Building a Reliable Message Delivery System Using the CORBA Event Service

Srinivasan Ramani[1], Balakrishnan Dasarathy[2], and Kishor S. Trivedi[1]

[1]Center for Advanced Computing and Communication
Department of Electrical and Computer Engineering
Duke University, Durham, NC 27708-0291, USA
{sramani, kst}@ee.duke.edu
[2]Telcordia Technologies
445 South Street
Morristown, NJ 07960, USA
bdasarat@telcordia.com

Abstract. In this paper we study the suitability of the CORBA Event Service as a reliable message delivery mechanism. We first show that products built to the CORBA Event Service specification will not guarantee against loss of messages or guarantee order. This is not surprising, as the CORBA Event Service specification does not deal with Quality of Service (QoS) and monitoring issues. The CORBA Notification Service, although it provides much of the QoS features, may not be an option. Therefore, we examine application-level reliability schemes to build a reliable communication means over the existing CORBA Event Service. Our end-to-end reliability schemes are applicable to management applications where state resynchronization is possible and sufficient. The reliability schemes proposed provide resilience in the face of failures of the supplier, consumer, and the Event Service processes.

1 Introduction

CORBA [1], spearheaded by the Object Management Group (OMG), provides a basis for portable distributed object-oriented computing applications [2] and is a marriage of the object-oriented paradigm with a client/server architecture [3]. Before the advent of CORBA, clients had to resort to sockets or remote procedure calls such as those provided by DCE or Sun RPC to communicate with objects on remote servers [6, 7]. CORBA abstracts away the complexities of distributed object communication between clients and servers.

The Event Service, one of the Common Object Services (COS) [1] provided by CORBA, is intended to support de-coupled, asynchronous communication among objects. For many applications, a synchronous communications model will not scale well. An elegant solution for these applications is to have the server push messages into a "channel" and let any interested client connect to the channel and pick up the messages. In Event Service parlance, the server is called a *supplier*, the clients are called *consumers*, and the channel is called the *event channel*.

The Event Service however is not reliable. This is because the CORBA Event Service specification does not deal with QoS (Quality of Service) or run-time management issues. There exist applications such as Telcordia's mission critical network management system for which reliable, in order message delivery is important. In addition, these applications should also be resilient to failures of the supplier, consumer and the Event Service itself. In an attempt to remedy this situation, OMG has come out with the Notification Service specification [5] that explicitly deals with QoS issues. But the Notification Service implementation is not widely available for all ORBs (Object Request Brokers) and also there is no commitment from vendors to make an implementation available for several platforms. Moreover, the Notification Service does not guarantee against loss of messages, as there is no pro-active monitoring.

Fault tolerance via replicated objects, such as in [4] provides some resilience in the face of event channel failures. In this approach, suppliers retain a copy of messages pushed in to the event channel in a backup queue. But this scheme could still drop messages, as it does not prevent queue overflows. Also the use of ORB-specific API's prevents the adaptability of the scheme to other ORBs.

2 Log files and retry policies – Are they adequate?

Vendors have sought to provide resilience by providing additional features in their Event Service implementations. These include, a retry mechanism that uses a log file to deal with failed message deliveries or messages displaced from overflowing queues. The ordering of messages is no longer guaranteed. This scheme is also not guaranteed to work as the log files might reach their maximum size limit. There is no programming or administrative interface to monitor the queues. Some vendors have mechanisms to deal with object failures (for example, the Visibroker [8] ORB has an Object Activation Daemon to provide some resiliency). But this is not a CORBA standard and hence implementation-dependent.

3 Application-level reliability mechanism to provide resilience

In an experiment that we setup, we found that the maximum supplier throughput is 49.9 messages per second, if no message loss is to occur. Having identified the lack of adequate guarantees in the CORBA Event Service specification, we investigated application-level reliability mechanisms at the supplier and the consumer.

3.1 Model for reliability: Resynchronization

Many applications for which the Event Service is applicable need to propagate messages that resemble status updates. If and when messages are detected to be missing, what is required is not the history of changes but the current status. The same is true after failures of the supplier, consumer or the Event Service lead to message

loss. Our reliability scheme illustrated in Figure 3 exploits this property and is based on resynchronization of states.

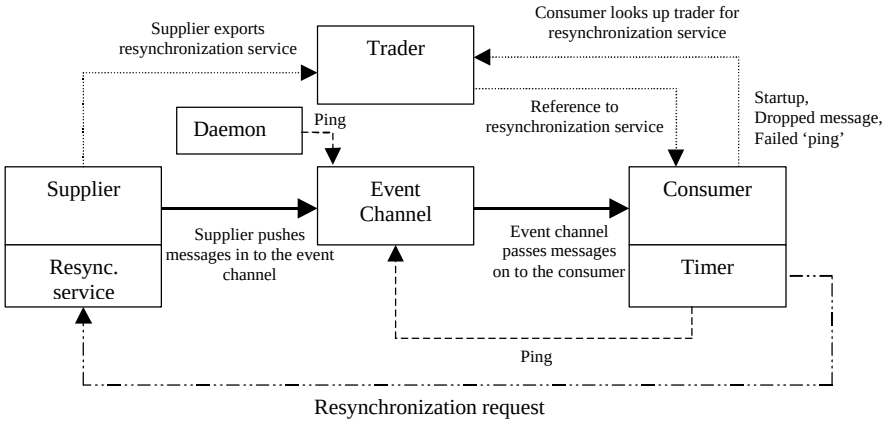

Fig. 1. Scheme to provide resilience to the CORBA Event Service

With this scheme, no log/retry mechanism of the Event Service is used, nor is the rebinding of the restarted objects dependent on any implementation-dependent facility. The highlights of our application-level reliability scheme are:

- When the supplier connects to the event channel for the first time or is restarted after a failure, it does a resynchronization to bring all the consumers up to date. The supplier puts some sequencing information in each message.
- The supplier provides a "resynchronization request service" that it announces through the Trader Service[1]. This service is to be used by consumers to send a resynchronization request to the supplier whenever required.
- When a consumer connects to the event channel – either for the first time or when it comes back up after a failure, it requests a resynchronization.
- When the consumer notices that a message has been lost (using the sequence information), it requests a resynchronization.
- A daemon to "ping" periodically and restart the Event Service if necessary.

We now explain how our scheme deals with failure scenarios.

Supplier failure:

When the supplier reboots, the recovery is simple – it simply resynchronizes.

Consumer failure:

When a consumer goes down, because of the de-coupled nature of the link between the supplier and the consumer, the supplier remains unaware of the failure. When the consumer comes up again, it reconnects to the event channel and requests a resynchronization. A consumer may, of course, choose not to act on the status updates, if it already has the latest status.

[1] The Trader Service (one of the COS) acts as the "yellow pages" for object services. The provider of a service exports the location and description of the service to a "trader". A client requiring a service can look up the trader by supplying the description.

Event Service failure:

To detect the failure of the Event Service, a daemon is used to "ping" the Event Service periodically. If the daemon notices that the Event Service has died, it restarts the Event Service. In a "push" model, a publisher notices that the event channel it is connected to has died when it tries to push a message, and tries to rebind with the new Event Service using the trader. The consumer can be unaware of the failure forever since in the push model, the Event Service initiates a message transfer. In our scheme, we make the consumer "ping" the event channel whenever a timer times out. The timer restarts whenever a message reaches the consumer. If the ping operation fails, the consumer tries to rebind to the event channel. The resynchronization strategy also obviates the need for mechanisms to deal with messages lost in the event channel.

Queue overflow:

Finally, if there is a queue overflow, the consumer will detect the dropped message(s) and request a resynchronization.

4 Effectiveness of the reliability mechanism - Experiments

In our earlier experiments, we determined that a supplier throughput of 49.9 messages per second or lower was needed to avoid message loss. We use this rough estimate while doing a resynchronization. Of course, if a resynchronization message were dropped, then the consumer would request a new resynchronization. The results in Table 1 report the mean and standard deviation for each measure, obtained by repeating each experiment 25 times.

Table 1. Effect of Supplier Speed on Resynchronization Overhead

Supplier throughput (#messages per sec)			% of normal messages received by consumer		#resynchronization messages pushed per 10000 normal messages	
Effective throughput		Normal messages				
Mean	S.Dev	Mean	Mean	S.Dev	Mean	S.Dev
265.6	29.8	247.8	26.3	10.5	580.9	224.7
98.1	1.3	96.9	96.1	3.9	122.4	127.0
49.9	0.0	49.9	99.9	0.1	2.0	5.0

Table 1 shows that when the supplier attempts a throughput higher than 49.90 messages per second, there are several resynchronizations before the supplier finishes pushing 10000 messages. Because of the additional resynchronization messages that are pushed and the overheads associated with the reliability mechanism, it now took longer to push 10000 normal messages. In Table 1, the column titled "normal messages" lists the supplier throughput for normal messages alone (that is, without considering the resynchronization messages). This is calculated as "*1 / (Total time to push 10000 normal messages (in sec)/10000)*". If the supplier throughput is substantially higher than 49.90 messages per second, then it can be seen from the table that a low percentage of these normal messages eventually make it to the consumer. This means that the (slower) resynchronization process is doing most of

the status updates. As the supplier throughput approaches the recommended value, the effective throughput of the supplier matches the throughput of normal messages from the supplier. This is because almost all the normal messages are successfully delivered to the consumers and the resynchronization messages are rarely required.

The general guideline for effectively putting into practice our reliability scheme is as follows:

- Determine the supplier throughput that almost always delivers messages without loss in the event channel. Use a much lower value as the throughput for the messages.

For the particular application and hardware platform on hand, optimization studies can be done to establish safe and most efficient operating ranges.

5 Summary and Concluding Remarks

In this paper we have studied the suitability of the CORBA Event Service as a reliable message delivery mechanism. To deal with its lack of reliability, we examined application-level schemes to build a reliable communication means over the existing CORBA Event Service. The reliability schemes, which are general enough to be applicable to any Event Service implementation, deal with message losses and also provide resilience in the face of failures of the supplier, consumer, and the Event Service processes. Our end-to-end reliability scheme is suitable for applications for which status resynchronization is possible and sufficient.

References

1. Object Management Group. “The Common Object Request Broker: Architecture and Specification”. Revision 2.3, 1998. ftp://ftp.omg.org/pub/docs/formal/98-12-01.pdf.
2. M. Henning and S. Vinoski. Advanced CORBA Programming with C++. Addison Wesley, Reading, Massachusetts, 1999.
3. D. Pedrick, J. Weedon, J. Goldberg and E. Bleifield. Programming with Visibroker – A Developer’s Guide to Visibroker for Java. John Wiley & Sons, 1998.
4. T. Luo et al. “A Reliable CORBA-Based Network Management System”. *Int. Conf. on Communications, ICC99*, Vancouver, BC, Canada, June 1999.
5. Object Management Group. “Notification Service: Joint Revised Submission”. January 25, 1999. http://www.omg.org/cgi-bin/doc?telecom/98-11-01.pdf.
6. J. Shirley, W.Hu and D. Magid. Guide to Writing DCE Applications. O'Reilly & associates, Inc., May 1994.
7. W.R. Stevens. UNIX Network Programming, Vol. 1, Second Edition, Prentice-Hall, Upper Saddle River, New Jersey, 1998.
8. Visibroker. http://www.inprise.com/visibroker

Acknowledgements

Our thanks to Mark Segal, Brian Coan, Michael Skurkay, Neal Bickford and Sarah Tisdale of Telcordia Technologies for their support and review of this paper.