Reliability Issues with CORBA Event Channels

Xavier Défago
Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne (Switzerland)
defago@lse.epfl.ch

1 Overview

In order to increase flexibility and reduce complexity, the use of distributed platforms such as CORBA for building complex distributed systems is growing significantly. Although CORBA platforms provide a better support than conventional RPC mechanisms for deploying large distributed applications, few developments have been made so far to offer the necessary support for increasing reliability and fault-tolerance. We present here the behavior of inter-object communications in the CORBA reference, and compare it with the requirements of some type of applications, by focusing on reliability issues. Then we show how it is possible to bridge the gap between the specification and our requirements, when considering the problem of a server object generating information for a number of independent clients.

The paper is organized as follows: Section 2 overviews the CORBA specification and describes the basic properties of two of its communication models. In Section 3, we analyze the semantic requirements of a family of applications. We show in Section 4, how we can increase the reliability of communication and thus meet our requirements. Finally, Section 5 is dedicated to our observations on a current implementation of the event channels.

2 CORBA Event Service

Distributed object systems are becoming widely available and are becoming a preferred way to deal with distribution in complex systems. The modularity and flexibility offered by the object-orientation of the approach is a key factor when it comes to build evolving and complex applications. This section presents the basic concepts and properties of the CORBA architecture.

2.1 Overview of CORBA and its Event Service

A comprehensive description of CORBA would go far beyond the scope of this article, and many good books can be found on this topic [OPR96, MZ95]. Nevertheless, let us briefly describe the underlying concepts of this approach to distributed computing.

The central component of the architecture is the Object Request Broker — usually called ORB — whose principal role consists in transparently relaying requests to objects, across the distributed environment. CORBA is the specification of a standard interface for ORBs which also includes the definition of a set of basic services such as global naming or event notification.

In this article, we consider two different types of communication offered by CORBA. As shown in Figure 1, the *remote object invocation* consists in calling the method of a remote object through the ORB. The concept is very similar to a conventional RPC, and is even considered as such. The main difference lies in a smaller granularity than what conventional RPCs usually allow, as it acts on objects rather than processes[1].

The *event channels* are part of the CORBA event service and have very little to do with RPCs. The abstraction is that of a buffer with an arbitrary number of producers and consumers — also called clients. Oblivious of the clients, producers put messages into the channel and continue their execution.

[1]A single process can hold numerous objects.

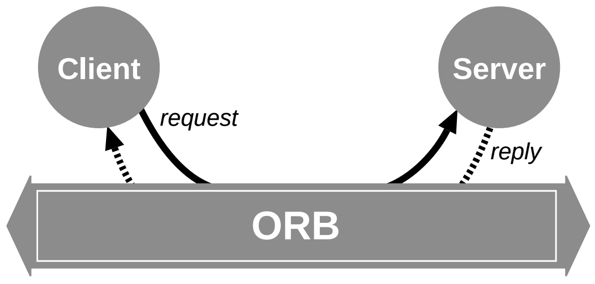

Figure 1: Remote method invocation.

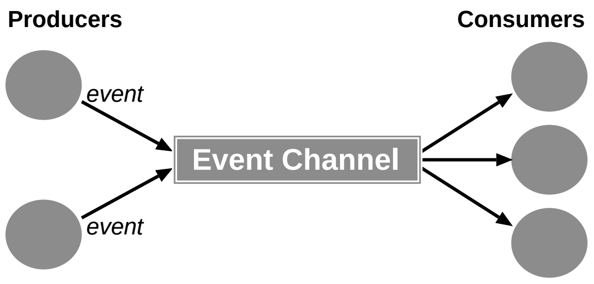

Figure 2: Event channel.

The messages are distributed to every client listening to that channel. From the point of view of the clients, the channel acts as a single producer which produces the messages. This mechanism, illustrated in Figure 2, is normally used for event notification but, as we will show later, it also provides a nice abstraction to the problem of multicasting information.

2.2 Properties

As mentioned before, the main goal of the CORBA specification [OMG95, OMG96] is to enforce ORB vendors to provide a common interface, thus imposing a standard way to deal with distributed objects in general. In order to satisfy the needs of as many kinds of applications as possible, the requirements put on the semantic of communication have been kept minimal. It is left to the implementors of ORBs to provide the different levels of semantic adapted to the various types of applications they want to support.

From the point of view of the application programmer, the right level of semantic might not be available. Hence, it might become necessary to rely only on the minimal set of guarantees offered by the CORBA specification. In this section, we present the properties that any ORB needs to ensure, for both the remote object invocation and the event channels.

2.2.1 Remote method invocation

We need to consider two types of invocation; *synchronous*[2] and *one-way*. The first type of communication corresponds to the normal case, where a client invokes the method of a server object and waits for a reply. Additionally, some methods can be declared as `oneway` and no reply is expected from the caller.. This distinction is important as the semantic guaranteed for synchronous method calls is slightly different from the semantic guaranteed for one-way methods.

When a synchronous method call is performed, a success always means that the invocation completed and was handled exactly once on the remote object. But whenever an exception is raised during the call, the semantic of execution is only guaranteed to be *at-most-once*. Altogether, this means that a synchronous remote method invocation has an at-most-once semantic, extended by some information on the potential failure of the operation. Finally, synchronous method calls issued by the same client are guaranteed to be processed in a FIFO order.

One-way method invocations have a weaker semantic than synchronous calls. The semantic is also *at-most-once* but, unlike synchronous calls, there is no way for the sender to know whether the call was successful or not. Successful one-way method calls are also guaranteed to be processed in a FIFO order.

2.2.2 Event channels

The event service provides a somewhat different approach to communication than the standard remote object invocation. The abstraction offered by the event channels allows to decouple the information producers from the clients. Furthermore, event channels are inherently asynchronous and propagate events in a FIFO manner.

The specification of the event channels [OMG96] does not define the semantic of message delivery, and clearly states that an implementation of the event service does not need to provide a stronger semantic than "at-most-once" delivery of the events. For efficiency reasons, implementors of the event service are

[2]The *deferred synchronous* type of invocation commented in the CORBA specification has the same semantic than a *synchronous* call; only their interface differ.

advised to provide various levels of semantic for their channels. The application programmer can then select the most appropriate semantic for each channel used in the application.

3 On the reliability of Event Channels

As mentioned in the previous section, the specification of the event channels does not prevent a message to be lost for a subset of the involved clients.

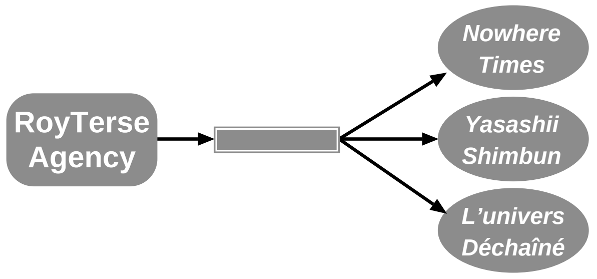

Figure 3: News agency sending news.

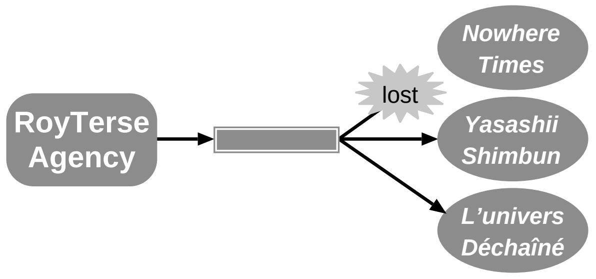

Figure 4: Situation to avoid.

Let us now consider the server of a news agency illustrated in Figure 3 sending news to its customers, using an event channel. In the editorial office of newspapers, a client listens to the event channel and prints out the news issued by the agency. But, if the event channel is allowed to lose messages, one of the customers might miss a very important scoop. There are three ways the information can be lost.

- The message cannot be delivered due to a malfunction of the client. This problem is clearly the responsibility of the client and neither channel nor server can do anything about it.
- The message can get lost, due to a malfunction of the server. This problem is a severe failure but has nothing to do with the communication channels. The risk can be reduced by replicating the server.
- As shown in Figure 4, the message can be lost by the communication channels. It might be delivered to some customers and not to others. This is a source of problems with the client, as it shows an unfair treatment between concurrent clients.

In this type of applications, we see that a slightly stronger semantic on the delivery of messages is needed. Provided server and clients do not fail, we want every message put into the channel to be received by every correct client.

Furthermore, we need to consider a few additional constraints. The pertinence of a message depends directly on the age of the information it conveys; as it gets older, a message gradually becomes useless. Hence, the need for the information to be made available[3] as soon as possible.

We assume that normal clients are able to cope with the rate at which messages are issued by the producer. The case of a very slow client is considered abnormal — and assumed to be rare — and can only be tolerated if it does not deteriorate the performance of the whole system and hinders others. In the context of this application we consider such a slow client to be faulty, which leads to the choice we make in Section 4 of killing it. Note that the buffering done by the event channels helps to mask punctual problems of the client, hence contributes to reduce the likeliness of a faulty client.

3.1 Reliable multicast

In a conventional distributed environment, we would use a Reliable Multicast primitive [HT93] in order to make sure that different clients receive the same set of messages. An informal definition of this primitive could be the following. If a correct process[4] multicasts a message m, then all correct processes eventually deliver it. Furthermore, if a correct process delivers a message m, then m was multicasted by some process

[3]The communication scheme that we consider, does not allow us to make any assumption on the time it takes for a message to reach its destination. In other words, we consider the underlying network to be asynchronous.

[4]Reliable Multicast was defined in terms of process but we could easily transpose this to objects or agents.

and all other correct processes eventually deliver m. The semantic of delivery is also "at-most-once" but, we introduce here a property of atomicity on the delivery of messages — an *all-or-nothing* property.

3.2 Requirements

If we still consider our news distributor application, we can clearly see the need for atomicity. Even though such a property is difficult to ensure in practice without a considerable overhead, we will see that a slightly weaker property can be enough for the type of application we consider.

Most of our problems actually reside in that event channels can lose messages oblivious to both the producer and the customers. If we now assume that customers get a notification whenever an event is lost by the channel, it becomes possible for them to react. We will show in the next section how to implement such a property and how it can help to enhance the reliability of event channels.

4 Design Issues for reliable Event Channels

Our approach to increasing the reliability of event channels can be decomposed into four parts. The first step consists in detecting when a message has been lost, in order to emit a notification. The second step aims at making the delivery of messages atomic. The third step helps reduce the probability of actual loss by retrying unsuccessful transmissions. The last step ensures that event channels remain FIFO.

4.1 Notification of message loss

In order to be able to detect when an message has been 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 is the first part of our tag as it makes messages issued by two different producers distinguishable. In order to identify two different messages issued by the same producer, we add a second field holding a local identifier (id). This id consists in a 64 bits sequence number[5] that is incremented each time a new message is generated. Therefore, due to the FIFO property of the event channels, clients will detect lost messages based on missing sequence numbers.

4.2 Atomic delivery

In order to satisfy the property of atomicity, we need to ensure that no correct client will ever miss a message that has been received by other correct clients. We can deal with this in a simple way; when a client is notified that it lost a message, it quietly passes away. Due to the crash of the client, the property of atomicity is satisfied as it was not a correct client — this is just playing on the cause-effect relationship.

This is an approach, which assumes that the probability of message loss is very low. The choice of killing a client is justified by the assumptions we made in Section 3 and other types of application might choose a different reaction[6]. Nevertheless, as we see later, this still needs to be refined in order to be usable in a practical case.

If the network traffic is not an issue and the number of customers is expected to be quite low, then we can implement the protocol for the reliable broadcast described in [HT93] which represents a pessimistic approach.

4.3 Message replay

As we saw in 4.2, a message loss implies that we need to kill the client in order to satisfy the atomicity of delivery. Even though the probability of message loss in a local area network should usually be close to 0.01%, this should be deemed unsatisfactory. Considering the practical fact that physical resources are finite in nature, we are facing an unavoidable trade-off between slowing down the producers, increasing the traffic on the network or killing slow clients. In order to keep the philosophy of the event channel, where producers are oblivious of the clients, we decided to drop slow clients rather than require the producer to wait or increase the network traffic. This choice was motivated by the need of our server not

[5]A 64 bit counter is enough for a number of messages equal to what would be generated after 585'000 years if the time to generate a single message is 1μs, which is already 10'000 times faster than what we could measure.

[6]In order to keep the atomicity, the client cannot deliver any other message.

to depend on slow clients and by the potential opportunity for the client to recover. We can clearly see our need to maximize the probability for a client to get its message, and this is exactly what we propose to do here.

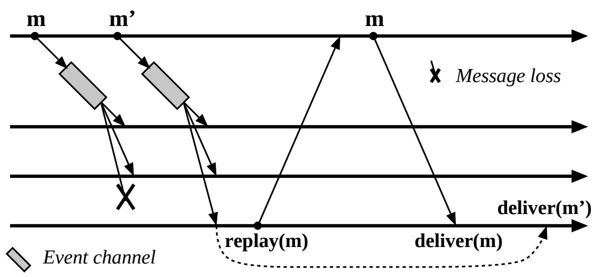

Figure 5: Replay of a lost message.

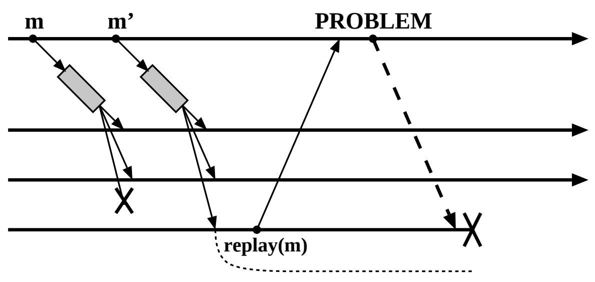

Figure 6: Loss is not recoverable.

When a client detects that a message has been lost, it contacts the producer by using the CORBA reference embedded in the message identifier. It issues a request for the lost message using a remote method invocation and waits for a reply (see Figure 5). If the producer has not crashed in the meantime and the message is still available in its send buffers, the message will be resent and the client can continue. If a problem occurs — the message is not available or the producer has crashed — the reply will be an exception and the client will die, as shown in Figure 6.

As we can see, this mechanism only serves to reduce the probability of a customer dying because of a message loss. This choice assumes that a customer is better dead than inconsistent.

4.4 Keeping the FIFO property

So far, our event channels gained the property of atomicity on the delivery of messages but, on the other hand, the FIFO order property is not valid anymore. This is a very serious matter for it breaks the very definition of the event channels. This can actually be solved in the same manner than FIFO multicasts are defined in terms of reliable multicasts [HT93].

If we distinguish the reception of a message and its delivery[7], the mechanism is quite easy to explain. When a message m is lost, the delivery of the next message m' is delayed until m is successfully delivered.

5 Implementation issues

This article was motivated by some observations that we have made while building a prototype for an application similar to the news agency server we presented in Section 3. This prototype aims at evaluating the relevance of using the CORBA standard exclusively for our target application. One of the main concerns is to rely on a standard definition rather than features specific to a particular vendor. The application is expected to evolve over a long period of time.

We have built this prototype using Orbix™[8] as our ORB, and OrbixTalk™for its full implementation of the Event Service.

OrbixTalk™ is supposed to implement a reliable multicast protocol [ION96] with a semantic that corresponds to our requirements. Some measures showed that it was not the case and we could observe an average loss of one out of 25'000 messages, without a notification[9]. We discovered that it was nevertheless conform to the CORBA specification of the Event Service, hence our motivation to design a protocol that would suit our needs and not depend on a specific implementation of the event service.

6 Conclusion

This paper has presented a way to increase the reliability of communications when using the CORBA event channels. While working on the prototype of an application where the reliability of the diffusion

[7]A message is *received* from the lower protocol layer and *delivered* to the upper layer.
[8]Orbix™and OrbixTalk™are trademarks of IONA Technologies Ltd.
[9]IONA was working on fixing this bug at the time this paper was written.

of information was a primary concern, we found that the guarantees made by the CORBA specification were not enough. We have defined a mechanism that increases the reliability of these communications and which is likely to meet the requirements of many other distributed applications.

We strongly believe that, despite their weak semantic, CORBA Event Channels provide a good basis for building complex and reliable distributed systems.

References

[HT93] Vassos Hadzilacos and Sam 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.

[ION96] IONA Technologies Ltd. *OrbixTalk Programming Guide*, July 1996.

[MZ95] Thomas J. Mowbray and Ron Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. John Wiley & Sons, Inc, 1995.

[OMG95] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1995. Revision 2.0.

[OMG96] Object Management Group. *CORBAservices: Common Object Services Specification*, July 1996.

[OPR96] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: The Common Object Request Broker Architecture*. Prentice Hall, 1996.