Reliable CORBA Event Channels

Xavier Défago Pascal Felber Rachid Guerraoui
Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Switzerland

Abstract

This paper presents a pragmatic way to build a Reliable CORBA Event Service. Our approach is pragmatic in the sense that, rather than building the service from scratch, we show how to obtain it, through a simple transformation, from any standard (unreliable) CORBA 2.0 Event Service. Our extension does not introduce any modification to the CORBA specification, nor any communication overhead. The Reliable CORBA Event Service provides the adequate semantics for building reliable *notification-based* applications, and an interesting light-weight and open alternative to existing group oriented systems.

1 Introduction

There are several areas, such as process control, finance, and telecommunications, where applications have strong reliability requirements. Typically, such applications tend to avoid having a single point of failure, and are distributed over different nodes communicating through reliable primitives that prevent message loss and ensure atomicity guarantees. Among such applications, we focus in this paper on reliable *notification-based* applications, such as trading systems and news agencies. These applications have a *publish/subscribe* semantics where *producers* need to *reliably* deliver information to a set of *consumers*. Developing such applications is greatly facilitated with a middleware whose communication primitives provide *reliable broadcast* semantics [8].

Group oriented systems like Isis [4], Horus [15], Totem [3] or Transis [2], provide reliable broadcast primitives and are generally considered as good candidates for implementing reliable notification-based applications. Nevertheless, these systems lead to proprietary solutions with limited portability and interoperability. Although efforts have been made recently to achieve better modularity (e.g., in Horus), the group oriented infrastructures usually contain several layers that are not necessarily required at upper levels. For instance, all group oriented systems that we know about rely on a group membership service, which for certain type of applications (e.g., notification-based applications) turns out to be useless and even performance penalizing.

In this paper, we explore the use of more open and modular middleware for the development of notification-based applications. More precisely, we evaluate the adequation of CORBA to provide reliable publish/subscribe semantics, and we show to overcome some of its reliability limitations.

CORBA is an object-oriented computing middleware standard, defined by the Object Management Group (OMG), that supports the production of flexible and reusable distributed objects communicating independently of the specific platforms and techniques used for their implementation. CORBA provides the basic mechanisms for remote invocation through the *Object Request Broker (ORB)*, as well as a set of services for object management, e.g., Persistence Service, Naming, Event Service, Life Cycle [14]. Nevertheless, neither the ORB nor the existing services provide tools for building reliable and highly available applications. In particular, no reliable broadcast primitive is provided in CORBA.

We present in this paper a way to augment CORBA with a reliable broadcast facility. Our approach is pragmatic in the sense that no modification of the Object Request Broker is necessary, 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 Event Service CORBA 2.0 standard implementation, without 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.

The remainder of this paper is structured as follows. Section 2 recalls the CORBA model, focusing on the CORBA Event Service. Section 3 discusses the adequation of the Event Service abstraction to notification-based applications, and points out its reliability limitations. Section 4 presents our extension to the standard Event Service, and describes the resulting Reliable Event Service. Section 5 discusses implementation issues and presents some performance measures. Section 6 compares our approach with related work and Section 7 summarizes the main contribution of the paper.

2 CORBA: Background

2.1 The OMA

The *Object Management Architecture (OMA)* [7] is a framework defined by the Object Management Group (OMG), which provides a conceptual infrastructure for building inter-operable, reusable, portable[1] software components based on open, standard object-oriented interfaces.

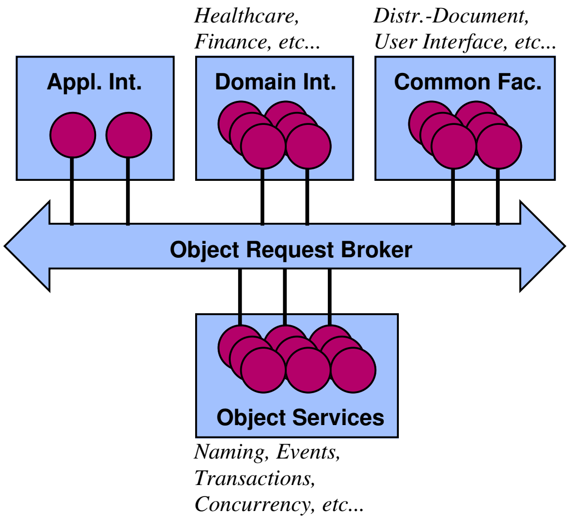


Figure 1: The OMA Architecture

Figure 1 shows the five major parts of the OMA reference model. The *Object Request Broker* (ORB) enables objects to transparently invoke remote operations and receive replies in a

[1] *Portability* means here the ability to use an implementation with different ORBs (by simply recompiling it) while *interoperability* means the ability of an implementation to cooperate with other implementations.

distributed environment. The *Object Services* are a collection of interfaces and objects supporting basic functionalities useful for most CORBA applications. The *Common Facilities* are a collection of interfaces and objects providing end-user-oriented capabilities useful across many application domains. The *Domain Interfaces* are meant to be used only in specific vertical application domains. Finally, the *Application Objects* are objects specific to end-user applications.

CORBA defines the notion of compliance for a distributed application or system. A client or server is said to be *CORBA compliant* if it relies only on the CORBA specification. An ORB implementation conforms to the specification if and only if it correctly executes any *CORBA compliant* application.

The ORB and the Services

The Object Request Broker can be viewed as an "object bus". CORBA was designed to allow heterogeneous components to interoperate through this bus. Integration of distributed objects is available across platforms, regardless of networking transports and operating systems. Each component interface is specified in the OMG *Interface Definition Language* (IDL), which is implementation independent. Clients use *object references* to identify remote objects and invoke operations on them. Objects are not tied to a client or server role: they can act both as client and as server.

Beside the ORB itself, the CORBA services are of particular interest to us. A service is basically a set of CORBA objects with their corresponding IDL interfaces, and these objects can be invoked through the ORB by any CORBA client. Services are not related to any specific application but are basic building blocks, usually provided by CORBA environments. Several services have been designed and adopted as standards by the OMG. Among these services are the Life Cycle Service, used for creating and deleting objects, the Persistence Service, used for storing the objects on persistent storage, and the Transaction Service that lets multiple distributed objects participate in atomic transactions.

CORBA Communication

A standard CORBA request — remote method invocation — results in the synchronous execution of an operation by an object. If the operation defines parameters or return values, data is communicated between the client and the server. A request is directed to a particular object. For the request to be successful, both the client and the server must be available. If a request fails because the server is unavailable, the client receives an exception and must take some appropriate action. This model is illustrated in Figure 2.

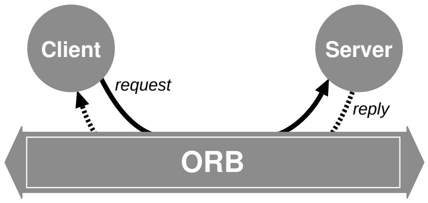


Figure 2: Remote method invocation

A remote method invocation is successful if the method is actually executed and returns a reply. If a user exception is raised during method execution, the invocation is also considered to be successful. Nevertheless, if the method cannot be invoked (e.g., a crash happens during its execution), the invocation is considered to be unsuccessful. If an exception is raised during the invocation, a hint concerning the completion of the invocation is available: *"completed"*, *"not completed"*, and *"indeterminate"*. The semantics of the three types of invocations are the following:

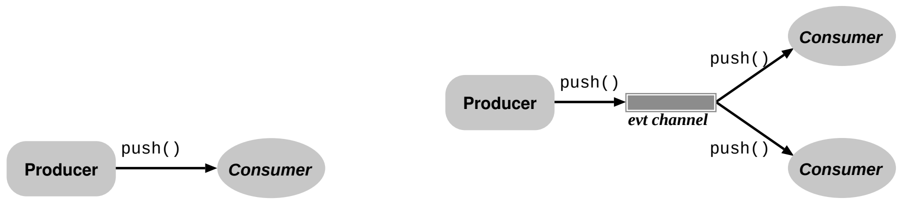
- *Synchronous invocation.* When a synchronous method call is performed, a success means that the invocation completed and was handled exactly once by the remote object. But whenever an exception with an *"indeterminate"* status is raised, the only guarantee is that the method was executed *"at-most-once"*. Altogether, this means that a synchronous remote method invocation has at-most-once semantics, extended by some information on the potential failure of the operation. Furthermore, synchronous method calls issued by the same client are guaranteed to be processed in a FIFO (first in first out) manner.

- *Deferred synchronous invocation.* The communication semantics of a deferred synchronous method call are the same as a synchronous one, i.e., successful operations are performed exactly once in a FIFO manner and operations resulting in an exception are performed *"at-most-once"*.

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

2.2 The CORBA Event Service

The Event Service decouples the communication between objects. It defines two roles for objects: the *supplier* role and the *consumer* role. Suppliers produce event data and consumers process event data. Event data are communicated between suppliers and consumers by issuing standard CORBA requests. Suppliers can generate events without knowing the identity of the consumers. Conversely, consumers can receive events without knowing the identity of the suppliers.



(a) without event channel (b) with event channel

Figure 3: Event Service Push Communication Model

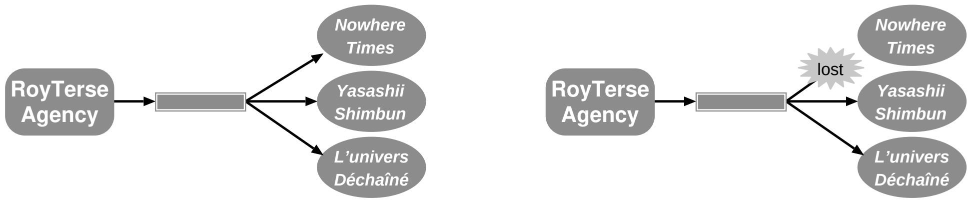
There are two approaches to initiating event communication between suppliers and consumers. These two approaches are called the *push model* and the *pull model*. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. In the push model, the supplier is taking the initiative; in the pull model, the consumer is taking the initiative.

An event channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events. Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests. Figure 3 illustrates the most widely used communication model, i.e., the push model.

3 Reliability Issues

3.1 An Example: News Agency

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



(a) News Agency Sending News

(b) Unwanted Situation

Figure 4: News Agency Example

A typical example is the *news agency* illustrated in Figure 4(a). The server sends news to its clients using a specific communication channel. In the editorial office of newspapers, a client listens to the news issued by the agency and prints them out. If some messages get lost, one of the newspapers may miss a very important information. We consider three situations in which the loss of a message may arise:

1. The message cannot be delivered due to a malfunction of the client. This problem is clearly the responsibility of the client and neither the communication channels nor the server can do anything to prevent it.

2. 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. The risk can be reduced by replicating the server [6].

3. As shown in Figure 4(b), the message can get lost by the communication channel. The message might be delivered to some clients and not to others. This is a source of problems as it shows an unfair treatment between competing clients.

The situation is similar for other notification-based applications. For instance, a trading system produces updates of the exchange rates. The traders subscribe to the service and are then aware of the evolution of the exchange rates.

3.2 Modeling the News Agency Using the Event Service

Modeling the News Agency example using the Event Service is straightforward. News communication channels are mapped to event channels, while news are mapped to events.

The use of CORBA for such an application 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.

3.3 Limitations of the Event Service

The Event Service is based on a centralized architecture, where a channel is just another CORBA object, and this 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. Implementors of the Event Service are advised to provide various semantic levels for their channels. The application programmer can then select the most appropriate semantics for each channel used in the application, using non-specified interfaces.

To solve the limitation concerning the centralized architecture of the event channels, two different approaches can be used:

- *Replicate the event channels.* Event channels are replicated, and hence, are no longer a single point of failure. This approach requires to use specific protocols that handle consistency of replicated objects, like group communication [5]. This approach is used in Isis News [4].
- *Decentralize the architecture.* A decentralized architecture implies that an event channel should not be implemented as a single object. For instance, it is possible in a local area network (LAN) to implement an event channel using an IP multicast address. Suppliers send messages using the multicast address of the channel, while consumers listen to the address of the channels they are registered to. This solution is very efficient in terms of performance, but has the drawback of making it difficult to chain event channels, and does not scale well to wide area networks (WAN).

Solving the limitation concerning the lack of clearly specified semantics implies to define a quality of service that is to be expected from an implementation of the Event Service, and how different qualities of service are requested. More specifically, an implementation of the Event Service may lose events and still comply to the specification. As mentioned in this specification, a valid implementation of the Event Service should 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. This policy is a real problem for the class of applications considered in this paper. We describe a protocol in Section 4 that tackles this problem by extending any Event Service to make it reliable.

4 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 (unreliable) Event Service that it extends.

The semantics we associate with the Reliable Event Service are close to those of a Reliable Multicast primitive [8]. Roughly speaking, 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. A typical implementation of this primitive consists in that each time a message is received by a client for the first time, it is multicasted to all other clients. This leads to a strong communication overhead since the number of messages generated belongs to $O(n^2)$ and it requires that each consumer keeps a list of all other consumers of the event channel. 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. In this section we present a mechanism with weaker properties, that suits our requirements for reliability and does not change the complexity of the underlying communication.

The main problem actually resides in that event channels may lose messages oblivious to both the producer and the consumers. If we assume that consumers get notified whenever an event is lost by the channel, it becomes possible for them to react. We show how to implement such a property and how it can help to enhance the reliability of basic event channels.

Therefore, we have implemented a communication protocol that provides stronger semantics than what is offered by current implementations of the Event Service. Our approach to increasing the reliability of the event channels takes efficiency and scalability issues into account. This approach 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.

4.1 Notification of Message Loss

Since there are no time bounds 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.

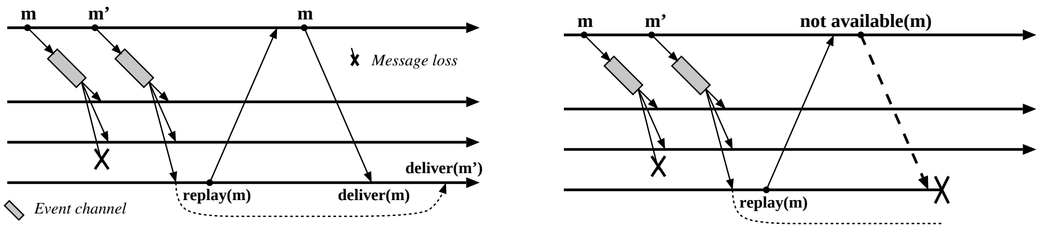
In order to detect the loss of a message by the channel, we add some extra information to each message: a unique message identifier. Each producer has a unique identity given by its CORBA object reference. This tag makes messages issued by two different producers distinguishable. In order to differentiate messages issued by the same producer, we add a second field holding a local identifier (id). This id consists of a 32 bit sequence number[2] that is incremented each time a new message is generated. Therefore, clients will eventually detect lost messages based on missing sequence numbers. If the event channels are not FIFO, the client may assume that a message is lost while it is only delayed. In this case, the client will launch the replay protocol (see below), and discard duplicated messages.

4.2 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 5(a)). If the producer has not crashed in the meantime, the message will be resent and the client may continue. As shown in Figure 5(b), 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. Nevertheless, considering the practical fact that physical resources are finite

[2] A 32 bit counter allows us to lose more than 4 billion consecutive messages.



(a) Replay of a Lost Message

(b) Loss is not Recoverable

Figure 5: Message Replay

in nature, we are facing an unavoidable trade-off between slowing down the producers with a positive acknowledgment system, increasing the traffic on the network with a reliable multicast primitive or, as described in Section 4.4, issuing exceptions to the slow clients.

4.3 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 5(a), is an adaptation of the FIFO multicast presented in [8].

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.

4.4 Atomic Delivery *(Application Dependant)*

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. This action ensures the atomicity of delivery since the death of the client implies that it was not correct. This is suitable for applications where the loss of a client is of little or no consequence.
- *Quit & Recover.* The client is considered crash 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[3].

- *Warning.* A warning message is issued to the end-user, telling that some information might not be up-to-date.

In order 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"*.

To summarize, this approach is flexible in the sense that it does not force a specific policy on the application. The level of reliability is then specified by the client.

5 Implementation Issues

In this section, we discuss implementation issues. We first evaluate the cost of retransmitting lost messages, and we discuss the issue of choosing an adequate size for retransmission buffers. Then, we present and comment throughput measures we made with our prototype implementation.

A prototype of our Reliable CORBA Event Service is based on the Orbix ORB [10], and OrbixTalk [11] implementation of the CORBA Event Service. OrbixTalk provides an implementation of the event channels based on IP multicast, which makes it quite efficient. Furthermore, the decentralized architecture of Orbix makes it potentially suitable for fault-tolerance.

5.1 Impact of Replaying Messages

We implemented a prototype with a producer that sends messages over an event channel and a consumer that receives these messages. Periodically, a message is lost by the producer[4]. The consumer detects that loss and asks for the message to be resent using a remote method invocation. As illustrated in Figure 6, we measured Δt_{replay}, which is the time interval between the emission of the message m and the reception of the request replay(m).

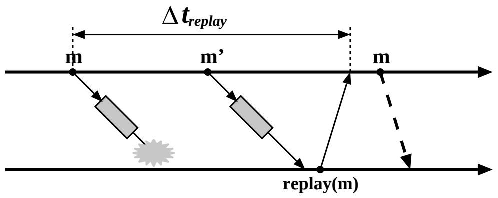


Figure 6: Measured Delay for Replaying Lost Messages

For retransmission, the producer keeps a buffer with the last messages it sent. The size of this buffer ($Buffer_{size}$) determines the maximum delay during which a message can be retransmitted (ΔT_{replay}). This delay also depends on the maximum throughput ($Thput_{max}$).

$$\Delta T_{delay} = Buffer_{size} / Thput_{max}$$

As a rough approximation, we expect Δt_{replay} to follow a gamma distribution [13]. We took 2000 samples for Δt_{replay}, observed the actual distribution and compared it with the predicted

[3] This corresponds to the state transfer found in group oriented systems such as Isis.

[4] We did a simulation where the producer just omits sending the message.

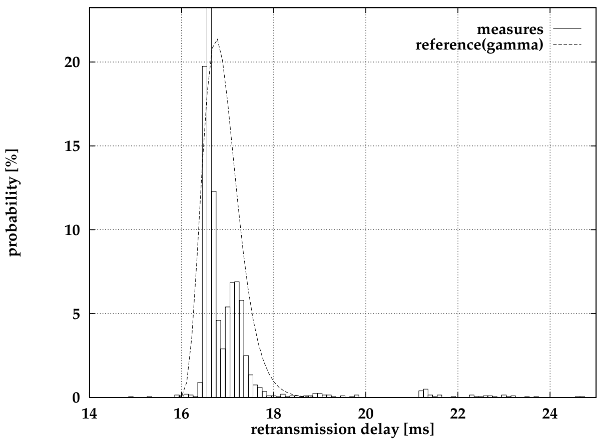


Figure 7: Predicted and Observed Distribution of the Retransmission Delay

distribution (Figure 7). This shows that values of Δt_{replay} are very concentrated but it also shows that the gamma distribution is only a rough approximation of the reality. For a more accurate model, we should consider a fractal based approach [12, 1], but this is beyond the scope of this paper.

Depending on ΔT_{replay}, the probability of a lost message to be successfully retransmitted is as follows:

$$P(receive) = (1 - P(loss)) \times P(\Delta t_{replay} \leq \Delta T_{replay})$$

With our set of measures, if the retransmission buffer is able to hold the last 3 messages, the probability of a message to be unavailable is $2.5 \cdot 10^{-3}$. We observed a probability of $4 \cdot 10^{-5}$ for a message to be lost by the network. With our mechanism, the probability to lose a single message without being able to retransmit it is $1 \cdot 10^{-7}$.

If a producer sends messages at a constant rate of 30 messages per second, for a duration of 8 hours, the probability that all messages reach their destination is 0.917 with our system. Without it, the same probability goes down to $9.78 \cdot 10^{-16}$! In this context, our mechanism just needs a buffer of 3 messages.

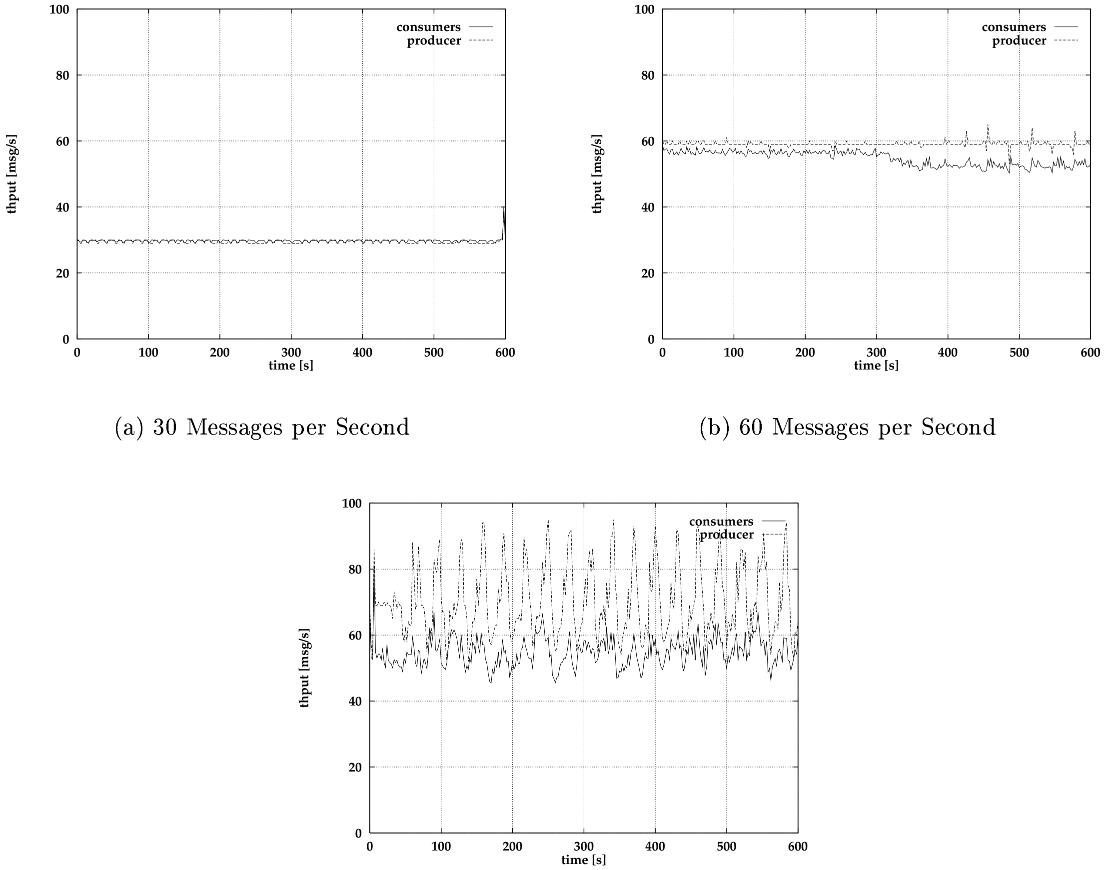
5.2 Throughput

We developed a test application with 10 consumers and one producer. The producer generates 1024 bytes messages at a fixed rate. The actual throughput at the producer and the consumers is measured over time. We took these measures over an overall period of 10 minutes.

In Figure 8(a), the producer generates 30 messages per second, and as shown in this figure, OrbixTalk copes with this rate. The traffic is very stable and no variation is observed.

When increasing the throughput to 60 messages per second (see Figure 8(b)) the customers cannot receive messages at the same rate and this leads to instabilities after a certain time. Finally, when increasing the throughput to 70 messages per second (in Figure 8(c)) the behavior becomes totally unstable. A non-exhaustive list of the potential reasons that may explain this oscillatory behavior is as follows:

- *Regulation*
  The regulation of the throughput is unstable. Depending on the conditions, a regulator may show an oscillatory behavior. Since the implementation of the regulator is rather straightforward, it might be prone to oscillations.



(a) 30 Messages per Second

(b) 60 Messages per Second

(c) 70 Messages per Second

Figure 8: Evolution of the Throughput

- *Process scheduling*
  The network and the process scheduling bear a stronger influence when the throughput increases. In other words, when the throughput increases, there is less time for the processes to react (send or deliver a message). The scheduling policy is not deterministic and do not guarantee a fixed level of responsiveness. The scheduling may induce bursts and therefore cause the system to vibrate.

- *Network Collisions*
  When the throughput increases, it reaches a level where the number of collisions causes many retransmissions, thus slowing down the producer. But, this forces the producer to emit messages at a quicker rate, when it tries to catch up.

6 Related Work

As mentioned earlier, although group oriented systems usually provide reliable broadcast primitives and are generally considered as good candidates for implementing reliable notification-based applications, they lead to proprietary solutions with limited portability and interoperability. The main difference with our Reliable Event Service lies in that a group oriented system provides much more than just a reliable multicast mechanism. This results in a significant amount of additional overhead for applications that only need to reliably multicast information. Furthermore, although it provides slightly stronger properties, the reliable multicast implemented in most group oriented systems is usually quite expensive in terms of communications, when compared to our approach. Finally, our Reliable CORBA Event Service is more flexible, since the final decision concerning the semantics of the system is left to the application programmer.

The Isis distributed news service [4] provides a facility similar to event channels. The service maintains a set of news "subjects" to which processes can post and read messages. Processes that post messages are providers of information, while processes that are interested in these subjects act as consumers. Isis News also provides a mechanism for message persistence.

The Isis distributed news service is implemented as a replicated news server that is invoked using the group communication primitives of the Isis toolkit. Therefore, the news service tolerates the failure of some of the news servers. In our model, this approach is similar to replicating event channels. It is a heavy-weight solution to reliable event notification, and news server replication augments the latency and degrades the performance of the system.

Orbix+Isis [9] is a product that integrates Orbix (IONA's implementation of CORBA) with the Isis distributed toolkit. It provides a CORBA interface to the Isis distributed news service.

The Object Group Service [6] provides replication of CORBA objects without using heavy-weight group communication toolkits (e.g. Orbix+Isis). It makes it possible for a group of CORBA objects to act as a single entity despite concurrent invocations and failures. Hence, it provides an adequate support for the construction of highly available distributed applications with replicated critical components. It would provide an easy way for replicating event channels, and thus provide the degree of reliability required by our application class. The tradeoff is performance degradation since it introduces replicated intermediary objects not required by a decentralized approach.

7 Conclusion

When evaluating the relevance of using a middleware for the development of a wide class of applications (e.g., notification-based applications like the news agency presented in Section 3), 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.

This paper explores the use of CORBA for the development of reliable notification-based applications. We present a way to build, on top of any existing CORBA Event Service, a Reliable CORBA Event Service, which adequately fits the required semantics of reliable notification-based applications. The extension we introduce does not require any modification to the CORBA specification, and can be applied to any Event Service CORBA 2.0 standard implementation.

Our current implementation suffers from a number of limitations inherent to the underlying Event Service that we use, i.e. 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).

References

[1] R. Addie, M. Zukerman, and T. Neame. Fractal Traffic: Measurements, Modeling and Performance Evaluation. In *Proceedings IEEE Infocom'95*, pages 977–984, Boston, MA, April 1995.

[2] 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.

[3] 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.

[4] 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.

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

[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] Object Management Group. *Object Management Architecture Guide.* John Wiley & Sons, Inc, 3rd edition, June 1995.

[8] 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.

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

[10] IONA Technologies Ltd. *Orbix-2 Programming Guide*, November 1995. Release 2.0.

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

[12] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the Self-Similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[13] A. Mukherjee. On the Dynamics and Significance of Low Frequency Components of Internet Load. *Internetworking: Research and Experience*, 5:163–205, October 1994.

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

[15] R. Van Renesse, K. Birman, and R. Cooper. The HORUS system. Technical report, University of Cornell (NY), 1993.