Reliability with CORBA Event Channels

Xavier Défago Pascal Felber Benoît Garbinato Rachid Guerraoui
Laboratoire de Systèmes d'Exploitation
Département d'Informatique
École Polytechnique Fédérale de Lausanne, Switzerland
e-mail: {xavier,pascal,benoit,rachid}@lse.epfl.ch

1 Overview

Several application domains such as finance, process control, and telecommunications, have strong reliability requirements. Typically, such applications tend to avoid having a single point of failure, and need to communicate with reliable primitives that prevent message loss and ensure atomicity guarantees. Among such applications, we have focused on reliable *notification-based* applications, such as trading systems and news agencies, where *producers* need to *reliably* deliver information to a set of *consumers*. Developing such applications is greatly eased with a middleware providing *reliable broadcast* semantics [7].

Group-oriented systems like Isis [3], Horus [9], Totem [2] or Transis [1], provide reliable broadcast primitives and are generally considered to be good candidates for implementing reliable notification-based applications. Nevertheless, these systems are proprietary solutions with limited portability and interoperability. Although efforts have been made recently to achieve better modularity (e.g., in Horus), the infrastructure of group-oriented middlewares usually consists of several layers that are not necessarily required at upper levels and usually turn out to be performance penalizing.

Orbix+Isis [8] is an effort at supporting replication of CORBA objects transparently, by integrating Isis in Orbix. This approach requires a modification of the ORB and leads to a non-standard and non-interoperable solution. The Object Group Service [6] provides replication of CORBA objects without using heavy-weight group communication toolkits (e.g. Isis) and would provide the degree of reliability required by our application class. The tradeoff is performance degradation since it introduces replicated intermediary objects.

We present here a way to augment CORBA with a reliable broadcast facility. Our approach is pragmatic in the sense that it requires no modification of the Object Request Broker, and we do not build a new CORBA service from scratch. Instead, we add reliability features to the existing CORBA Event Service, which already provides multicast-like communication. The extension we introduce requires no modification of the CORBA specification, and can be applied to any standard Event Service implementation, without any communication overhead. The resulting service, called *Reliable Event Service*, adequately fits the required semantics of reliable notification-based applications. It constitutes an interesting light-weight and open alternative to existing group-oriented systems.

2 Reliability Issues

We consider notification-based applications where communication is decoupled between consumers and suppliers of information, with specific reliability requirements. This type of applications is widespread in domains like process control, finance, or telecommunications.

The use of CORBA for such applications bears many advantages over other approaches. The portability and interoperability aspects of CORBA are strong assets. The paradigm offered by the event channels is well adapted to notification-based applications since it provides a flexible model for asynchronous communication among distributed objects. Furthermore, relying on one-to-one communication to implement this functionality would require some amount of bookkeeping to keep track of the consumers. Finally, depending on the implementation, there is a potential for the Event Service to be scalable while it is clearly not the case with one-to-one communication primitives.

2.1 Limitations of the Event Service

Since the Event Service is based on a centralized architecture, where a channel is just another CORBA object, it introduces a single point of failure. Furthermore, the CORBA specification is vague concerning the quality of service provided by event channels. It states that the Event Service does not need to provide stronger semantics than "best-effort" delivery of the events, although implementors of the Event Service are advised to provide various semantic levels for their channels. The problem of the centralized architecture of the event channels may in two ways:

- *Replicate the event channels.* Event channels are replicated, and hence, are no longer a single point of failure. This approach, used in Isis News [3], requires the use of specific protocols, like group communication [4], to keep replicated objects consistent.
- *Decentralized architecture.* In a decentralized architecture an event channel is not implemented as a single physical object but rather as a collection of collaborating objects. This approach makes it possible to build a protocol based on IP-multicast rather than point-to-point communication, thus improving efficiency and scalability.

A solution to the lack of clearly specified semantics requires the definition of a standard quality of service to be expected from any implementation of the Event Service and a standard way to select it. The specification may define different levels of quality of service, from which the application programmer may choose. Currently, a valid implementation of the Event Service needs to be at least "best-effort". In other words, it puts no actual requirement on the delivery semantics since "best-effort" is a subjective description rather than a real property. Since a vague and minimal description is not suitable for reliable applications, we describe a protocol in Section 3 that extends any Event Service to make it reliable.

3 A Reliable Event Service

We introduce here the Reliable Event Service that provides *reliable* event channels, by extending the quality of service of any existing (unreliable) Event Service. The approach we adopted provides the exact quality of service required by the application class considered, and focuses on providing good performances. Furthermore, it is orthogonal to the architecture (centralized/decentralized/replicated) of the Event Service that it extends.

The semantics we associate with the Reliable Event Service are close to those of a Reliable Multicast primitive [7]. This primitive ensures that different clients receive the same set of messages. An informal definition of this primitive could be the following: if a correct object multicasts a message m, then all correct objects eventually deliver m. Furthermore, if a correct object delivers a message m, then m was previously multicast by some object and all other correct objects will eventually deliver m. Briefly, Reliable Multicast has two properties: *at-most-once* and *atomicity* (*all-or-nothing*).

Ideally, we would use a reliable multicast primitive, but its strong properties have a very high cost in terms of communications overhead. In a typical implementation of this primitive, the number of messages generated belongs to $O(n^2)$ and it requires that each consumer keeps a list of all the other clients. In the context of a diffusion network (e.g., Ethernet) where the complexity of a multicast is $O(1)$, the complexity of the reliable multicast is still $O(n)$. Since the cost increases proportionally with the number of destinations, it is not scalable.

Our mechanism has weaker properties than a Reliable Multicast, but it suits our requirements for reliability and does not change the complexity of the underlying communication. It is split into three parts. The first part consists in detecting when a message has been lost, in order to emit a notification. The second part helps to reduce the probability of actual loss by retrying unsuccessful transmissions. Finally, the last part ensures that messages are delivered in a FIFO manner.

3.1 Notification of Message Loss

Since there is no time bound on the delivery of messages, it is not possible to distinguish a lost message from a slow one. Hence, we consider the message to be lost in both cases.

To detect when a message is lost by the channel, we add some extra information to each message: a unique message identifier. Each producer has a unique identity given by its CORBA object reference. This tag makes messages issued by two different producers distinguishable. In order to differentiate messages issued by the same producer, we add a second field holding a local identifier (id). This id consists of a sequence number that is incremented each time a new message is sent. Therefore, clients will eventually detect lost messages based on missing sequence numbers. If the underlying event channels are not FIFO, the client may assume that a message is lost while it is only delayed. In that case, the client will launch the replay protocol (see below), and discard duplicate messages.

3.2 Message Replay

When a client detects the loss of a message, it contacts the producer by using the CORBA reference embedded in the message identifier. The client issues a request for the lost message using a synchronous remote method invocation and waits for a reply (see Figure 1). If the producer has not crashed in the meantime, the message will be resent and the client may continue. If a problem occurs (e.g., the producer has crashed) the reply is an exception and the client is supposed to react adequately. This approach is actually based on the principle of negative acknowledgments. In order to be able to resend a message, the producer needs to keep a buffer with every message it sends.

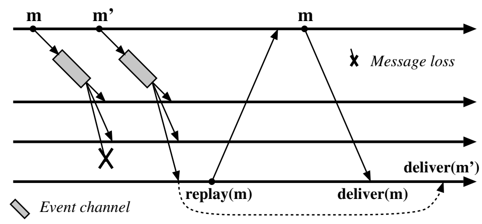

Figure 1: Replay of a lost message.

When the loss of a message is not recoverable, the most sensible approach consists in issuing an exception to the handled by the application. Since the adequate reaction to such a loss varies from one application to another, we leave the responsibility of reacting properly to the application programmer.

3.3 Ensuring FIFO Ordering

Since our protocol is aimed at working with any implementation of the event channels, we face an additional problem. If the underlying protocol ensures that received events are delivered in the same order than they were sent (FIFO property), replaying lost messages breaks this property. Hence, to avoid this problem, we add a mechanism that guarantees a FIFO delivery of events. This mechanism, illustrated in Figure 1, is an adaptation of the FIFO multicast presented in [7].

We first need to distinguish the reception of a message from its delivery. We call $receive(m)$ the reception of the message m by the lower protocol layer, and $deliver(m)$ the delivery of the message m from the lower layer to the upper layer. In some situations (e.g. upon a message loss) a message m', sent after m, may arrive before m. In other words, $receive(m')$ precedes $receive(m)$.

In order to ensure the FIFO property, the delivery of m' is delayed until m has been received and delivered. This implies that the FIFO order of delivery is preserved for the upper layer. In other words, $deliver(m)$ precedes $deliver(m')$. The FIFO property is thus guaranteed by our protocol, whether or not the underlying communication channel delivers the events in a FIFO order.

3.4 Appropriate Reaction

When a message has been lost and is no more available, the client has to react accordingly. The most appropriate reaction depends on the application. A non-exhaustive list of possible reactions to the loss of a message is:

- *Ignore (trivial case)*. The lost message is ignored. There was no need for our protocol and reliability is not necessary.
- *Quit*. The client is considered faulty, and hence, decides to commit suicide.
- *Quit & Recover*. The client is considered crashed, but it subscribes again to the event channel, as if it were just starting to listen to the event channel. In the initialization phase, a producer may send initial information to the newcomer.
- *Warning*. A warning message is issued to the end-user, telling that some information might not be up-to-date.

To satisfy the needs of a large number of applications, the most sensible approach consists in issuing an exception whenever a message cannot be retransmitted. This leaves the responsibility of reacting properly to the application programmer.

In order to guarantee the atomicity of delivery, it is necessary for the client not to be considered correct when it fails to deliver a message. Therefore, the only reactions that guarantee atomicity are *"Quit"* and *"Quit & Recover"*.

4 Implementation Issues

When evaluating the relevance of using a middleware for the development of notification-based applications, one of the main concerns is to rely on a standard definition rather than features specific to a particular vendor. Since these applications are expected to evolve over a long period of time, portability is a strong requirement.

Our current implementation suffers from a number of limitations inherent to the underlying Event Service that we use, i.e. IONA's OrbixTalk. In particular, it supports only the push model defined in the Event Service specification, and does not allow to chain event channels (i.e., there must be at most one event channel between a consumer and a supplier). More information can be found in [5]

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication sub-system for high availability. In *Proceedings of the IEEE 22nd International Symposium on Fault Tolerant Computing Systems*, 1992.

[2] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[3] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The Isis System Manual*. Dept of Computer Science, Cornell University, September 1990.

[4] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[5] X. Defago, P. Felber, and R. Guerraoui. Reliable corba event channels. Technical report, Département d'Informatique, EPFL, Switzerland, May 1997.

[6] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Proceedings of the IEEE 15th Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake, Canada, October 1996.

[7] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.

[8] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[9] R. van Renesse, K. Birman, and R. Cooper. The HORUS system. Technical report, University of Cornell, NY, 1993.