. 2004]. A QOS INTERCEPTOR can be utilized on the client- and the server-side. Multiple QOS INTERCEPTORS can be placed in the invocation path, where each of them is responsible to measure different performance-related QoS properties, making it is possible to find, for example, bottle-necks of long-running service invocations.

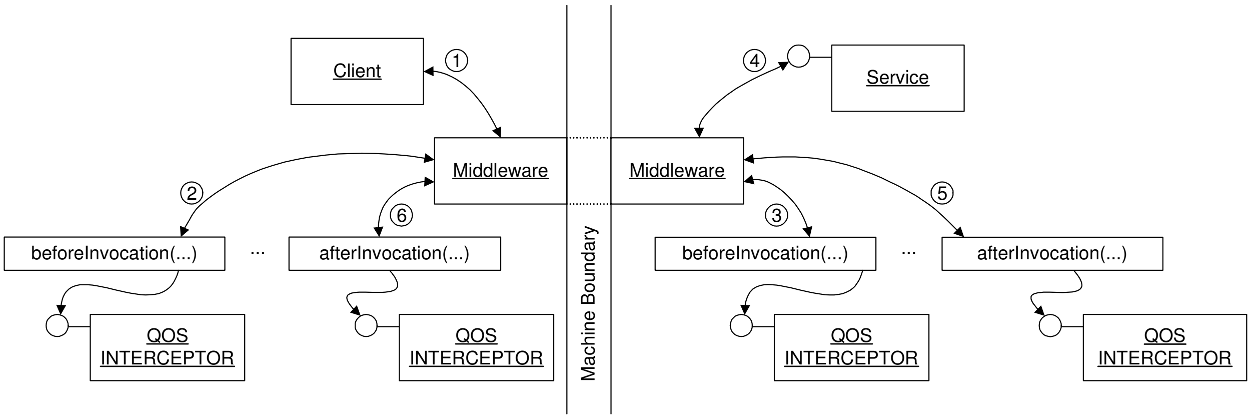


Fig. 7. The QOS INTERCEPTOR pattern

Many middleware frameworks, such as Apache CXF [The Apache Software Foundation c] or .NET Remoting provide possibilities in the middleware to attach a QOS INTERCEPTOR into the invocation path dynamically, using

APIs or configuration files. But, the complexity of the middleware's implementation increases by providing hooks or interfaces to attach and change QOS INTERCEPTORS in the invocation path dynamically.

A QOS INTERCEPTOR delivers precise measurements of network-specific 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. A QOS INTERCEPTOR does not provide performance overhead in the overall system because it measures the performance-related QoS properties locally.

❖ ❖ ❖

A QOS INTERCEPTOR does not measure the elapsed time of transferring the invocation data from the application layer to the middleware as well as the required time of transferring the invocation data from the middleware to the application layer. Hence, the measurements of negotiable performance-related QoS properties are slightly different in comparison to using the QOS INLINE pattern. Placing multiple QOS INTERCEPTORS into the invocation path can impact the preciseness of the QoS measurements. For example at the server-side, a QOS INTERCEPTOR that measures the processing time can influence the measured round-trip time at the client (see Figure 3).

Known Uses:

—The OpenORB project [The Community OpenORB Project] supports QOS INTERCEPTORS.

—The .NET Remoting framework [Microsoft] introduces the *RealProxy*, that is a QOS INTERCEPTOR to intercept remote object invocations.

—The Apache web service frameworks Axis [The Apache Software Foundation a], Axis2 [The Apache Software Foundation b], and Apache CXF [The Apache Software Foundation c] provide the features to use QOS INTERCEPTORS to intercept the messages exchanged between clients and services for measuring performance-related QoS properties.

—The QoS CORBA Component Model (QOSCCM) [Object Management Group (OMG 2008] uses the QOS INTERCEPTOR pattern to easily adapt an application for measuring performance-related QoS properties.

—The VRESCo runtime environment [Michlmayr et al. 2010] measures the performance-related QoS properties using QOS INTERCEPTORS.

—Li et al. [Li et al. 2006] use QOS INTERCEPTORS to intercept the messages between clients and services for measuring the performance-related QoS properties.

3.4 Pattern: QOS REMOTE PROXY

In service-oriented systems, service consumers and service providers have separated networks. Performance-related QoS measurements must be measured uniformly for each client and service.

❖ ❖ ❖

How to introduce a reusable, from the clients' and services' implementation separated, and uniform measuring solution to measure performance-related QoS properties within the service consumer's and service provider's networks?

Typically, the services are not located in the same network as the clients. A client must invoke the service via a wide area network. Service consumers and service providers want to introduce a measuring infrastructure within their own local area networks (LAN). The desired measuring solution should be uniform for each client and service, to enhance the deployment of new QoS-aware clients or services.

Therefore,

Implement and setup a QOS REMOTE PROXY in the service consumer's or service provider's network. In the service consumer's network, let each client invoke the services via the QOS REMOTE PROXY. In the service provider's network, make each service only accessible via a QOS REMOTE PROXY.

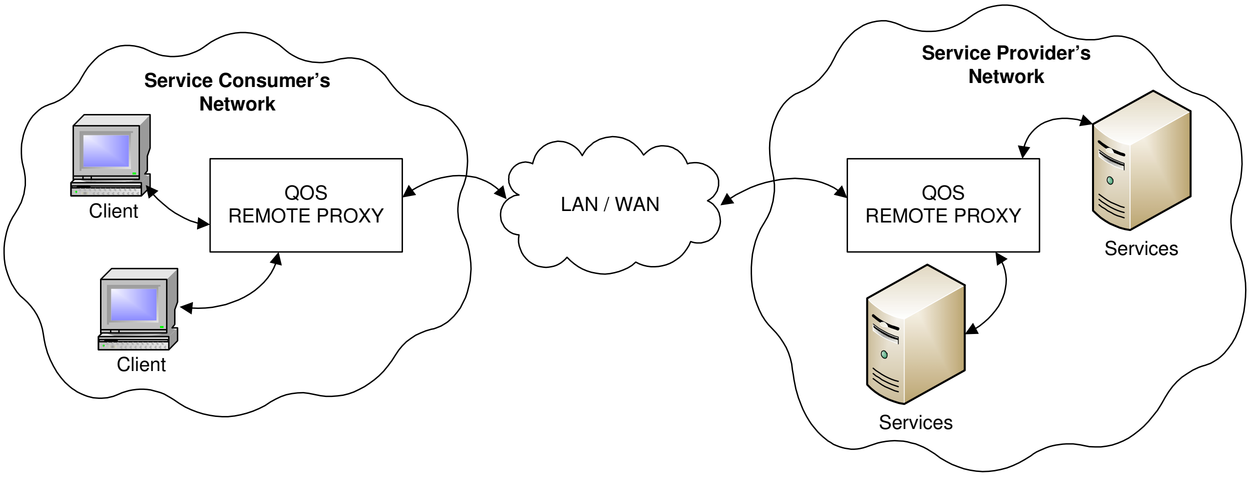


Fig. 8. The QOS REMOTE PROXY pattern

Figure 8 shows an typical service-oriented infrastructure where the clients and the services are not located in the same network. The service consumer's QOS REMOTE PROXY receives the client's request, performs the required QoS measurements, and forwards the request to the service. A QOS REMOTE PROXY within the service provider's network receives the client's requests (directly or via the service consumer's QOS REMOTE PROXY), performs the required QoS measurements, and forwards the request to the appropriate service. After the service has processed the request, it sends the response back to the QOS REMOTE PROXY that measures the required performance-related QoS properties and forwards the response to the client. The service consumer's QOS REMOTE PROXY receives the response (from the service directly or from the service provider's QOS REMOTE PROXY), measures performance-related QoS properties, and forwards the response to the appropriate client.

A QOS REMOTE PROXY provides a separation of concerns because the measuring of the performance-related QoS properties is separated from the clients' and services' implemenation. 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 service via the QOS REMOTE PROXY. It is also possible to configure each service that it can only be invoked via a 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 invoking the service directly. The measurements of the performance-related QoS properties at the QOS REMOTE PROXY differ from the client's and service's local QoS measurements, such as by using aforementioned patterns.

In case of having a multitude of clients, a QOS REMOTE PROXY within the service consumer's network can impact the system's performance because each client has to invoke the service via the QOS REMOTE PROXY. On the server-side, a QOS REMOTE PROXY can affect the performance of the system in case of lots of parallel incoming requests. To decrease the performance overhead, a QOS REMOTE PROXY inside the service provider's network can be implemented as a load-balancer [Buschmann et al. 2007]. A QOS REMOTE PROXY can also act as a gateway, reverse proxy, dispatcher, or a firewall [Buschmann et al. 2007].

Known Uses:

—Wang et al. [Wang et al. 2005] 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 [Object Management Group (OMG) 2008] introduce the VisiBroker [Borland 2009] environment that uses the QOS REMOTE PROXY pattern for measuring the performance-related QoS properties.

—The Apache TCPMon [The Apache Software Foundation d] 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.

—Sahai et al. [Sahai et al. 2002] introduce a QOS REMOTE PROXY for monitoring SLAs in web service-oriented distributed systems.

—Badidi et al. [Badidi et al. 2006] present WS-QoSM, a QoS monitoring solution that measures the QoS properties following the QOS REMOTE PROXY pattern.

—The Cisco IOS IP SLA [Cisco 2011] follows the QOS PROXY pattern that has the responsibility of measuring the performance-related QoS properties.

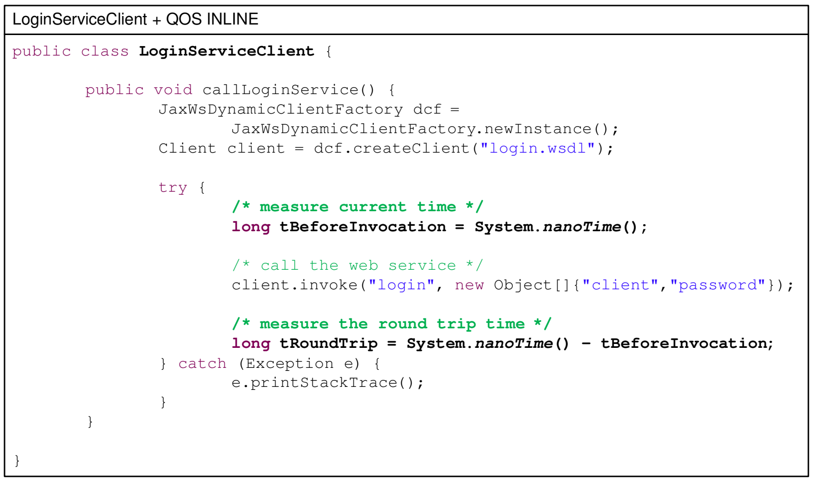
4. SELECTED EXAMPLE: MEASURING PERFORMANCE-RELATED QOS PROPERTIES OF A WEB SERVICE

This section exemplifies the presented patterns for measuring performance-related QoS properties within a service-oriented system. We exemplify the patterns on a web service that offers the functionality to login into a remote system. The service's `login` operation receives a username and a password from the client and checks if the client is authorized to enter. We have implemented the clients and services using the Apache CXF web service framework [The Apache Software Foundation c]. In the following, we present the client-side implementation of the presented patterns.

4.1 Pattern: QOS INLINE

Figure 9 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.

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 [Gamma et al. 1995]. 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.



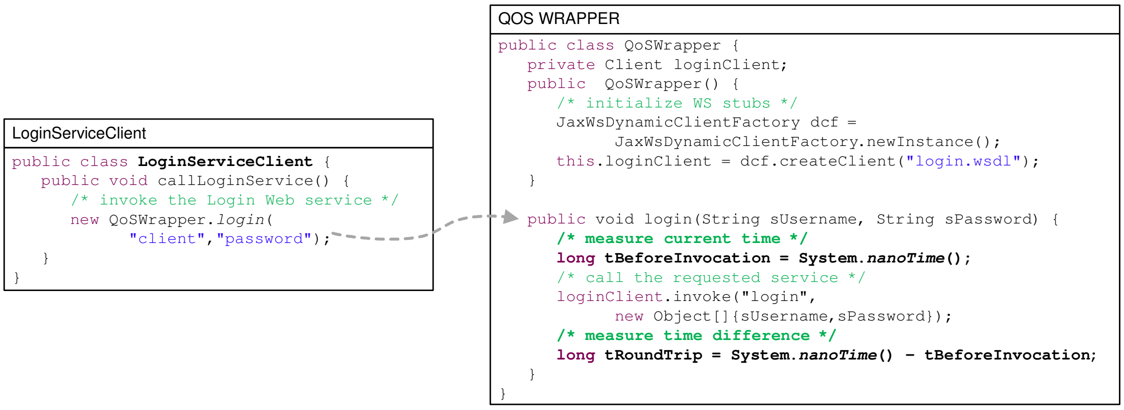
LoginServiceClient + QOS INLINE

```
public class LoginServiceClient {

	public void callLoginService() {
		JaxWsDynamicClientFactory dcf =
			JaxWsDynamicClientFactory.newInstance();
		Client client = dcf.createClient("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();
		}
	}

}
```

Fig. 9. Measuring the round-trip time following the QOS INLINE pattern

4.2 Pattern: QOS WRAPPER

In Figure 10 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.



LoginServiceClient

```
public class LoginServiceClient {
   public void callLoginService() {
      /* invoke the Login Web service */
      new QoSWrapper.login(
            "client","password");
   }
}
```

QOS WRAPPER

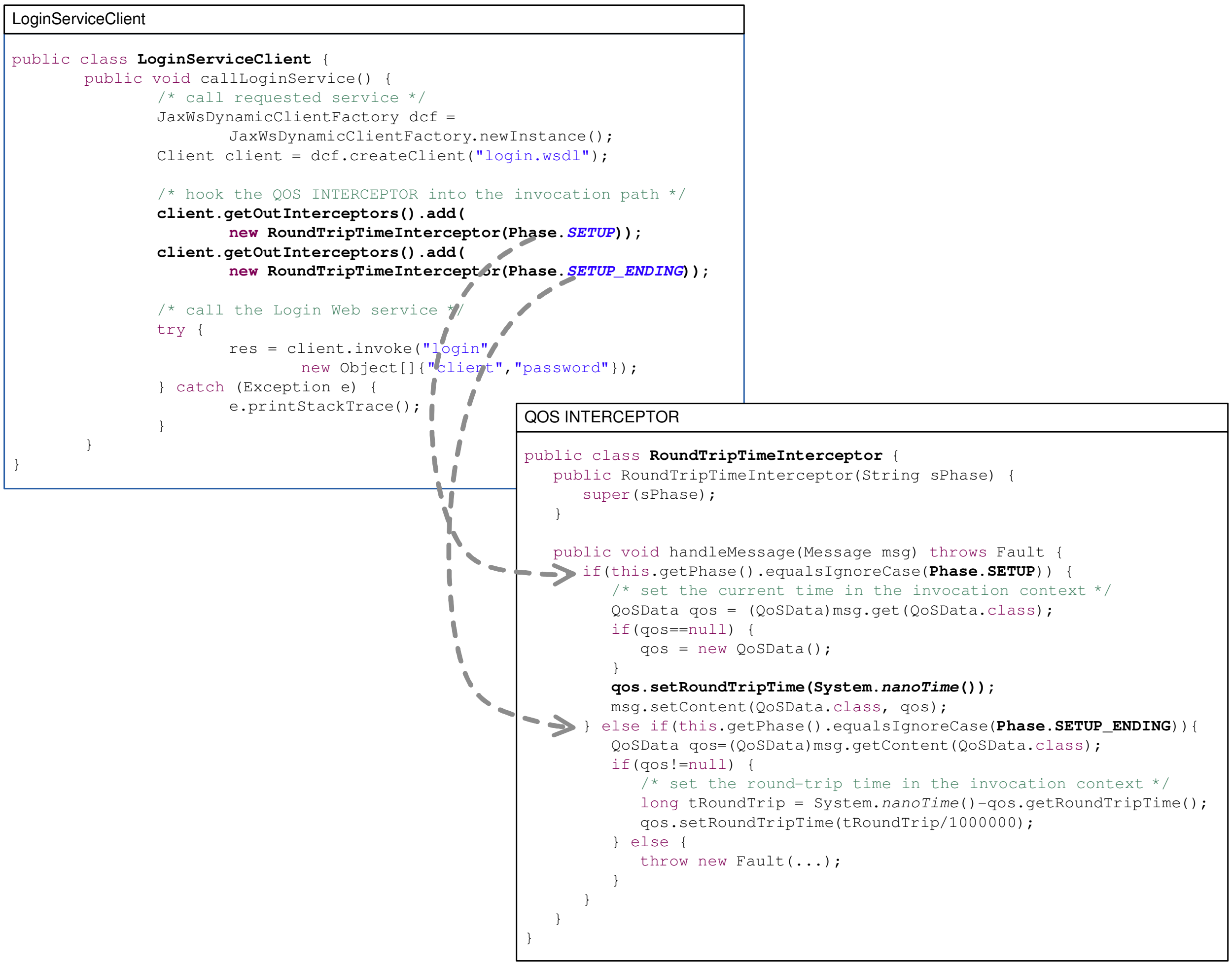
```
public class QoSWrapper {
   private Client loginClient;
   public  QoSWrapper() {
      /* initialize WS stubs */
      JaxWsDynamicClientFactory dcf =
            JaxWsDynamicClientFactory.newInstance();
      this.loginClient = dcf.createClient("login.wsdl");
   }

   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;
   }
}
```

Fig. 10. Measuring the round-trip time following the QOS WRAPPER pattern

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.

4.3 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.



LoginServiceClient

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

                /* hook the QOS INTERCEPTOR into 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();
                }
        }
}
```

QOS INTERCEPTOR

```
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(...);
         }
      }
   }
}
```

Fig. 11. Measuring the round-trip time following the QOS INTERCEPTOR pattern

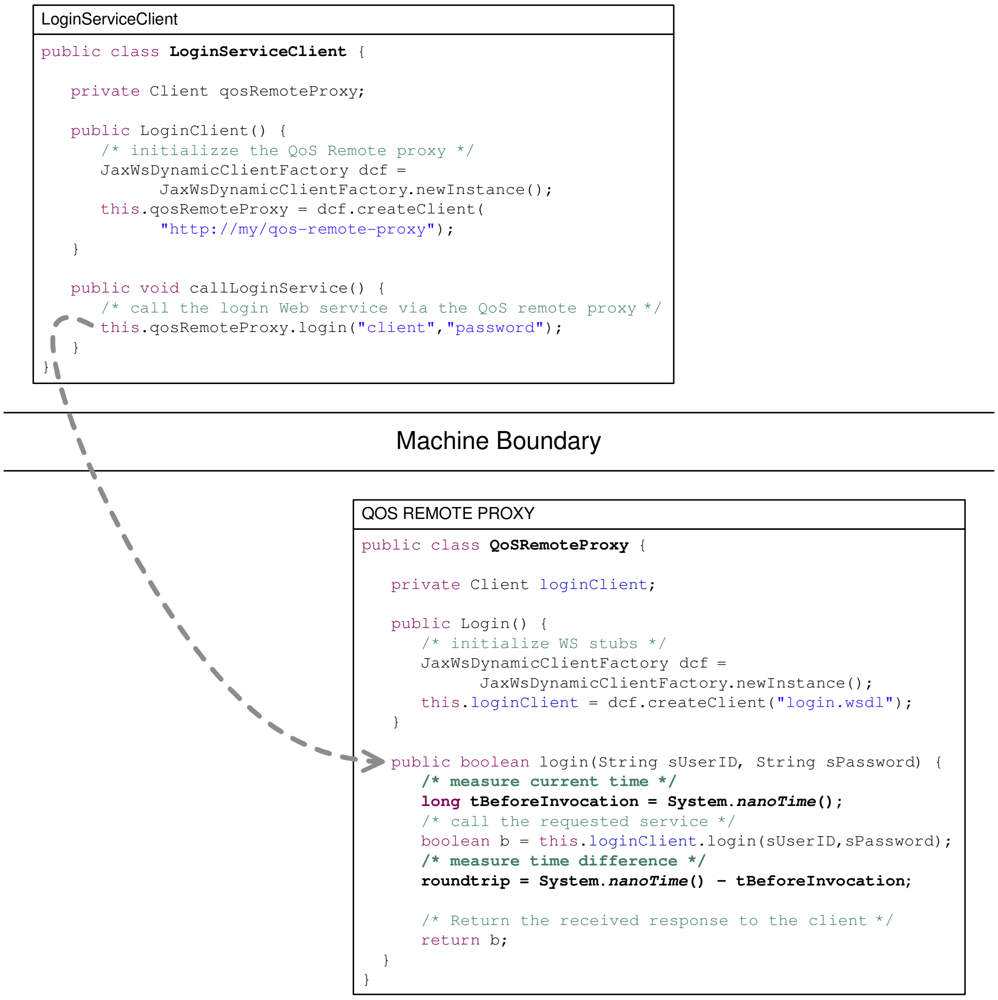
Figure 11 shows an excerpt of the client's implementation and the implemented QOS INTERCEPTOR for measuring the round-trip time of a web service invocation. First, the client initializes the generated stubs of the web service, creates objects of the interceptors, and defines where to place them into 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.

4.4 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 LoginServiceClient {

   private Client qosRemoteProxy;

   public LoginClient() {
      /* initializze the QoS Remote proxy */
      JaxWsDynamicClientFactory dcf =
            JaxWsDynamicClientFactory.newInstance();
      this.qosRemoteProxy = dcf.createClient(
            "http://my/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("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;
   }
}
```

Fig. 12. Measuring the round-trip time following the QOS REMOTE PROXY pattern

We illustrate our Apache CXF implementation of a QOS REMOTE PROXY in Figure 12. 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

In this section we discuss possible ways to implement the patterns and lists possibilities of future work to store and evaluate the QoS measurements.

5.1 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 [Kiczales et al. 1997].

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.

5.2 Model-driven Development (MDD) of the Patterns

Using model-driven development (MDD) makes it possible to generate schematic recurring code automatically [Stahl et al. 2006; Schmidt 2006; Kelly and Tolvanen 2008]. In this section we discuss the automatic generation of the presented patterns.

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.

5.3 Storing and Evaluating the QoS Measurements

The presented patterns cover the aspects of measuring performance-related QoS properties in service-oriented 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 [Hohpe and Woolf 2003]. 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 [Voelter et al. 2004] to store a service invocation's performance-related QoS measurements. The available information of the INVOCATION CONTEXT depends on the interceptor's location in the invocation path. To overcome this problem, the QOS

INTERCEPTOR submits the INVOCATION CONTEXT to some centralized QoS monitor that is responsible for storing and evaluating the performance-related QoS measurements.

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. Many research challenge exist to implement systems that react to the warnings and avoid violations automatically.

Some QoS monitoring infrastructures utilize an event-driven architecture to monitor performance-related QoS agreements, such as [Michlmayr et al. 2010; Zeng et al. 2008; Tran et al. 2010]. In an event-driven architecture, a complex event processing (CEP) engine receives events from the clients and services. The CEP evaluates the received QoS events during the runtime against pre-defined QoS rules. For example, if a client sends an event when invoking a service and the client does not send a second correlated event, then the CEP knows that the service is not available or does not respond in the agreed time frame.

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 [Gamma et al. 1995], the Pattern-Oriented Software Architecture (POSA) series [Buschmann et al. 1996; Schmidt et al. 2000; Buschmann et al. 2007], and in the Remoting Patterns book [Voelter et al. 2004]. 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 patterns are summarized briefly in the Appendix A.

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.

Acknowledgments

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. We also want to thank the participants of the writers' workshop at the PLoP conference, Robert Hanmer, Eunsuk Kang, Matthew Hansen, Kiran Kumar, Cédric Bouhours, and Shivanshu Singh. All of them gave great feedback to improve the quality of the patterns and the paper.

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

REFERENCES

AFEK, Y., MERRITT, M., AND STUPP, G. 1996. Remote Object Oriented Programming with Quality of Service or Java's RMI over ATM.

AURRECOECHEA, C., CAMPBELL, A. T., AND HAUW, L. 1998. A survey of QoS architectures. *Multimedia Systems 6*, 138–151.

BADIDI, E., ESMAHI, L., SERHANI, M. A., AND ELKOUTBI, M. 2006. WS-QoSM: A Broker-based Architecture for Web Services QoS Management. In *Innovations in Information Technology, 2006*. 1–5.

BORLAND. 2009. VisiBroker – A Robust CORBA Environment for Distributed Processing.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley.

CISCO. 2011. Cisco IOS IP Service Level Agreements (SLAs). `http://www.cisco.com/go/ipsla` *(last accessed: February 2011)*.

FRØLUND, S. AND KOISTINEN, J. 1998. Quality of Service Specification in Distributed Object Systems Design. In *COOTS*. USENIX, 1–18.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

HAUCK, R. AND REISER, H. 2000. Monitoring Quality of Service across Organizational Boundaries. In *Proceedings of the Third International IFIP/GI Working Conference on Trends in Distributed Systems: Towards a Universal Service Market*. Springer-Verlag, London, UK, 124–137.

HOHPE, G. AND WOOLF, B. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

JIE JIN, L., MACHIRAJU, V., AND SAHAI, A. 2002. Analysis on Service Level Agreement of Web Services. Tech. rep., HP Laboratories.

KELLER, A. AND LUDWIG, H. 2003. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management 11*, 57–81. 10.1023/A:1022445108617.

KELLY, S. AND TOLVANEN, J.-P. 2008. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect–Oriented Programming. In *ECOOP*. 220–242.

LAMANNA, D. D., SKENE, J., AND EMMERICH, W. 2003. SLAng: A Language for Defining Service Level Agreements. In *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*. FTDCS '03. IEEE Computer Society, Washington, DC, USA, 100–.

LI, Z., JIN, Y., AND HAN, J. 2006. A Runtime Monitoring and Validation Framework for Web Service Interactions. In *Proceedings of the Australian Software Engineering Conference*. IEEE Computer Society, Washington, DC, USA, 70–79.

LOYALL, J., SCHANTZ, R., ZINKY, J., AND BAKKEN, D. 1998. Specifying and Measuring Quality of Service in Distributed Object Systems. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on 0*.

MANI, A. AND NAGARAJAN, A. 2002. Understanding quality of service for Web services – Improving the performance of your Web services. `http://www.ibm.com/developerworks/library/ws-quality.html`, *last accessed: February 2011*.

MICHLMAYR, A., ROSENBERG, F., LEITNER, P., AND DUSTDAR, S. 2010. End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo. *IEEE Transactions on Services Computing 3*, 193–205.

MICROSOFT. .NET Remoting. `http://msdn.microsoft.com/en-us/library/kwdt6w2k(v=vs.71).aspx`.

OBERORTNER, E., ZDUN, U., AND DUSTDAR, S. 2009. 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*. IEEE Computer Society, Washington, DC, USA, 20–25.

OBJECT MANAGEMENT GROUP (OMG). 2008. Common Object Request Broker Architecture/Internet Inter-ORB Protocol (CORBA/IIOP).

OBJECT MANAGEMENT GROUP (OMG. 2008. Quality Of Service For CCM (QOSCCM).

O'BRIEN, L., MERSON, P., AND BASS, L. 2007a. Quality Attributes for Service-Oriented Architectures. In *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*. IEEE Computer Society, Washington, DC, USA, 3.

O'BRIEN, L., MERSON, P., AND BASS, L. 2007b. Quality Attributes for Service-Oriented Architectures. In *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*. IEEE Computer Society, Washington, DC, USA.

RAN, S. 2003. A Model for Web Services Discovery with QoS. *SIGecom Exch. 4,* 1, 1–10.

ROSENBERG, F., PLATZER, C., AND DUSTDAR, S. 2006. Bootstrapping Performance and Dependability Attributes of Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*. IEEE Computer Society, Washington, DC, USA, 205–212.

SAHAI, A., MACHIRAJU, V., SAYAL, M., JIN, L. J., AND CASATI, F. 2002. Automated SLA Monitoring for Web Services. In *IEEE/IFIP DSOM*. Springer-Verlag, 28–41.

SAHAI, A., SAHAI, A., DURANTE, A., DURANTE, A., MACHIRAJU, V., AND MACHIRAJU, V. 2001. Towards Automated SLA Management for Web Services. Tech. rep., Software Technology Laboratory, HP Laboratories.

SAS. ARM – Application Response Measurement. *(last accessed: February 2011)*.

SCHMIDT, D. C. 2006. Model-Driven Engineering. *IEEE Computer 39,* 2.

SCHMIDT, D. C., ROHNERT, H., STAL, M., AND SCHULTZ, D. 2000. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA.

STAHL, T., VOELTER, M., AND CZARNECKI, K. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

THE APACHE SOFTWARE FOUNDATION. Apache Axis. `http://axis.apache.org/`.

THE APACHE SOFTWARE FOUNDATION. Apache Axis2. `http://ws.apache.org/axis2/`.

THE APACHE SOFTWARE FOUNDATION. Apache CXF. `http://cxf.apache.org/`.

THE APACHE SOFTWARE FOUNDATION. Apache TCPMon. `http://ws.apache.org/commons/tcpmon/`.

THE COMMUNITY OPENORB PROJECT. OpenORB. `http://openorb.sourceforge.net/`.

THE SERVICE LEVEL AGREEMENT ZONE. 2007. SLA Information Zone. `http://www.sla-zone.co.uk/` *(last accessed: 09/2010)*.

TRAN, H., HOLMES, T., OBERORTNER, E., MULO, E., CAVALCANTE, A. B., SERAFINSKI, J., TLUCZEK, M., BIRUKOU, A., DANIEL, F., SILVEIRA, P., ZDUN, U., AND DUSTDAR, S. 2010. An End-to-End Framework for Business Compliance in Process-Driven SOAs. In *Proceedings of SYNASC*.

VOELTER, M., KIRCHER, M., AND ZDUN, U. 2004. *Remoting Patterns – Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware.* Wiley & Sons.

WANG, Q., YE, Q., AND CHENG, L. 2005. 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*. IEEE Computer Society, Washington, DC, USA, 819–824.

WOHLSTADTER, E., TAI, S., MIKALSEN, T., ROUVELLOU, I., AND DEVANBU, P. 2004. GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions. In *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. IEEE Computer Society, Washington, DC, USA, 189–199.

YU, W. D., RADHAKRISHNA, R. B., PINGALI, S., AND KOLLURI, V. 2007. Modeling the Measurements of QoS Requirements in Web Service Systems. *Simulation 83,* 1, 75–91.

ZENG, L., LINGENFELDER, C., LEI, H., AND CHANG, H. 2008. Event-Driven Quality of Service Prediction. In *Proceedings of the 6th International Conference on Service-Oriented Computing*. ICSOC '08. Springer-Verlag, Berlin, Heidelberg, 147–161.

A. SUMMARY OF THE PATTERNS

Requirements:	- Minimal performance overhead - Preciseness - Reusability - Separation of concerns
Pattern:	QOS INLINE
Description:	Instrument the client's and the service's implementation with local measuring points by placing them directly into their implementation.
Forces:	- minimal performance overhead - precise QoS measurements - does not influence other measurements
Consequences:	- no separation of concerns - not reusable
Known Uses:	- Mani and Nagarajan [Mani and Nagarajan 2002]
Pattern:	QOS WRAPPER
Description:	Instrument the client's and service's implementations with local QOS WRAPPERS that are responsible for measuring the performance-related QoS properties. Let a client invoke a service using a client-side QOS WRAPPER. Extend a service with a server-side QOS WRAPPER that receives the client's requests.
Forces:	- minimal performance overhead - separation of concern - reusable - precise QoS measurements
Consequences:	- can not measure network-specific QoS properties
Known Uses:	- Afek et al. [Afek et al. 1996] - Quality Objects (QuO) [Loyall et al. 1998] - Wohlstadter et al. [Wohlstadter et al. 2004] - The Application Resource Measurement (ARM) API [SAS] - Rosenberg et al. [Rosenberg et al. 2006]
Solution	Pattern: QOS INTERCEPTOR
Description:	Hook QOS INTERCEPTORS into the middleware that intercept the message flow between the client and the service. Let the QOS INTERCEPTORS measure the performance-related QoS properties of service invocations.
Forces:	- minimal performance overhead - separation of concerns - reusable
Consequences:	- access to middleware necessary - can influences other measurements, hence - can lead to not precise QoS measurements
Known Uses:	- The OpenORB project [The Community OpenORB Project] - .NET Remoting [Microsoft] - Axis [The Apache Software Foundation a], Axis2 [The Apache Software Foundation b], and Apache CXF [The Apache Software Foundation c] - QoS CORBA Component Model (QOSCCM) [Object Management Group (OMG 2008] - The VRESCo runtime environment [Michlmayr et al. 2010] - Li et al. [Li et al. 2006]
Solution	Pattern: QOS REMOTE PROXY
Description:	Implement and setup a QOS REMOTE PROXY in the service consumer's or service provider's network. In the service consumer's network, let each client invoke the services via the QOS REMOTE PROXY. In the service provider's network, make each service only accessible via a QOS REMOTE PROXY.
Forces:	- separation of concerns - reusable - can be implemented utilizing the QOS INLINE, QOS WRAPPER, or QOS INTERCEPTOR pattern
Consequences:	- can have performance overhead - can influence other measurements - can lead to imprecise QoS measurements
Known Uses:	- The QoS-Adaptation proxy [Wang et al. 2005] - The VisiBroker [Borland 2009] environment of the Corba IIOP specifications [Object Management Group (OMG) 2008]. - The Apache TCPMon [The Apache Software Foundation d] tool - Sahai et al. [Sahai et al. 2002] - WS-QoSM [Badidi et al. 2006] - The Cisco IOS IP SLA [Cisco 2011]

Table I. : Patterns to measure performance-related QoS properties