Available online at www.sciencedirect.com

ScienceDirect

Performance Evaluation 64 (2007) 646–663

PERFORMANCE EVALUATION
An International Journal

www.elsevier.com/locate/peva

QoS management in service-oriented architectures

Daniel A. Menascé[a,*], Honglei Ruan[a], Hassan Gomaa[b]

[a] *Department of Computer Science, George Mason University, Fairfax, VA 22030, USA*
[b] *Department of Information and Software Engineering, George Mason University, Fairfax, VA 22030, USA*

Received 15 May 2006; accepted 1 October 2006
Available online 17 November 2006

Abstract

The next generation of software systems will be highly distributed, component-based and service-oriented. They will need to operate in unattended mode and possibly in hostile environments, will be composed of a large number of 'replaceable' components discoverable at run-time, and will have to run on a multitude of unknown and heterogeneous hardware and network platforms. This paper focuses on QoS management in service-oriented architectures in which service providers (SP) provide a set of interrelated services to service consumers, and a QoS broker mediates QoS negotiations between SPs and consumers. The main contributions of this paper are: (i) the description of an architecture that includes a QoS broker and service provider software components, (ii) the specification of a secure protocol for QoS negotiation with the support of a QoS broker, (iii) the specification of an admission control mechanism used by SPs, (iv) a report on the implementation of the QoS broker and SPs, and (v) the experimental validation of the ideas presented in the paper.
© 2007 Elsevier B.V. All rights reserved.

Keywords: QoS; Service oriented architectures; Performance; QoS broker

1. Introduction

The next generation of complex software systems will be highly distributed, component-based, service-oriented, will need to operate in unattended mode and possibly in hostile environments, will be composed of a large number of 'replaceable' components discoverable at run-time, and will have to run on a multitude of unknown and heterogeneous hardware and network platforms. Three major requirements for such systems are performance, availability and security. Performance requirements imply that such systems must be adaptable and self-configurable to changes in workload intensity. Availability and security requirements suggest that these systems have to adapt and reconfigure themselves to withstand attacks and failures. In this paper, we concentrate on QoS requirements for performance (e.g. response time and throughput) and some related security considerations.

Quality of service (QoS) is an important concern in dynamic service composition and selection given that many service providers provide similar services with common functionality but different QoS and cost. It is necessary to provide a negotiation mechanism between clients (or applications) and service providers to reach mutually-agreed

* Corresponding address: Department of Computer Science, The Volgenau School of Information Technology and Engineering, George Mason University, Room 160, MS 5C8, 4400 University Drive, Fairfax, VA 22030, USA. Tel.: +1 703 993 1537.
E-mail addresses: menasce@cs.gmu.edu (D.A. Menascé), hruan@gmu.edu (H. Ruan), hgomaa@ise.gmu.edu (H. Gomaa).

0166-5316/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2006.10.001

QoS goals. It is also critical for service providers to be able to guarantee QoS agreements at runtime. Consider, as an example, that several stockbrokers provide stock quote services with different response times and costs. A customer can choose a stockbroker who satisfies QoS requirements within the budget by negotiating with each stockbroker. If a stockbroker cannot satisfy the QoS requirements of the customer, the stockbroker can make a counter offer with lower cost and/or higher response times. Alternatively, the client can turn to another broker to have its needs satisfied. Once a stockbroker has been selected, the broker must guarantee the response time of the service at runtime. Another example is that of an online travel agent that uses airline reservation, hotel reservation, car rental reservation and payment processing services provided by a multitude of third-party service providers. The response time and throughput characteristics of the travel agency depend on the QoS of these external service providers. Therefore, QoS negotiation and adherence to negotiated QoS goals is of utmost importance to the adequate performance of the travel agency.

We consider component-based QoS management in service-oriented architectures (SOA) in which two types of components are considered: QoS broker component (QB) and service provider component (SP). The QoS broker negotiates on behalf of the SPs. Therefore, a QoS broker must have the ability to manage and reserve the resources of SP components so that proper Quality of Service (QoS) guarantees at the SPs can be honoured. A SP component provides services to other components (or clients) and implements admission control in order to be compliant with already committed QoS guarantees. The requirements of the solution presented in this paper are: (i) QoS-based service registration ability: a QB should allow service providers to register the performance characteristics for their services; (ii) QoS-based service discovery ability: a QB should be able to find required services on the basis of their functionality and performance; (iii) Negotiation capability: the QB should be able to carry out QoS negotiations on behalf of multiple SPs residing on the same or different hardware platforms; (iv) Resource reservation ability: once a QoS agreement is reached between the client and the QB on behalf of a SP, the QB should reserve resources in the proper SP to honour the committed requests; (v) Limited SP involvement in the QoS negotiation: the communication between client and SPs should be kept to a minimum during QoS negotiation; (vi) Certifiable negotiation: a QB should be able to provide a client with an unforgeable certificate of negotiation. This certificate should be presented by the client to the SP to prove that it has obtained the QoS guarantees from the QB.

A SOA provides an effective approach to building component-based distributed applications. The implementation details of components in a SOA are invisible to service consumers. External components neither know nor care how they perform their function, merely that they return the expected results. A component's service APIs are published and discovered by other components on an as-needed basis. An example of this model is the Web Services paradigm [42], which combines SOAP [39], the Web Services Description Language (WSDL) [41], and the Universal Description, Discovery, and Integration (UDDI) [40]. Discovering services based on their functionality and QoS characteristics is an important consideration in Web Services and UDDI registries [20,34].

In large, dynamic, highly distributed, loosely coupled, component-based systems, it is necessary to use a framework for software system composition and dynamic reconfiguration that can support the evolution of functional and performance requirements as well as environmental changes (e.g. network, hardware and software failures). Software systems in these environments need to be composed of autonomous components that are chosen in such a way that they can satisfy requests with certain QoS requirements. We propose a QoS-based approach to distributed software system composition. Our method uses resource reservation mechanisms at the component level to provide soft (i.e. average values) QoS requirements at the software system level. We do not claim to provide hard real-time QoS guarantees.

Different metrics can be used for measuring and providing a given QoS level, such as response time, throughput and concurrency level. This paper describes a QoS broker and service provider (SP) components, where a client component can request a service from a particular SP with certain levels of QoS requirements. The QoS broker determines whether the SP can provide this level of QoS. In the affirmative case, the required resources of the SP are reserved. Otherwise, the QoS broker negotiates with the client on behalf of the SP for a different QoS level. If, eventually, a mutually agreed QoS level is negotiated, the QoS broker takes this QoS request into account when it negotiates with other clients on behalf of this SP. In a distributed or Internet-based application, there may exist multiple SPs providing the same functionality with different levels of QoS guarantees. The QoS broker functionality can be extended to determine which SP, among multiple SPs, can best satisfy the QoS request. The main contributions of this paper are: (i) the description of an architecture that includes a QoS broker and service provider software components, (ii) the specification of a secure protocol for QoS negotiation with the support of a QoS broker, (iii) the specification of an admission control mechanism used by SPs, (iv) a report on the implementation of the QoS broker and SPs, and (v) the experimental validation of the concepts presented in the paper. The description provided here deals with a single QoS broker serving several SPs. However, it is easy to extend the architecture to a set of QoS brokers that manage replicated entries in order to improve reliability and performance as explained in Section 7.

The rest of this paper is organized as follows: Section 2 discusses work related to this paper. Section 3 presents the general architecture of the QoS broker. Section 4 describes the QoS negotiation protocol used by the QoS broker to establish QoS compliant sessions on behalf of a SP. Section 5 describes how QoS evaluation is done. Section 6 describes how a performance model is dynamically built by the QoS broker to evaluate requests for session establishment. Section 7 discusses fault tolerance and performance issues related to the QoS broker approach. Section 8 describes the results of experiments carried out to validate the ideas presented in the paper. Finally, Section 9 presents some concluding remarks and future work.

2. Related work

Performance is an important consideration when designing the software architecture [10,11] of a complex distributed software system. The ability to estimate the future performance of a large and complex distributed software system at design time, and iteratively refine these estimates at development time, can significantly reduce overall software cost and risk. Analyzing the performance of a software architecture affords a quantifiable tradeoff analysis with the objective of minimizing risks in software designs. Approaches based on analytical performance models of software architectures are described in [12,27–30].

The issue of application level QoS has received increased attention recently [43]. Queuing network models have been widely used for predicting the performance of software systems. Kraiss et al. [15] used an M/G/1 queuing model to ensure response time guarantees for message-oriented middleware (MOM) systems. Petriu et al. [32] showed how Layered Queuing Network (LQN) models can be used to predict CORBA-based middleware performance. The TAPAS [5] project demonstrated how a Queuing Network (QN) model can be used as a formal analytical model, translated from a UML design diagram, to predict the performance of a system during the design phase. In our work, a two-level multiclass closed QN model was used to evaluate whether a service provider component is eligible to be added into an application with certain QoS guarantees.

Ran [34] presents a model for QoS support in Web services with an extended UDDI registry. This model does not address, as we do, the fact that QoS properties, such as response time and throughput, vary dynamically with workload intensity levels. No report on an implementation and validation of the approach in [34] has been given. Tian et al. [37] made an effort to tackle the QoS gap between the Web Service layer and the network layer by extending WSDL files with QoS information. Both studies did not address how a service provider guarantees its QoS claims. In the Web services domain, dynamically selecting a service provider that best meets consumer's needs is a challenge. Maximilien et al. [17] addressed this problem using a Web Services Agent Framework (WSAF) and ontologies. In WSAF, an agent, associated with a service interface, selects an implementation that best meets a consumer's QoS preferences by querying participating agencies which hold QoS data. Since WSAF does not provide the ability to reserve the resources required for the selected level of QoS, the client is not guaranteed to receive the promised level of QoS. This is an important requirement. WSAF did not address security issues in the communication between agents and agencies. This requirement is also important, since agents and agencies usually communicate QoS data over insecure networks.

QoS in Grid computing was studied in GARA [6,7], and G-QoSM [1]. In GARA [6,7], resource reservation and actual allocation are separated in order to support advanced resource reservation for critical requests. G-QoSM [1] extended a UDDI registry. A QoS broker is introduced and is responsible for QoS negotiation and adaptation. But the QoS broker lacks resource reservation ability.

Several studies on providing QoS support in middleware have been conducted. Tien et al. [38] described a model in which QoS negotiation proceeds by propagating a QoS contract among involved resource managers. Unlike our approach, the work described in [38] does not provide for counter offers to be sent to service consumers. The approach described here reserves resources on a temporary basis while a consumer decides whether to accept or reject a counter offer. Quartz [36] needs a large set of QoS parameters in order to implement portability between different application areas. CQoS [14] provides QoS support by breaking the basic behaviours of processing flow of a request/reply into multiple event points, each of which is bounded with one or more handlers. The middleware-dependency of interceptors reduces the portability of CQoS. Pruyne [33] chained different interceptors in tandem to

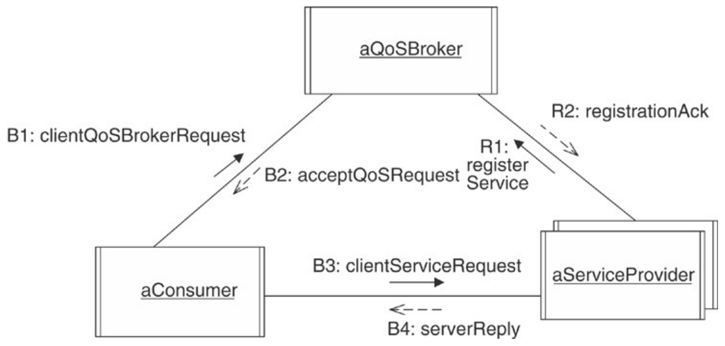


Fig. 1. Service registration, QoS negotiation, and service access.

implement different QoS aspects such as availability and admission control. Even though both of these studies can enhance some QoS properties, such as security and fault tolerance, they are not able to provide any explicit QoS guarantees on metrics such as response time and throughput.

Different QoS negotiation protocols were presented for QoS provision. The NRP presented in Dermler et al. [4] performs negotiation based on client-specified QoS value ranges and given resource availability on end-systems and communication links. Koistinen [8] described a multicategory QoS negotiation protocol that allows clients to compute the aggregate worth for each offer. Cavanaugh [3] developed a negotiation architecture in which a QoS manager detects QoS violations and initiates QoS negotiation by sending expected remedy actions to the resource manager. Our QoS negotiation protocol works by changing the concurrency level of a request until a proper offer is found using analytic queuing network models.

3. Basic QoS broker architecture

A QoS aware application (or simply Q-application), is composed of one or more service provider (SP) components that can be dynamically chosen. A QoS broker (QB) is capable of engaging in QoS negotiations on behalf of various SP components. Therefore, the QoS broker has the ability to manage and reserve the resources of SPs so that proper QoS guarantees can be honored.

As indicated in Fig. 1, SPs must register with a QB before the broker can act on their behalf. During the registration phase, an SP must inform the QB about its service descriptions and the service demand characteristics for each service at each of the devices of the hardware platform where the SP runs. A consumer of a service provided by one of the SPs represented by the QB enters in a QoS negotiation with the QB. This negotiation can result in QoS requests being accepted, rejected, or in counter offers (for degraded QoS levels) being made to the consumer, who can accept or reject the counter offer. Once a negotiation is successful, the QB sends the consumer a secure token that can be used by the consumer to request services from the SP. The QB maintains a table of QoS commitments (ToC) made to previous requests so that it can dynamically evaluate if an incoming QoS request can be granted, rejected, or if a counter offer is feasible.

In the Web Services domain, the QoS broker can be implemented as an extension of a UDDI registry. The difference between the QoS broker and traditional UDDI is that the QoS broker has negotiation and resource reservation capabilities.

A *QoS request* ρ is used to start a session with a SP. A *session* is characterized from the point of view of the session initiator by $N (N \geq 1)$ concurrent threads, all submitting requests to the same service of the SP. There is no limit to the total number of requests that can be submitted by each of the N threads. It is a trivial matter to limit the number of requests in a session or the duration of a session if required by Service Level Agreements (SLAs). However, a thread cannot submit a subsequent request until a reply from the previous one is received. More precisely, $\rho =$ (SessionID, SPID, ServiceID, N_{req}, RMAX, XMIN, ExpireDateTime, K_{pu}^{C}) where SessionID uniquely identifies a session generated by the client, SPID identifies the SP with whom the session is to be established, ServiceID identifies the service of SPID to be requested by the session, N_{req} is the requested concurrency level, i.e. the maximum number of requests concurrently executing the requested service in the session, RMAX is the maximum average request response time required for the session, XMIN is the minimum throughput required for the session, ExpireDateTime

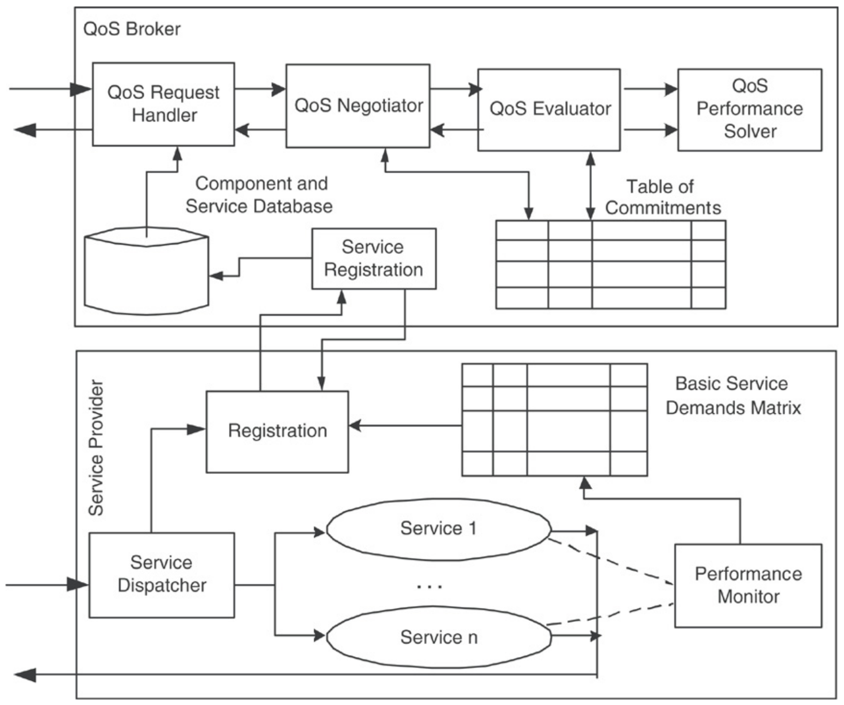


Fig. 2. Architecture of a QoS broker and a SP.

is the expiration date and time of the session, and K_{pu}^{C} is the public key of the requester which will be used by a SP for authentication purposes. Typically, K_{pu}^{c} is delivered as part of a digital certificate signed by a trusted certificate authority.

4. QoS negotiation protocol

We now describe, with the help of Fig. 2, the architecture of a QoS broker component and of a SP component. A SP component (see bottom part of Fig. 2), like any service-oriented component, has a registration module that implements the interaction with the QoS broker for registering its service descriptions and the basic service demand matrix, which holds the service demands of each service at each device of the hardware platform on which it runs. A service dispatcher at the SP receives requests for service and sends the requests to an available thread to execute the service. The service dispatcher implements admission control within an accepted session by not accepting additional requests for that session if the number of concurrently executing requests for the session is equal to its negotiated concurrency level. The dispatcher also has the ability to verify whether a request is legitimate by verifying the attached digital signature of the requester.

A QoS broker (see top part of Fig. 2) consists of a QoS Request Handler, a QoS Negotiator, a QoS Evaluator, and a QoS Performance Solver. The QoS Request Handler implements the QoS Negotiation Protocol with the help of the QoS Negotiator, QoS Evaluator, and Performance Model Solver. A state transition diagram for the QoS Negotiation Protocol as seen by the QoS broker is shown in Fig. 3. The figure shows five states for a session: no session established, pending session, QoS request being evaluated, status of token for counter offer being evaluated, and session established. The notation used to label state transitions is of the type event/action. Events are message arrivals with optional conditions indicated in square brackets or the completion of an internal action. Actions consist of sending messages or starting the execution of internal actions.

Three types of messages can be received from the client by the QoS Requests Handler:

- QoS request: this message is a request to start a session with certain QoS requirements. This request can be accepted, rejected, or a counter offer with a different concurrency level, larger response time, or smaller throughput, can be generated as a result of the request. The decision about accepting, rejecting, or providing a counter offer is

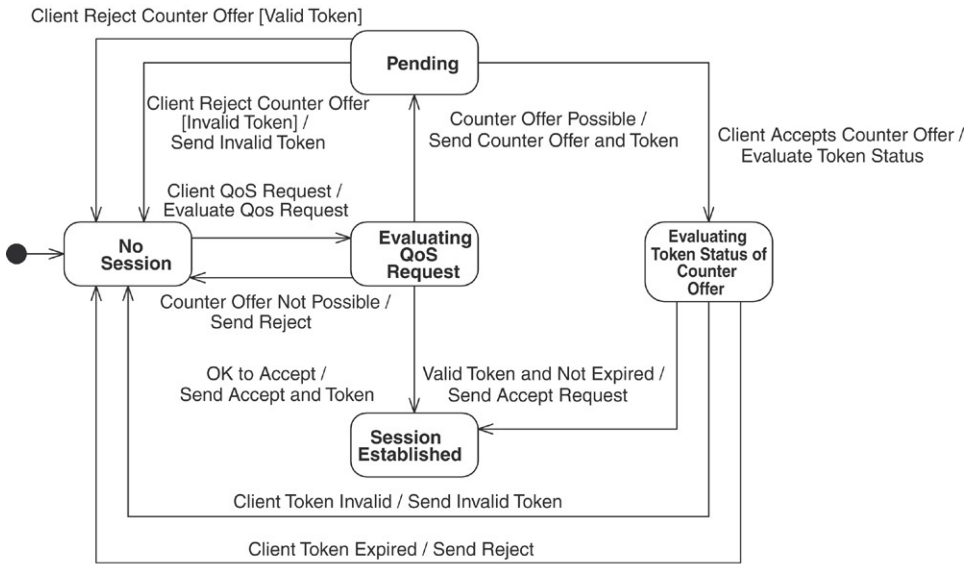

Client Reject Counter Offer [Valid Token]
Pending
Client Reject Counter Offer [Invalid Token] / Send Invalid Token
Counter Offer Possible / Send Counter Offer and Token
Client Accepts Counter Offer / Evaluate Token Status
Client QoS Request / Evaluate Qos Request
No Session
Evaluating QoS Request
Evaluating Token Status of Counter Offer
Counter Offer Not Possible / Send Reject
OK to Accept / Send Accept and Token
Valid Token and Not Expired / Send Accept Request
Session Established
Client Token Invalid / Send Invalid Token
Client Token Expired / Send Reject

Fig. 3. State transition diagram for the QoS negotiation protocol.

taken by the QoS broker, on behalf of a candidate SP, as explained in Section 5. When a QoS request ρ is accepted on behalf of a SP, the QoS broker places the information about the request into a Table of QoS Commitments (ToC), which holds all committed and unexpired sessions. A *token*, digitally signed by the QoS broker, is generated and sent back to the requester with the acceptance message. The token consists of SPID, CommitmentID, SessionID, ServiceID, N_{offer}, ExpireDateTime, and K_{pu}^{C}. SPID consists of the IP address of the selected SP and the port number on which the SP is listening; CommitmentID is the index in the TOC of the committed session; N_{offer} is the maximum concurrency level offered by the SP in order to provide a certain response time and throughput; SessionID, serviceID, ExpireDateTime, and K_{pu}^{C} are as defined in the last paragraph of Section 3. If the QoS broker cannot accept ρ, it may be able to generate a counter offer, which is sent to the client along with a token. The request, modified per the counter offer, is placed in the ToC and flagged as 'temporary' to give the requester a chance to accept or reject the counter offer. Counter offers expire after a certain time period. If the acceptance arrives too late, the corresponding temporary commitment is deleted from the ToC and the requester is notified.

- Accept counter offer: this message is used to indicate to the QoS broker that a counter offer is accepted. In this case, the temporary commitment, if not yet expired, is made permanent.
- Reject counter offer: this message indicates that a counter offer was rejected. The temporary commitment is deleted from the ToC.

Figs. 4 and 5 present the pseudo-code of the QoS Negotiation Protocol. The function QoSFeasibility returns either accept, reject, or counter offer, as explained in Section 5. The notation $S_X[y]$ denotes the digital signature of y signed with X's private key.

A QoS broker may be shared by several SPs and it maintains a ToC in which each entry corresponds to a SPID and a set of related commitments. Each commitment consists of CommitmentID, SessionID, ServiceID, N_{offer}, RMAX, XMIN, and ExpireDateTime. SPID, SessionID, ServiceID, N_{offer}, and ExpireDateTime are as described before.

Fig. 6 illustrates the interactions between a client, a QoS broker, and a SP, using a UML sequence diagram for a session:

(1) The SP registers with the QoS broker its service descriptions and basic service demand matrices (see Section 6) with the QoS broker. The QB confirms with the SP by sending its public key ($K_{\text{pu}}^{\text{QB}}$) to the SP.
(2) The client sends a QoS request $\rho =$ (SessionID, SPID, ServiceID, N_{req}, RMAX, XMIN, ExpireDateTime, K_{pu}^{C}) to the QB.
(3) The QB evaluates/negotiates the QoS request on behalf of the SP. In case of a successful negotiation, the QB generates a token = (SPID, CommitmentID, SessionID, ServiceID, N_{offer}, ExpireDateTime, K_{pu}^{C}), signs the token with its private key, $K_{\text{pr}}^{\text{QB}}$, and then sends the signed token back to the client.

```
Receive QoSRequest (request):
Begin
   SPResult = QoSFeasibility (request);
   If SPResult = Accept
   Then Begin /* accept request */
              Enter QoS Commitment into the ToC;
              Create Token for Request;
              Sign the Token with the QoS Boker's private key;
              Send AcceptRequest (Token, $S_{QB}$[Token]) to client
         End
   Else  If SPResult = CounterOffer
         Then Begin /* send counter offer */
                   Create temporary entry for request in ToC;
                   Create a Token for request;
                   Sign the Token with the QoS broker's private key;
                   Send CounterOffer (Token, $S_{QB}$[Token]) to client
               End
         Else /* SPResult = Reject */
              Send RejectRequest to client
End
```

Fig. 4. QoS negotiation protocol: QoSRequest.

```
Receive AcceptCounterOffer:
Begin
   If Client Token is valid & temporary entry in ToC has not expired
   Then Begin
            Turn temporary entry in ToC into a permanent one;
            Send AcceptRequest to client
        End
   Else  If Client Token is valid & has expired
         Then Begin
                  Delete temporary entry from the ToC;
                  Send RejectRequest to client
              End
         Else Send InvalidToken to client
End

Receive RejectCounterOffer:
Begin
   If Token is valid
   Then Delete temporary entry from the ToC
   Else Send InvalidToken to client
End
```

Fig. 5. QoS negotiation protocol: accept counter offer and reject counter.

(4) The client can then start to send service request messages to the SP identified by SPID. These messages have the format α = (Token, S_{QB}[Token], ServiceRequest, S_C[ServiceRequest]), where ServiceRequest is uniquely

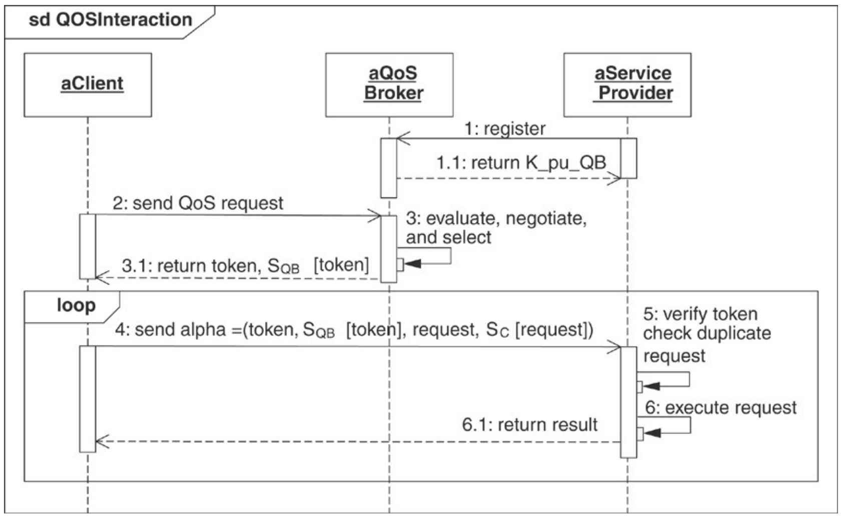

sd QOSInteraction
aClient
aQoS Broker
aService Provider
1: register
1.1: return K_pu_QB
2: send QoS request
3: evaluate, negotiate, and select
3.1: return token, SQB [token]
loop
4: send alpha =(token, SQB [token], request, SC [request])
5: verify token check duplicate request
6: execute request
6.1: return result

Fig. 6. Sequence diagram representation of the interaction between client, QoS broker and service provider.

identified by a request ID in the session. In other words, a service request message is composed of a token signed by the QB and a service request signed by the client. The token is sent with the request to allow the SP to verify the authenticity of the request.

(5) The SP first verifies the QB's signature using $K_{\text{pu}}^{\text{QB}}$ to ensure the token has not been modified and that it was indeed issued by the QB. This can be done because the SP received the QB's public key when it registered with the QB (see step 1). If the signature can be verified, N_{offer} and K_{pu}^{C} are extracted from the token. The SP then verifies the signature S_C[ServiceRequest] using K_{pu}^{C}. The SP also checks whether the current request is a duplicate of a previously received request. This can be done by keeping the latest request ID received from each session. Duplicate requests are discarded to prevent replay attacks.

(6) After successful verifications, if the request has not expired (by comparing the current time with ExpireDateTime extracted from the token), the SP executes the service request and sends the result back to the client (steps 4–6 are repeated for each service request in the session).

The purpose of verifying S_C [ServiceRequest] is to prevent replay attacks, specifically, to prevent a third party from reusing the token. This goal can be achieved because every request in a session has a unique request ID. Therefore, every service request generated by an authorized client has a different signature. Even though a third party may intercept a request, it will not able to generate the same signature as the authorized client without knowing the private key (K_{pr}^{C}) of the client. Even though it may maliciously repeat sending the intercepted message to the SP, any duplicate message will be discarded by the SP. This may lead to denial of service. We do not claim to specifically deal with this issue in our solution. This problem has been addressed by many other researchers and the solutions developed there can certainly be used in our case.

5. QoS request evaluation

Client requests arriving at the QoS broker are evaluated by the QoS evaluator. The evaluation is done with the help of the QoS performance solver, which is described in Section 6. The following notation is used to describe how a QoS request is evaluated.

- ρ: QoS request being evaluated.
- ω: set of all committed QoS requests.
- $\bar{x}$: indicates a violation of a QoS goal associated to x, where x is either a QoS goal, a QoS request, or the set of already committed requests.
- $\hat{x}$: indicates a satisfaction of a QoS goal associated to x, where x is either a QoS goal, a QoS request, or the set of already committed requests.
- $\arg\max_N\{c\}$: the maximum value of N that makes condition c true.

- $\arg\min_N\{c\}$: the minimum value of N that makes condition c true.
- $X(n)$: throughput of the request being evaluated when the concurrency level for that request is n. This throughput is obtained by solving a performance model considering the already commmitted requests.
- $R(n)$: average response time of the request being evaluated when the concurrency level for that request is n. This response time is obtained by solving a performance model considering the already commmitted requests.
- N: concurrency level for request being evaluated.
- N_{req}: requested concurrency level for the request being evaluated.
- N_{off}: offered concurrency level for the request being evaluated.
- R_{off}: offered response time for the request being evaluated.
- X_{off}: offered throughput for the request being evaluated.

QoS evaluation is divided into seven parts as described below. Case (1) corresponds to the case when both ρ and ω are satisfied. In cases (2) through (4), none of the already committed requests are violated but one or two of the QoS goals (i.e. response time or throughput) of the request under evaluation are violated. The possible outcomes in these cases are either acceptance or a counter offer. Finally, in cases (5) through (7), at least one of the already committed requests is violated. The possible outcomes include rejections or counter offers.

(1) The current request and all committed sessions can be accepted ($\hat{\rho} \wedge \hat{\omega}$): Accept ρ.

(2) Only the response time goal for the current request is violated but all committed sessions are satisfied ($\bar{R}_{\max} \wedge \hat{\omega}$). A possible remedy can be found by decreasing the offered concurrency level. Note that this action does not negatively affect the already committed requests. More precisely,

```
if $\exists\ (1 \le N < N_{\text{req}}) \mid \hat{\rho}$
then $C_{\text{off}}\ (N_{\text{off}} = \arg\max_N\{\hat{\rho}\})$
else $C_{\text{off}}\ (N_{\text{off}} = \arg\min_N\{\hat{X}_{\min}\},\ R_{\text{off}} = R(N_{\text{off}}))$
```

(3) Only the throughput goal for the current request is violated but all committed sessions are OK ($\bar{X}_{\min} \wedge \hat{\omega}$). A possible remedy is to increase the offered concurrency level as long as it does not violate the already committed requests. More precisely:

```
if $\exists\ (N > N_{\text{req}}) \mid (\hat{\rho} \wedge \hat{\omega})$
then Accept $\rho$ with $N_{\text{off}} = \arg\min_N\{\hat{\rho} \wedge \hat{\omega}\}$
else $C_{\text{off}}\ (N_{\text{off}} = \arg\max_N\{\hat{R}_{\max} \wedge \hat{\omega}\},\ X_{\text{off}} = X(N_{\text{off}}))$
```

(4) The response time and throughput goals for the current request are violated but all committed sessions are OK ($\bar{R}_{\max} \wedge \bar{X}_{\min} \wedge \hat{\omega}$). In this case, one tries to satisfy the response time goal of the current request by lowering the offered concurrency level. This action does not negatively affect the already committed requests. If it is not possible to satisfy the response time goal, an attempt is made at satisfying the throughput goal of the current request without violating the already committed requests. If it is not possible to satisfy either the response time or the throughput goals of the current request, then send a counter offer based on the requested concurrency level. More precisely:

```
if $\exists\ (1 \le N < N_{\text{req}}) \mid \hat{R}_{\max}$
then $C_{\text{off}}\ (N_{\text{off}} = \arg\max_N\{\hat{R}_{\max}\},\ X_{\text{off}} = X(N_{\text{off}}))$
else if $\exists\ (N > N_{\text{req}}) \mid (\hat{X}_{\min} \wedge \hat{\omega})$
    then $C_{\text{off}}\ (N_{\text{off}} = \arg\min_N\{\hat{X}_{\min} \wedge \hat{\omega}\},\ R_{\text{off}} = R(N_{\text{off}}))$
    else $C_{\text{off}}\ (R_{\text{off}} = R(N_{\text{req}}),\ X_{\text{off}} = X(N_{\text{req}}))$
```

(5) The response time goal of the current request and at least one of the committed sessions are violated ($\bar{R}_{\max} \wedge \bar{\omega}$). In order to attempt to solve the problem with the already committed requests, one has to decrease the concurrency

level of the current request. If it is not possible to find such concurrency level greater than zero, then the current request must be rejected. Otherwise, a counter offer must be sent. More precisely:

```
if $\exists\ (1 \le N < N_{\text{req}}) \mid (\hat{\rho} \wedge \hat{\omega})$
then $C_{\text{off}}\ (N_{\text{off}} = \arg\max_N \{\hat{\rho} \wedge \hat{\omega}\})$
else if $\exists\ (1 \le N < N_{\text{req}}) \mid (\hat{\omega} \wedge \hat{X}_{\min})$
    then $C_{\text{off}}\ (N_{\text{off}} = \arg\max_N \{\hat{\omega} \wedge \hat{X}_{\min}\},\ R_{\text{off}} = R(N_{\text{off}}))$
    else if $\exists\ (1 \le N < N_{\text{req}}) \mid \hat{\omega}$
        then $C_{\text{off}}\ (N_{\text{off}} = \arg\max_N \{\hat{\omega}\},\ R_{\text{off}} = R(N_{\text{off}}),$
                $X_{\text{off}} = X(N_{\text{off}}))$
        else Reject $\rho$
```

(6) The throughput goal of the current request and at least one of the committed sessions are violated ($\bar{X}_{\min} \wedge \bar{\omega}$). In order to attempt to solve the problem with the already committed requests, one has to decrease the concurrency level of the current request. If it is not possible to find such concurrency level greater than zero, then the current request must be rejected. Otherwise, a counter-offer must be sent. More precisely:

```
if $\exists\ (1 \le N < N_{\text{req}}) \mid \hat{\omega}$
then $C_{\text{off}}\ (N_{\text{off}} = \arg\max_N \{\hat{\omega}\},\ X_{\text{off}} = X(N_{\text{off}}))$
else Reject $\rho$
```

(7) The response time and throughput goals for the current request are violated and at least one of the committed sessions are violated ($\bar{X}_{\min} \wedge \bar{R}_{\max} \wedge \bar{\omega}$). In order to attempt to solve the problem with the already committed requests, one has to decrease the concurrency level of the current request. If it is not possible to find such concurrency level greater than zero, then the current request must be rejected. Otherwise, a counter-offer must be sent. More precisely:

```
if $\exists\ (1 \le N < N_{\text{req}}) \mid \hat{\omega}$
then $C_{\text{off}}\ (N_{\text{off}} = \arg\max_N \{\hat{\omega}\},\ R_{\text{off}} = R(N_{\text{off}}),\ X_{\text{off}} = X(N_{\text{off}}))$
else Reject $\rho$
```

6. Performance model building

The QoS Performance Solver helps with the QoS request evaluation by modelling the impact of each request on a particular SP, which is already committed to servicing other QoS sessions. For that purpose, the QoS Performance Model Solver solves an analytical multiclass queuing network (QN) model. In the performance model, each class corresponds to one session. The model is dynamically built each time that a QoS request for a particular SP is to be evaluated. Since there may be several SPs running on the same hardware platform competing for its resources, we need to consider all sessions for all the SPs at the hardware platform when building the performance model. If there are V sessions for all these SPs in the table of commitments, the performance model will have $V + 1$ classes, V for the committed sessions for all these SPs and one for the session being evaluated. Thus, a new performance model has to be built and evaluated for each new QoS request ρ.

In many cases, response times are influenced by contention for software resources (e.g. software threads). For that reason, we used in the Performance Model Solver the two-level closed multiclass queuing network model (SQN-HQN) developed by Menascé [21]. Other models such as the Layered Queuing Network (QN) model could be used [35]. In our previous work [26], we had only accounted for hardware contention in our performance models. In the current work, we use a combination of a software queuing network (SQN) and a hardware queuing network (HQN) as illustrated in Fig. 7. The figure illustrates a SQN composed of two servers, SP1 and SP2, each with its own thread pool and its own queue for threads. A request may be queuing for a thread or using a thread for its execution. While a thread executes, it contends for the use of physical resources such as CPU and disk. This is illustrated in the bottom part of the figure, which shows a HQN with two resources: a CPU and a disk. Any thread in execution may be using the CPU or disk or waiting to use any of these two resources.

The parameters of such a model are:

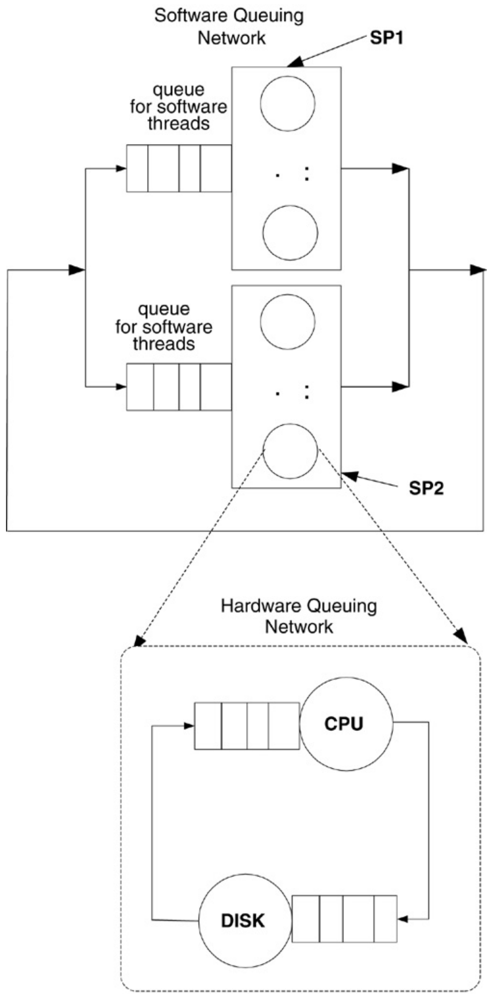


Fig. 7. A software queuing network (SQN) model combined with a hardware queuing network (HQN) model.

- $\vec{N} = (N_1, \dots, N_R)$: vector of customer populations for the SQN, where N_r $(r = 1, \dots, R)$, is the multithreading level of class r and R is the number of classes.
- D: a $J \times K \times R$ matrix of service demands where J is the number of software modules competing for hardware sources, K is the number of devices (e.g., CPU, disks) at the hardware platform, and R is the number of classes. An element $D_{j,i,r}$ of this matrix represents the total service time at physical resource i of a class r request while executing software module j. Service demands can be computed from the utilization and throughputs using the Service Demand Law [22]: $D_{j,i,r} = U_{j,i,r}/X_{0,r}$ where $U_{j,i,r}$ is the utilization of device i by class r when executing software module j, and $X_{0,r}$ is the system throughput of class r requests.

We describe now how $\vec{N}$ and D are built every time a QoS request ρ is to be evaluated. We define first the Base Matrix of Service Demands B as the $J \times K \times M$ matrix where M is the number of services provided by all the SPs at the hardware platform. An element $B_{j,i,m}$ of this matrix represents the service demand at physical resource i of a service m while executing software module j. The Base Matrix of Service Demands is built by the QoS broker from the service demands registered by the SPs.

In the ToC, each entry corresponds to a SPID and a set of commitments. Let ToC$_h$ be the subset of the ToC that only contains entries for services running on hardware platform h and let the request under evaluation be associated with a service running on h. Let V be the number of entries in ToC$_h$, let ToC$_h[k]$.ServiceID $(k = 1, \dots, V)$ be the service id of the commitment at entry k of ToC$_h$ and let ToC$_h[k]$.N be the concurrency level of requests to be executed at service ToC$_h[k]$.ServiceID. We assume that ServiceID uniquely identifies a service offered by the SPs running at

New Request:SPID = 1, Service ID = 1, N = 12

Base Matrix of Service Demands (in msec):

		SP 1		SP 2	
		ServiceID		ServiceID	
Device	Software Module	0	1	2	3
CPU	1	25	34	18	20
Disk	1	30	50	15	24
CPU	2	12	14	55	45
Disk	2	20	23	31	25

Table of Commitments (ToC):

Commitment ID	Service ID	N	...
1	3	10	...
2	2	15	...
3	0	8	...
4	0	12	...
5	1	13	...
6	3	8	...

Matrix of Service Demands (in msec):

		Class						
Device	Software Module	1	2	3	4	5	6	7
CPU	1	20	18	25	25	34	20	34
Disk	1	24	15	30	30	50	24	50
CPU	2	45	55	12	12	14	45	14
Disk	2	25	31	20	20	23	25	23

Vector N:	10	15	8	12	13	8	12

Fig. 8. Example of dynamic model building.

that hardware platform. The parameters of the QN model are obtained from ToC_h and from the Base Matrix of Service Demands as follows. The number of classes in the model is equal to the number of commitments, V, plus 1 (for the new QoS request being evaluated). The first V columns of the matrix of service demands, D, are initialized based on the columns of the Base Matrix of Service Demands that correspond to the service associated to the commitment. That is, each column of the Base Matrix of Service Demands corresponds to a service, and contains the service demands at both hardware and software resources. If a service does not use a software resource, the corresponding service demands at the hardware devices are set to zero. The last column of the matrix D is obtained from the column in matrix B associated to the service requested in the new QoS request.

The process of model building is illustrated in Fig. 8 for a QoS request that requires 12 concurrent executions of service 1 of SP1. The Base Matrix of Service Demands indicates that SP1 and SP2 offer two services each, and that they share the same hardware platform, composed of one CPU and one disk. The ToC in Fig. 8 shows that there are six commitments. For example, commitment number 3 has a concurrency level of 8 for service 0. This commitment is associated with class 3 in the model. Therefore, column 3 of the service demand matrix is the column associated with service 0 in the Base Matrix of Service Demands. The last column of the matrix of service demands is associated with the request being evaluated. Therefore, the service demands for the new request come from column 2 (i.e. service 1) of matrix B. The first six elements of the population vector $\vec{N}$ are obtained from the column labelled N in the ToC and the last element is the value of N (i.e. 12) indicated in the QoS request.

7. Fault-tolerance and performance issues

Two important issues have to be addressed in the QoS broker architecture described in this paper: fault tolerance and performance. It can be argued that a failure of the QB could disrupt operations of the services that depend on it. Resilience to failure of the QB can be improved by taking the following actions: (i) the QB must store in stable storage the IDs of all the services registered with it as well as their service demands, (ii) the QB must write an entry in a write-ahead log [13] every time that an accept request or counter offer is sent to a client but before the message is sent, (iii) the QB must write an entry to the log when it receives an accept counter offer or an end-of-session message, and (iv) the QB must write an entry to the log every time that a temporary commitment expires. The log can be used to rebuild the QB's ToC in case of a failure of the QB. This can be done in a straightforward manner by processing the write-ahead log backwards as is done in database management system failure recovery [13].

Another addition to the scheme proposed here is to replace a single QB with a collection of QBs, as is done in replicated X.500 directory systems or in replicated UDDI registries [31]. At any point in time, only one QB should be

Table 1
Base matrix of service demands (in sec)

Device	Software module	Service 0	Service 1	Service 2	Service 3
CPU	SP 1	0.05055	0.03455	0	0
Disk	SP 1	0.02246	0.01494	0	0
CPU	SP 2	0	0	0.05862	0.04270
Disk	SP 2	0	0	0.02823	0.01963

responsible for brokering QoS requests for all the SPs running on a given machine. Distributed election algorithms [9] can be used to elect a new QB should the QB for a given set of SPs fail. This is possible if the state information (i.e. the ToC and service registrations) of the failed QB is replicated. A replication scheme allows for load balancing techniques to be used to improve performance of the set of QBs.

8. Experimental evaluation

Two experiments were run with the same client workload making demands on two SPs. The first experiment was run without the QoS broker and the second with the broker. The results are then compared to evaluate the impact of the QoS broker.

8.1. Experimental setup

The experiments were performed using three machines connected through a LAN. One of the machines runs two SPs, each of which provides two types of services: Service 0 and Service 1 are provided by SP1 and Service 2 and Service 3 by SP2. These four services provide stock quotes by accessing a database. These four services are independent of each other. Each SP enqueues all incoming requests dedicated to it and runs ten parallel threads to execute these requests. This situation is modelled by a software module that can be considered as a multi-resource queue in the SQN. Another machine runs the QoS broker implemented in Java. The third machine runs several client processes.

It is assumed that the clients know the location of the SPs, so no service location is involved. The Base Matrix of Service Demands is initialized by repeatedly executing requests for each of the services and by measuring the utilization of the various resources during this process. The results of these measurements are shown in Table 1 for the CPU and disk and for each service of each SP.

In order to compare the performance of a SP that does not use the QB with that of a SP that uses the QB, we first generated session logs with entries of the form (SessionID, ServiceID, N_{req}) and stock logs with entries of the form (SessionID, StockName, date, time). SessionID is an identification of a session; ServiceID identifies a service; N_{req} is the requested concurrency level, i.e. the maximum number of concurrent requests to execute service ServiceID in the session; StockName, date, and time represent a stock to be queried. Then, clients in both the non-QB case and the QB-case send requests based on the session log and the stock log. The details of the experiments are as follows:

(1) Log generation

We used ten client processes generating requests for services offered by SP1 and ten processes generating requests for services offered by SP2. Each client runs N parallel threads and each thread makes T sequential requests to a service of its assigned SP. Each of the 20 client processes generates its own session log and stock log by repeating the following steps 10 times:

- Increase the session id (SessionID) by one (starting from zero).
- Randomly generate a service id (ServiceID) between the two services offered by the SP assigned to the process.
- Randomly select $N \times T$ ($N = 10, T = 5$) stock entries (StockName, date, time) among P ($P = 480$ in our experiments) stock entries.
- Write (SessionID, ServiceID, N) and (SessionID, StockName, date, time) into the session log and the stock log; each process creates its own session log and its own stock log. Thus, there are 20 session logs and 20 stock logs.

(2) Non-QB experimental procedure

(a) Each of the 20 client processes first builds a join of the session log with the stock log using SessionID as the key to produce entries of the form (SessionID, StockName, date, time).

(b) Each client process starts by reading the session log. For each entry, the client process does the following:
- Read an entry (SessionID, ServiceID, N) from the session log.
- Create N ($N = 10$ in our experiments) concurrent threads. For each thread, T ($T = 5$ in our experiments) stocks are selected from the corresponding stock collection in the joined session and stock logs. The start time, SessionStart, of the session is recorded by the process.
- Start the N concurrent threads. Each thread repeats T times a loop in which a stock (StockName, date, time) is selected and a request (SessionID, ServiceID, StockName, date, time) for service ServiceID is sent; a reply must be received before a new request is sent. At the end of the loop, each thread adds to the accumulated session response time, SessionAccumRespTime, the total response time for the T requests submitted by the thread.
- After all threads have finished, the session end time, SessionEnd, is recorded and an end session message is sent to the server.
- The client process computes the average response time, R_s, for the session as:
  $R_s = \text{SessionAccumRespTime}/(N \times T)$,
  and the average session throughput X_s as
  $X_s = (N \times T)/(\text{SessionEnd} - \text{SessionStart})$.
- Add the tuple (SessionID, ServiceID, N, R_s, X_s) as an entry in the process log (Note, each client process has its own log).
- Combine all session response time logs into a single client response time log.

After all sessions of a client have finished, the client's response time log is written to disk.

The above non-QB experimental procedure was repeated ten times. For each service, the average response time $\bar{R}_{\text{nonQB}}$ and average throughputs $\bar{X}_{\text{nonQB}}$ were computed based on all data in the ten experiments. Specifically, service 0 has 26,500 data points; 23,500 data points were measured for service 1; 22,000 for service 2; and 28,000 for service 3.

(3) QB experimental procedure

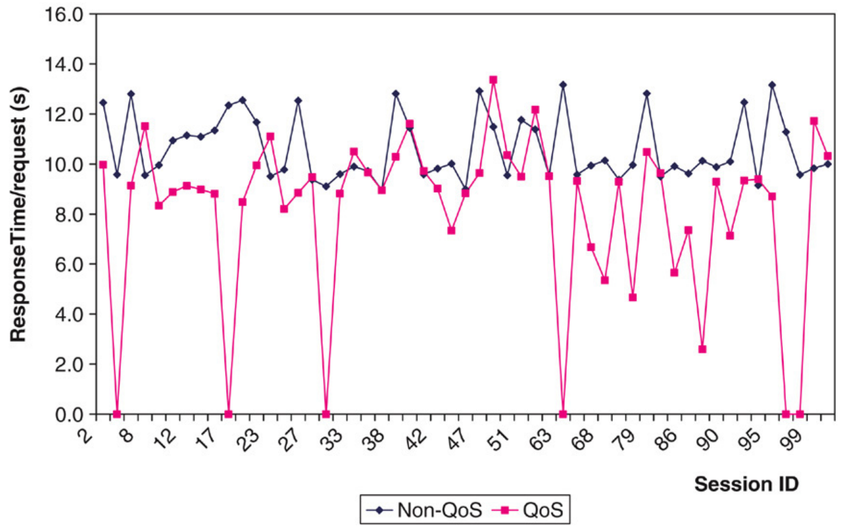
(a) In the QB case, each of the 20 client processes starts by performing the actions described in step 2(a) of the non-QB experimental procedure.

(b) Then, for each session, each process generates a QoS request $\rho =$ (SessionID, SPID, ServiceID, N_s, RMAX, XMIN) where (i) N_s is computed as $N_s = \bar{X}_{\text{nonQB}} \times \bar{R}_{\text{nonQB}}$ using Little's Law; (ii) RMAX is obtained as follows. During the non-QB case experiments, n response time values were obtained (over 20,000 values in our case). According to the Central Limit Theorem, the sample average response time is normally distributed with mean $\bar{R}$ and standard deviation $\sigma/\sqrt{n}$ where $\bar{R}$ is the sample average response time and σ is the sample standard deviation. Of course, this does not mean that the response time is normally distributed. Thus, one obtains a value R_{norm} from a $N(\bar{R}, \sigma/\sqrt{n})$ distribution and then reduce it using a factor $f (0 \leq f < 1)$ that indicates that the response time in the QB case should be smaller than that obtained in the non-QB case. Therefore, $\text{RMAX} = R_{\text{norm}} \times (1 - f)$; (iii) XMIN is obtained from Little's Law as $\text{XMIN} = N_s/\text{RMAX}$.

(c) A QoS negotiation procedure is started by sending the QoS request ρ to the QoS broker. If the QoS request is accepted in its original form or as a counter offer, the client starts N concurrent threads as in the non-QB experiment. In our experiments, after the negotiation, the clients always start N threads regardless of the values of N_s or N_{offer}. This was done to keep the load submitted to the SP as close as possible to the non-QB case. The SP rejects requests that violate the negotiated concurrency level.

8.2. Measurement results

The non-QB and QB experiments were repeated 10 times each and the average response time and average throughput for each session were computed. As explained before, each experiment contains 200 sessions and each session makes 50 requests. Fig. 9 shows the average response time of SP1 for services 0 and 1, for a reduction factor of $f = 0.2$. Because SPs perform admission control in the QB case, QoS requests may be rejected. So, the average reported in the graph for the QB experiments is computed over the non-rejected requests in the 10 executions. If all ten executions of a session result in rejected requests, a value of zero is reported for the response time. The graphs of Fig. 9 only contain data for sessions belonging to SP1. Data for SP2 was deleted from the global session stream in



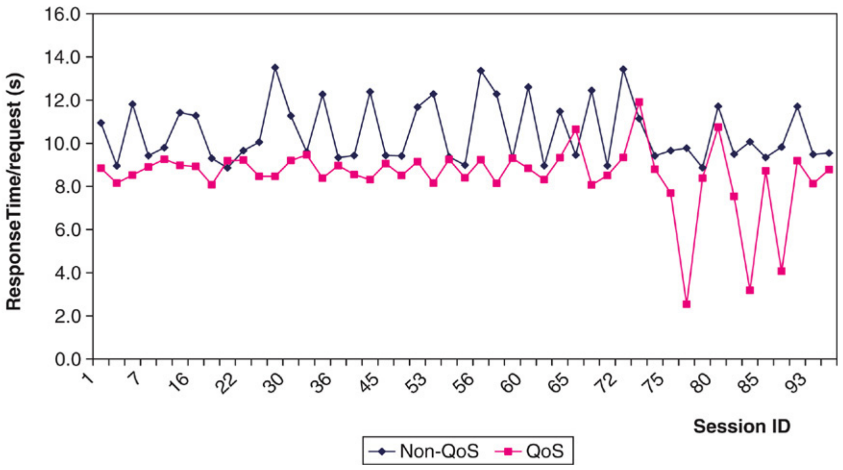


Fig. 9. (top) Response time for service 0 in SP1 ($f = 0.2$). (bottom) Response times for service 1 in SP1 ($f = 0.2$).

order to generate the graphs. As indicated in the figures, in all sessions for both services, the average response time was lower in the QB case when compared with the non-QB case.

As expected, the performance gain, i.e. decreased response time (16.7%) for SP1 in the QB case, was consistent with the requested response time reduction of 20%. The overall performance of the QoS case relative to the non-QoS case for each service with different f values is shown in Table 2. In the table, % RT Reduction stands for the percentage of average response time reduction in the QB case, and % XPUT stands for the percentage of average increase in throughput. The percentage of QoS requests accepted, rejected, and receiving counter offers are also shown in the table.

As Table 2 shows, with a requested 35% response time reduction, there was a 35.2% and 35.0% total response time reduction for SP1 and SP2, respectively; with a requested 20% response time reduction, there was a 16.7% and 21.2% response time reduction for SP1 and SP2, and with no response time reduction requested, there was 4.1% and 4.9% response time reduction, respectively. In the latter case, the control mechanism seems to be overreacting and may be rejecting more QoS requests than necessary. But, for the more stringent QoS goals, in the former case, the control mechanism approximates very well to the requirements. As shown in the table, in all cases, the two SPs always performed better in the QB case than in the non-QB case, at the cost of dropping some sessions. Since in our experiments clients accept all counter offers, we are not expecting the results to exactly match our reduction goals. However, the results shown here reasonably match the requirements.

It should be noted that the experiments reported in this section reflect the impact of the QoS broker on the performance of the system since they include an actual implementation of the broker. It is important to realize as

Table 2
Summary of results for $f = 0.0$, $f = 0.20$, and $f = 0.35$

		$f = 0\%$	$f = 20\%$	$f = 35\%$
	% RT reduction	4.1	16.7	35.2
	% XPUT increase	9.0	24.0	54.7
SP 1	% Rejected	17.0	22.7	36.9
	% Counter offer	6.9	9.3	9.9
	% Acceptance	76.1	68.0	53.2
	% RT reduction	4.9	21.2	35.0
	% XPUT increase	12.1	24.0	54.7
SP 2	% Rejected	15.2	29.3	39.8
	% Counter offer	10.8	9.7	10.4
	% Acceptance	74.0	61.0	49.8

well that the QoS broker is only used during the negotiation phase, which occurs once per session. So, the negotiation cost is amortized over all service requests of a session. The QoS broker proposed and analyzed here uses efficient analytical models to evaluate QoS requests, therefore, affording fast negotiation capabilities.

9. Concluding remarks and future work

This paper has presented an architecture for negotiating QoS goals in Service Oriented Architectures. A QoS broker uses analytical models to evaluate QoS requests. Analytical models were used for the first time in [24] for predicting performance and exercising QoS control for e-commerce sites [18,24] and for Web servers [25]. Bennani and Menascé used online analytic models for dynamic server allocation in autonomic data centres [2]. Others used the same approach and applied the idea to performance management of cluster-based Web services [16]. In [18,24], the idea was to dynamically change configuration parameters of a computer system in order to match QoS requirements. In the case of the research discussed in this paper, analytical models are used to evaluate QoS requests so that admission control can be carried out.

The approach presented here was completely implemented in Java and extensive experimentation was conducted to assess its effectiveness. For experimentation purposes, we decided to use artificial services so that we could have more control over the experiments. The results indicated that the QoS broker was able to successfully negotiate QoS requirements on behalf of QoS-aware service providers and in all cases either match the response time and throughput requirements or exceed them by some margin. The price to be paid in return for meeting the QoS goals is that some sessions may be rejected. The introduction of the QoS broker eases the composition of QoS-aware applications, especially in the Web Services domain.

We now examine how the requirements, described in the introduction, are met by our solution: (i) QoS-based service registration ability: the QB allows service providers to register the QoS demands for their services; (ii) Negotiation capability: as demonstrated here, the QB is able to carry out QoS negotiations on behalf of multiple SPs from the same or different hardware platforms. Each SP must reside on a single hardware platform. These SPs need to register with the QB in order for it to act as their QoS negotiator. (iii) Resource reservation ability: once a QoS agreement is reached between the client and the QB on behalf of a SP, the QB reserves resources in the SP by capturing the commitments in a table of commitments. When a new performance model has to be built to evaluate subsequent requests, previous commitments, including temporary ones, are taken into account. (iv) Limited SP involvement in the QoS negotiation: the SP is not involved during the QoS negotiation; (v) Certifiable negotiation: the QB provides the client with an unforgeable token signed by the QB. The SP is then able to verify the signature of the QB and be certain that the negotiation specified by the token was carried out by the QB. The cost of this certification is as follows: The QB needs to carry out one private key operation in order to sign a token. Then, the QB needs to perform a public key operation to verify the token's signature in the following three cases: a counter offer is received or a rejection is received. The SP needs to perform a public key operation per service request in order to verify the signature of the token. Each signing operation and each signature verification operation also includes a secure hash operation. It should be noted that private key operations are more computationally expensive than public key operations. Also, secure hash functions are significantly less expensive than private or public key operations [23].

As part of our future work, we will develop a utility algorithm for the selection of service provider components. We intend to use the framework developed and tested here to implement distributed applications. We also intend to extend the framework into the Web Services domain by integrating the QoS broker with the UDDI registry. We also intend to investigate how our approach can be extended to the situation where components need to invoke other components to perform their services. We are also investigating the problem of determining QoS goals for Q-components that are part of a distributed application. In other words, we want to find the best possible component-level requirements given global application QoS goals. We already have some preliminary results on this problem [19]. Other issues not covered here have to do with the costs associated with providing certain QoS goals and how this fact could be taken into account during the negotiation. It is clear that the area of QoS management in service-oriented architectures has many distinct facets and can, therefore, be the target of substantial future work.

References

[1] R.J. Al-Ali, A. Hafid, O.F. Rana, D.W. Walker, QoS adaptation in service-oriented grids, in: Proc. 1st Intl. Workshop Middleware for Grid Computing (MGC2003) at ACM/IFIP/USENIX Middleware 2003, 16–20 June, Rio de Janeiro, Brazil, 2003.

[2] M. Bennani, D.A. Menascé, Resource allocation for autonomic data centers using analytic performance models, in: Proc. Second Intl. Conf. Autonomic Computing, 13–16 June, Seattle, WA, 2005.

[3] C. Cavanaugh, L.R. Welch, B. Shirazi, E. Huh, S. Anwar, Quality of service negotiation for distributed, dynamic real-time systems, in: IPDPS Workshop on Bio-Inspired Solutions to Parallel Processing Problems, BioSP3, 15 April, Fort Lauderdale, FL, 2002, pp. 757–765.

[4] G. Dermler, W. Fiederer, I. Barth, K. Rothermel, A negotiation and resource reservation protocol (NRP) for configurable multimedia applications, in: Proc. Third IEEE Intl. Conf. Multimedia Computing and Systems 1996, 17–23 June, Hiroshima, Japan, 1996, pp. 113–116.

[5] W. Emmerich, D. Lamanna, G. Piccinelli, J. Skene, Method for service composition and analysis. http://www.newcastle.research.ec.org/tapas/deliverables/index.html.

[6] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy, A distributed resource management architecture that supports advance reservations and coallocation, in: Proc. Intl. Workshop Quality of Service 1999, UCL, 1–4 June, London, 1999, pp. 27–36.

[7] L. Foster, A. Roy, Quality of service architecture that combines resource reservation and application adaptation, in: Proc. Eighth Intl. Workshop Quality of Service, IWQOS 2000, 5–7 June, Westin William Penn, Pittsburgh, 2000, pp. 181–188.

[8] S. Frolund, J. Koistinen, Quality of service aware distributed object systems, Hewlett-Packard Company Technical Report. http://www.hpl.hp.com/techreports/98/HPL-98-142.html, 1998.

[9] H. Garcia-Molina, Elections in distributed computing systems, IEEE Transactions on Computers 31 (1) (1982) 48–59.

[10] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures, in: Addison-Wesley Object Technology Series, Reading, MA, 2005.

[11] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, in: Addison-Wesley Object Technology Series, Reading, MA, 2000.

[12] H. Gomaa, D.A. Menascé, L. Kerschberg, A software architectural design method for large-scale distributed data intensive information systems, Journal of Distributed Systems Engineering 3 (1996) 162–172.

[13] J.N. Gray, The transaction concept: Virtues and limitations, in: Proc. Intl. Conf. Very Large Databases, 1981, pp. 144–154.

[14] J. He, M.A. Hiltunen, M. Rajagopalan, R.D. Schlichting, Providing QoS customization in distributed object systems, in: IFIP/ACM Intl. Conf. Distributed Systems Platforms, Middleware 2001, 12–16 November, Heidelberg, Germany, 2001, pp. 351–372.

[15] A. Kraiss, F. Schoen, G. Weikum, U. Deppisch, Towards response time guarantees for e-service middleware, IEEE Data Engineering Bulletin 24 (1) (2001) 58–63.

[16] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, A. Youssef, Performance management for cluster based web services, in: Proc. Int. Net. Mgt. Conf., IM2003, March 2003.

[17] E.M. Maximilien, M.P. Singh, A framework and ontology for dynamic web services selection, IEEE Internet Computing 8 (5) (2004) 84–93.

[18] D.A. Menascé, Automatic QoS control, IEEE Internet Computing 7 (1) (2003) 92–95.

[19] D.A. Menascé, Mapping service-level agreements in distributed applications, IEEE Internet Computing 8 (5) (2004) 100–102.

[20] D.A. Menascé, QoS issues in web services, IEEE Internet Computing 6 (6) (2002) 72–75.

[21] D.A. Menascé, Two-level iterative queuing modeling of software contention, in: Proc. 10th IEEE Intl. Symp. Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS'02, 11–16 October, Fort Worth, Texas, 2002, pp. 267–276.

[22] D.A. Menascé, V.A.F. Almeida, Capacity Planning for Web Services: Metrics, models, and methods, Prentice Hall, Upper Saddle River, NJ, 2002.

[23] D.A. Menascé, V.A.F. Almeida, Scaling for E-business: Technologies, models, performance, and capacity planning, Prentice Hall, Upper Saddle River, NJ, 2000.

[24] D.A. Menascé, D. Barbará, R. Dodge, Preserving QoS of E-commerce sites through self-tuning: A performance model approach, in: Proc. 2001 ACM Conf. E-commerce, 14–17 October, Tampa, FL, 2001.

[25] D.A. Menascé, M. Bennani, On the use of performance models to design self-managing computer systems, in: Proc. 2003 Computer Measurement Group Conf., 7–12 December, Dallas, TX, 2003.

[26] D.A. Menascé, H. Ruan, H. Gomaa, A framework for QoS-aware software components, in: Proc. Fourth Intl. Workshop Software and Performance, WOSP'04, January 2004, pp. 186–196.

[27] H. Gomaa, D.A. Menascé, Design and performance modeling of component interconnection patterns for distributed software architectures, in: Proc. Workshop on Software Performance, ACM Press, Ottawa, Canada, September 2000, pp. 117–126.
[28] H. Gomaa, D.A. Menascé, Performance engineering of component-based distributed software systems, in: R. Dumke, C. Rautenstrauch, A. Schmietendorf, A. Scholz (Eds.), Performance Engineering, in: LNCS, no. 2047, Springer Verlag, 2001, pp. 40–55.
[29] D.A. Menascé, H. Gomaa, A method for design and performance modeling of client/server systems, IEEE Transactions on Software Engineering 26 (11) (2000) 1066–1085.
[30] D.A. Menascé, H. Gomaa, L. Kerschberg, A performance-oriented design methodology for large-scale distributed data intensive information systems, in: Proc. First IEEE International Conf. on Eng. of Complex Computer Systems, Southern Florida, USA, November 1995.
[31] OASIS, UDDI Version 2.03 Replication Specification. http://uddi.org/pubs/Replication-V2.03-Published-20020719.htm, July 2002.
[32] D. Petriu, H. Amer, S. Majumdar, I. Abdull-Fatah, Using analytic models for predicting middleware performance, in: Proc. Second Intl. Workshop Software and Performance, Ottawa, Canada, 17–20 September, 2000.
[33] J. Pruyne, Enabling QoS via interception in middleware, Hewlett-Packard Company Technical Report. http://www.hpl.hp.com/techreports/2000/HPL-2000-29.html, 2000.
[34] S. Ran, A model for web services discovery with QoS, ACM SIGEcom Exchanges 4 (1) (2003) 1–10.
[35] J.A. Rolia, K.C. Sevcik, The method of layers, IEEE Transactions on Software Engineering 21 (8) (1995) 689–700.
[36] F. Siqueira, V. Cahill, Quartz: A QoS architecture for open systems, in: The 20th Intl. Conf. Distributed Computing Systems, ICDCS 2000, 10–13 April, Taipei, Taiwan, 2000, pp. 197–204.
[37] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, J. Schiller, A concept for QoS integration in web services, in: Fourth Intl. Conf. Web Information Systems Engineering Workshops, WISEW'03, Roma, Italy, December 2003, pp. 149–155.
[38] D.L. Tien, O. Villin, C. Bac, Resource managers for QoS in CORBA, in: Second IEEE International Symp. Object-Oriented Real-Time Distributed Computing, 2–5 May, Saint-Malo, France, 1999, pp. 213–222.
[39] Simple Object Access Protocol, SOAP. http://www.w3.org/TR/SOAP.
[40] Universal Description, Discovery, and Integration of Business for the Web. http://www.uddi.org.
[41] Web Services Description Language, WSDL. http://www.w3.org/TR/wsdl.
[42] World Wide Web Consortium, W3C. Web Services Activities. http://www.w3.org/2002/ws/.
[43] C.M. Woodside, D.A. Menascé, QoS for distributed Applications, guest editors introduction, IEEE Internet Computing 10 (3) (2006).

Daniel A. Menascé is the Associate Dean for Research and Graduate Studies and a Professor of Computer Science at the Volgenau School of Information Technology and Engineering at George Mason University, Fairfax, Virginia. He received a Ph.D. in Computer Science from the University of California at Los Angeles. He is a Fellow of the ACM, a senior member of the IEEE, and a member of IFIP Working Group 7.2. He received the 2001 A.A. Michelson Award from the Computer Measurement Group. Menascé is the author of over 185 papers and five books.

His current research interests include autonomic computing, e-commerce, performance analysis, and software performance engineering. His research has been funded by several organizations including the National Science Foundation, NASA, and DARPA. Menascé consulted for several US government agencies and several large companies.

Honglei Ruan is a Web developer at ValueOptions, Reston, Virginia. She received a Bachelor of Science degree in Computer Science Engineering from Northeastern University at Qinhuangdao, China in 2000, and a Master of Science degree in Computer Science from George Mason University in 2002. From 2002 to 2003 she worked as a Research Assistant in the E-Center for E-Business at George Mason University.

Hassan Gomaa is Chair and Full Professor in the Department of Information and Software Engineering at George Mason University, Fairfax, Virginia. He received a Ph.D. in Computer Science from Imperial College of Science and Technology, London University. He has worked in both industry and academia, and has published over 140 technical papers and three textbooks on software design. His current research interests include object-oriented modeling and design for concurrent, real-time, and distributed systems, software product line engineering, component-based software architectures, and software performance engineering. His research has been funded by several organizations including the National Science Foundation, NASA and DARPA. He has taught several in-depth industrial courses on software design in North America, Europe, and Asia.