QoS-Aware Software Components

Daniel A. Menascé • *George Mason University*

In previous columns, I've discussed some of the challenges in quality of service (QoS) for Web services[1] and the computation of response time in composite Web services.[2] In this installment, I address the more general issue of QoS in service-oriented component-based distributed systems.

The next generation of complex software systems will be highly distributed, component-based, and service-oriented. They will need to operate in unattended mode, possibly in hostile environments, and they'll be composed of many "replaceable" components discoverable at runtime. Moreover, they 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 these systems must be adaptable and self-configurable to changes in workload intensity. Availability and security requirements suggest that these systems also must adapt and reconfigure themselves to withstand attacks and failures. In this column, I focus specifically on QoS requirements for performance and describe the framework for QoS-aware distributed applications that I developed with my colleagues at George Mason University.[3]

QoS-Aware Applications

A QoS-aware application — or, more simply, a Q-application — is dynamically composed of QoS-aware components, or Q-components, which run on a single hardware platform. As Figure 1a (next page) indicates, before a Q-component becomes part of a Q-application, it must register with some service directory, such as in Universal Description, Discovery, and Integration.[4] A Q-application then can discover the Q-components that provide given services by interacting with the directory service. After this, a QoS negotiation between the Q-application and the Q-component occurs. If the negotiation is successful, the Q-component becomes part of the Q-application; other components in the Q-application can then access the Q-component's services (see Figure 1b).

A Q-component's main tasks are to register its services, engage in QoS negotiation (which can result in QoS requests being accepted or rejected, or in counteroffers being made to the requester), provide services for concurrent requests while preserving QoS guarantees, and maintain a table of the QoS commitments made to other components. QoS negotiation is done at the session level — that is, in a sequence of service requests.

A Q-Component's Architecture

To better understand the concepts of Q-applications and Q-components, think of an analogy to the human body and cells. The human body (an application) is composed of many different kinds of cells (components) such as skin cells, brain cells, liver cells, and so on, all of which have unique functions and features just as different software components offer different types of services. However, many similarities or functions exist in all cells. They all have a membrane, for example, which regulates the movement of water and nutrients in and out of the cell. Similarly, all Q-components have a common set of functionalities aimed at QoS negotiation and control, regardless of the services they provide.

Figure 2 (next page) depicts the architecture of a Q-component that offers n services. The shadowed boxes in the figure correspond to the elements that make the component QoS-aware. A Q-component, like any regular service-oriented component, has a service registration module that implements its interaction with a directory service. A service dispatcher receives requests to execute a service and sends them to an available thread that can perform the execution. With Q-components, service execution occurs only if the request is part of an accepted QoS session. The service dispatcher implements admission control within accepted QoS sessions. The dispatcher rejects service requests that would cause the session's negotiated concurrency level to be exceeded.

 1089-7801/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

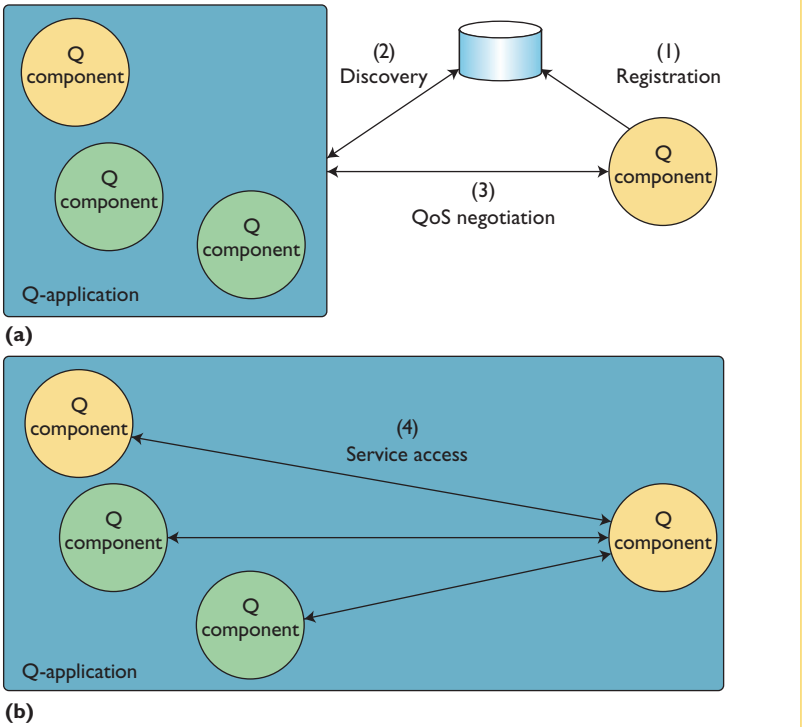

Figure 1. A quality of service (QoS)-aware application (also called a Q-application). (a) A QoS-aware component (also called a Q-component) registers and is discovered by a Q-application, which engages in QoS negotiation with the component. If successful, the component becomes part of the application, and (b) the application's other components can access this component's services.

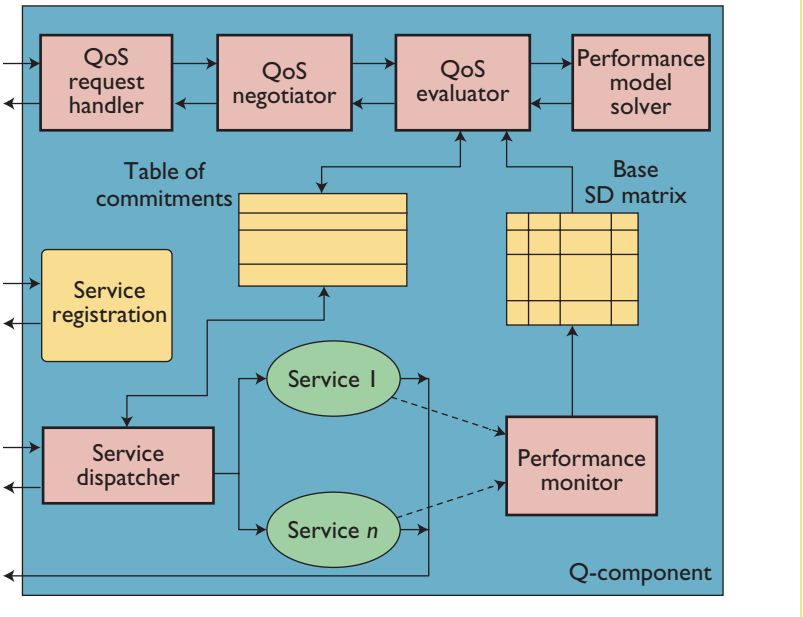

Figure 2. The architecture of a Q-component. The pink boxes indicate the elements exclusive to QoS negotiation and admission control.

The QoS request handler implements the QoS negotiation protocol with the help of the QoS negotiator, QoS evaluator, and performance model solver. The QoS request handler can receive four types of messages:

- `QoSRequest` is a request to start a session with certain QoS guarantees. If the new request violates existing commitments or if no viable counteroffers exist at a lower QoS level, it must be rejected. A QoS request is accepted when existing commitments and the requirements of the new request are met. In this case, the Q-component places the request in the table of commitments (TOC; see Figure 2); this process generates an encrypted token, which goes back to the requester with the acceptance message. This token must go with any service requests to validate them as part of an accepted session. When a Q-component generates a counteroffer, the modified QoS request stays in the TOC in a temporary state to give the requester a chance to accept or reject the counteroffer, which expires after a certain time period. If the acceptance arrives too late, the Q-component notifies the requester.
- `AcceptCounterOffer` indicates to a Q-component that a counteroffer is accepted. In this case, the temporary commitment, if not yet expired, is made permanent.
- `RejectCounterOffer` indicates that a counteroffer was rejected. The Q-component deletes the temporary commitment from the TOC.
- `EndSession` message indicates to the Q-component that a session has ended. The Q-component deletes the TOC entry that corresponds to this session.

The QoS evaluator performs the QoS request evaluation with the help of the performance model solver, which solves a closed multiclass queuing network

(QN) model via approximate mean-value analysis.[5] In the QN model, each class corresponds to one session; the model is built dynamically each time a QoS request must be evaluated. If the TOC contains V sessions, the performance model will have $V + 1$ classes, V for the committed sessions and one for the session under evaluation. More details about computing the model parameters appear elsewhere.[3]

Experimental Evaluation

To evaluate this approach's efficacy, my colleagues at George Mason University and I implemented in Java a Q-component that provides three services. A client generated a random workload and submitted it to a similar component with its QoS elements disabled. We recorded this workload and the resulting QoS levels (such as response time and throughput), and then replayed and recorded the workload against the Q-component over 10 experiments. When we used the Q-component, we set the negotiated response time to the original response time for the non-QoS case reduced by a factor f.

Figure 3 indicates average response time for service 2 (one of the Q-component's services) for the non-QoS and QoS controlled cases. The x-axis indicates the IDs of the sessions associated with service 2. The QoS case results represent averages over 10 executions. However, because a Q-component performs admission control, QoS requests could be rejected, so the average reported in the graph is over the number of nonrejected requests in the 10 executions. If all 10 executions result in rejected QoS requests, a value of zero is reported for the response time.

We obtained Figure 3's results by using a reduction factor f for the response time equal to 25 percent — in other words, we requested a 25 percent reduction in response time during QoS negotiation with respect to the value obtained during the non-QoS case's execution. The actual reduction in response time was 24.4 percent over all three types of services. This is an excellent match with the 25 percent reduction goal; the Q-component negotiated QoS levels and exercised admission control as expected. Thirty-three percent of the sessions negotiated in the case of Figure 3 were rejected.

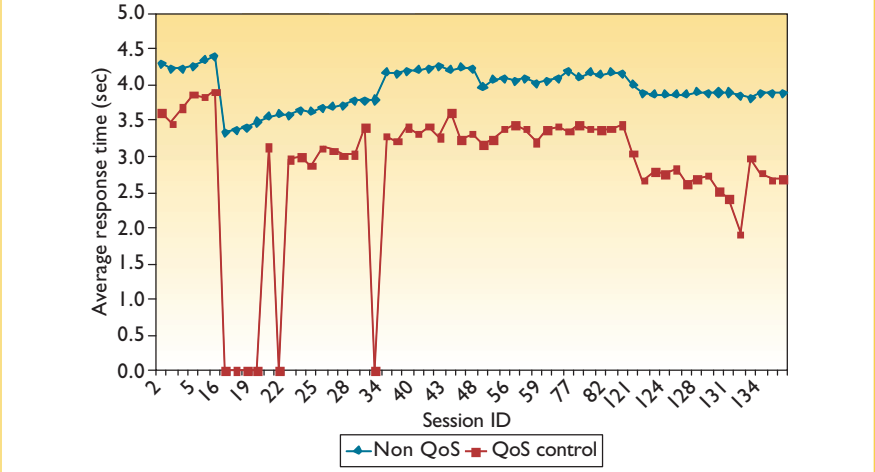

Figure 3. Average response time for service 2 sessions. Service 2 is one of a typical Q-component's services; the figure depicts non-QoS (blue curve) and QoS controlled cases (red curve). The Q-component exhibits a better response time than the non-QoS one at a cost of rejecting some sessions.

What's Next?

The work I discuss here represents an attempt to provide software components with the capabilities needed to negotiate and control QoS levels and react to a changing workload. Several interesting challenges lie ahead, including how to build Q-applications out of Q-components and how to determine the breakdown of application-level QoS goals into component-level QoS goals.

As more applications start to use Web services discovered on the fly, and as the number of competing services with similar functionality increases, QoS will become a differentiating factor; the techniques discussed here will become even more important.

References

1. D.A. Menascé, "QoS Issues in Web Services," *IEEE Internet Computing*, vol. 6, no. 6, 2002, pp. 72–74.
2. D.A. Menascé, "Response-Time Analysis of Composite Web Services," *IEEE Internet Computing*, vol. 8, no. 1, 2004, pp. 90–92.
3. D.A. Menascé, H. Ruan, and H. Gomaa, "A Framework for QoS-Aware Software Components," *Proc. 2004 ACM Workshop on Software and Performance* (WOSP'04), ACM Press, 2004, pp. 186–196.
4. F. Curbera et al., "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, 2002, pp. 86–93.
5. D.A. Menascé, V.A.F. Almeida, and L.W. Dowdy, *Performance by Design: Computer Capacity Planning by Example*, Prentice Hall, 2004.

Acknowledgments

Grant number ACI 0203872 from the US National Science Foundation partially supported this work.

Daniel A. Menascé is a professor of computer science, the codirector of the E-Center for E-Business, and the director of the MS in E-Commerce program at George Mason University. He received a PhD in computer science from UCLA and published the books *Performance by Design*, *Capacity Planning for Web Services*, and *Scaling for E-Business* (Prentice Hall, 2004, 2002, and 2000). He is a fellow of the ACM and a recipient of the 2001 A.A. Michelson Award from the Computer Measurement Group. Contact him at menasce@sc.gmu.edu.