Asynchonrous Notifications Among Distributed Objects

Yeturu Aahlad, Bruce E. Martin,
Mod Marathe, and Chung Lee
SunSoft, Inc.

Asynchronous Notifications Among Distributed Objects

Yeturu Aahlad[1] Bruce E. Martin Mod Marathe[2] Chung Le

SunSoft, Inc.
2550 Garcia Avenue
Mountain View, California 94043 USA

Abstract

Distributed object systems typically support synchronous requests from one distributed object to another. Often, a more decoupled style of communication among distributed objects is appropriate. We describe an object service called *event channel* that decouples distributed object communication. We describe SunSoft's implementation of the event channel and illustrate its use in a stock trading application.

Keywords: distributed object-oriented systems, communication, events, OMG, CORBA, CORBA Services

1.0 Introduction

Distributed object systems[2][5] are emerging to support applications that access objects across distributed, possibly heterogeneous, system boundaries. Such systems define a distributed object model that is mapped appropriately to native concepts in a wide variety of systems.

Communication in distributed object systems typically consists of synchronous *requests*. A request is made by a requestor to a single target object. A request results in the synchronous execution of an operation by the target object. The requestor waits for a reply from the target. For the request to be successful, the requestor, the target and the network must be available. If a request fails because the target or network is unavailable, the requestor receives an exception.

The principal advantages of basing distributed object communication on synchronous requests are that it is easy paradigm for application programming and that it is an efficient and well-understood basic paradigm to implement in distributed object systems.

For many distributed applications, however, a more *decoupled* style of communication is appropriate. Instead of a requestor identifying the target, the requestor need not be aware of the target. Instead of a single target object, there may be multiple targets. Instead of waiting for a response or failure report, the requestor need not block while the targets process the request.

Consider the distributed stock trading application illustrated in figure 1. As stock prices change, stock objects simply supply the new prices to the distributed system. Interested objects consume the data. What the interested objects actually do with the data is really of no interest to the stock objects. One object may, for example, chart a graph of changes in the stock price. Another object may maintain a portfolio of stocks and make investments based on changes in stock prices. If the network partitions, the stock object does not want to be burdened with delivering the data to the interested objects.

A more decoupled communication paradigm between distributed objects allows applications to easily be extended without modifying existing objects. To extend the functionality of the application, new objects can be added that consume the data supplied by the stock objects. The stock objects need not be modified.

The topic of this paper is decoupling object communication in *CORBA-based distributed object systems* using the basic synchronous request paradigm of distributed object systems. There have been some studies related to decoupled communications in distributed systems.

---

1. Yeturu Aahlad's current address is Platinum Technology, Inc. Trinzic Lab. 555 Dolphin Drive, Redwood City, California 94065

2. Mod Marathe's current address is StrataCom Inc. 1400 Parkmoor Avenue, San Jose, California 95126

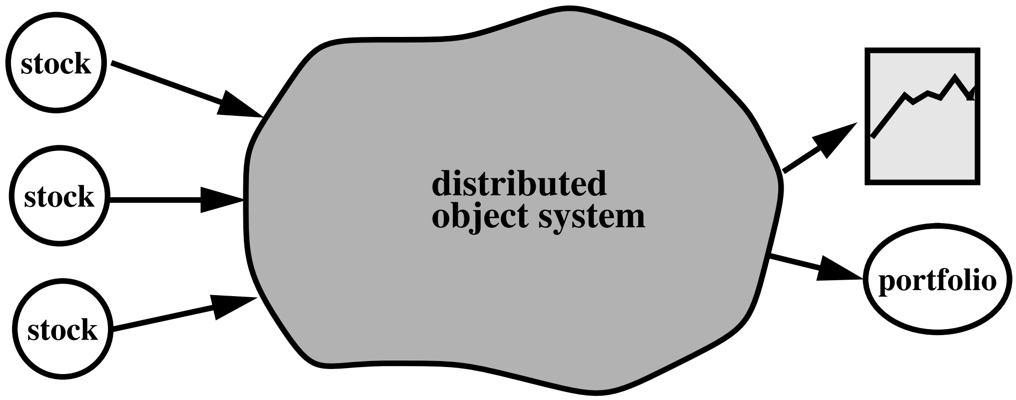

Figure 1. Decoupled object communication in a stock trading application

The Zephyr[1] notification service is a policy rich set of programs providing time-sensitive communications among clients and servers. Zephyr servers provide centralized routing, queuing and dispatching. A reliable and authenticated transmission protocol is supported by the client library. More advanced communication services are provided as additional layers.

In [11], the authors describe a simple and useful system that supports registration and notification of events in an RPC system. Third party services can also be introduced into the chain of notifications allowing for more complex functionality.

Our focus is events in CORBA-based distributed object systems. We first proceed with an overview of the principles of distributed object systems. We then describe an *event channel*, an object that decouples the communication between objects in a distributed object system, motivating our design based on the principles of distributed object systems. Finally, we describe our experience implementing event channels and using them in distributed applications.

2.0 Distributed Object Systems

Distributed object systems embody the following principles:

2.1 All entities are modeled as communicating objects

In a distributed object system all entities are modeled as *objects*. Although the systems being bridged by a distributed object system may include entities that are not objects, they are not available across system boundaries unless they can be presented as objects.

Objects are accessed by clients using object references. Object references can be passed from one object to another. An object holding an object reference for a target object can make a request for its services. The distributed object system provides location transparency to clients. The request may be local or it may cross heterogeneous system and administrative boundaries. It is the distributed object system's job to mask the differences from the client object.

2.2 Interfaces define objects

In a distributed object system, objects are defined by their *interfaces*. An interface specifies a set of operations that defines the behavior of the object. The implementation of an object is separate and invisible; clients cannot depend on implementation properties, such as programming language, transient or persistent representation of the object. There can be multiple implementations of an interface. An interface does not imply any particular implementation(s), and a new implementation may be added at any time. (See [3], [9] for more discussion of separating interfaces from implementations.)

2.3 Federated systems

Distributed object systems are *federated* systems. Existing, disconnected heterogeneous systems are connected and made to interoperate. Gateways mask differences between systems by implementing mappings between concepts in each system, including interfaces, object models, object references and name spaces.

Distributed object systems have the potential of connecting large numbers of objects across system and administrative boundaries. There should be no limit to this; that is, the system should scale.

Federation and scalability lead to truly distributed system objects. Services that depend on a single, centrally administered repository of information are not acceptable. In particular, there is no authority, even a distributed one, that has information about all objects or even part of the information about all objects of one type. Instead, federated services are connected to other instances of the same service to widen their scope of discourse.

2.4 Distributed object systems are open

Heterogeneous systems consist of components that typically come from a variety of suppliers. In order for the components to interoperate, the interfaces between the components need to be standard. Implementations of the components, however, need not be standard. Different suppliers can each provide implementations of a service supporting a standard interface.

The Object Management Group

The Object Management Group (OMG) is promoting standards for distributed object systems among system software vendors. The OMG has currently defined two sets of standards, known as CORBA and COSS. CORBA is the core communication mechanism which all OMG objects use: it enables objects to issue requests of each other. COSS provides standard services that support the integration of distributed objects.

CORBA

The Common Object Request Broker Architecture (CORBA) [6] defines an interface definition language (IDL) for distributed objects. The language allows designers to specify interfaces as a set of operations and attributes. The language supports subtyping of interfaces. A function can be passed an object that supports a subtype of the interface expected by the function.

The CORBA defines object references. Object references are typed by interfaces specified in IDL. Object references unify access to objects. The client using the object cannot tell if the object being accessed is local or remote, who implements the object, or how it is implemented.

The CORBA also defines an interface repository. The interface repository contains descriptions of IDL interfaces and data types. Such descriptions can be accessed at run time to implement type-safe interobject communication. Federating CORBA compliant systems requires correlating the interfaces in different interface repositories.

CORBA Services

The CORBA Services Specifications (COSS) [7] defines a set of services for distributed object systems. The services are specified in OMG IDL and are intended to operate in CORBA environments. Currently, COSS defines a name service for mapping human readable names to object references, a persistence service for persistently representing object state, an object life cycle service for creating, copying, moving and removing objects, a relationship service[4] to support graphs of distributed objects, transaction and concurrency control

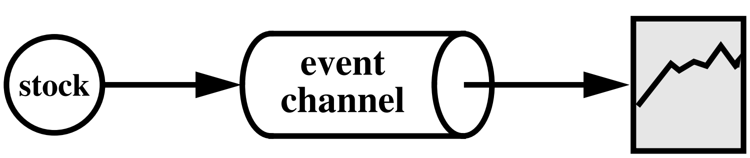

Figure 2. An event channel is an object service that decouples communication among distributed objects.

```
interface PushConsumer {

   void push(in any data);
      raises(Disconnected);

   void disconnect_push_consumer();

};
```

Figure 3. The PushConsumer interface.

services to implement atomic access to objects, an externalization service to externalize and internalize objects and an event service to decouple object communication. The COSS event service is based on the event service described in this paper.

3.0 Event channels

An event channel is an object that decouples object communication. Figure 2 illustrates the stock object and the chart object communicating using an event channel. The stock is the *supplier* of the event and the chart is the *consumer* of the event. An event channel is *both* a consumer and a supplier of events. In figure 2, the event channel consumes the events supplied by the stock object and then supplies those events which are consumed by the chart object.

3.1 Communicating events

Event communication is achieved via standard interobject requests. The event has data associated with it.[1] There are two styles of event communication, the *push* style and the *pull* style. The push style is similar to the *EventCatcher* notification style of [11].

In the push style, consumers support the *PushConsumer* interface, given in figure 3. Suppliers initiate the communication by invoking the *push* operation on the consumer. The *disconnect_push_consumer* operation terminates the event communication.

In figure 2, if the event channel is consuming events in the push style, it supports the *PushConsumer* interface; the stock invokes the *push* operation on the channel. Similarly, if the chart object is consuming events in the push style, it supports the *PushConsumer* interface; the channel invokes the *push* operation on the chart.

1. The data is of the CORBA IDL data type `any`. See section 3.5 for more discussion of data types.

In the pull style, suppliers support the *PullSupplier* interface given in figure 4. Consumers initiate the communication by invoking a *pull* operation on the supplier. Suppliers support two kinds of pull operations for returning events. The *pull* operation blocks until an event is available. The *try_pull* operation returns immediately, returning an event if one is available. The *disconnect_pull_supplier* operation terminates the event communication.

In figure 2, if the stock object is supplying events in the pull style, it supports the *PullSupplier* interface; the event channel invokes the *pull* operation on it. Similarly, if the event channel is supplying events in the pull style,

```
interface PullSupplier {

   any pull()
      raises(Disconnected);

   any try_pull(out boolean has_event)
      raises(Disconnected);

   void disconnect_pull_supplier();

};
```

Figure 4. The PullSupplier interface.

it supports the *PullSupplier* interface; the chart object invokes the *pull* operation on it.

In figure 2, the stock object can supply events to the event channel in either push or pull styles and the chart object can *independently* consume events from the event channel in either push or pull styles.

The two styles of event communication are very flexible. Push-style event communication is driven by the supplier of events, whereas pull-style event communication is driven by the consumer of events. A push-style consumer of an event channel does not have to actively solicit events. A pull-style consumer, on the other hand, has control over when an event is delivered to it. A push-style supplier to an event channel does not have to buffer events; it simply pushes events to the event channel as they happen. A pull-style supplier, on the other hand, can control the buffering policy.

Multi-way, anonymous communication

An event channel can supply multiple consumers. As illustrated in figure 5, the event channel supplies events to both the chart and the portfolio objects. The event

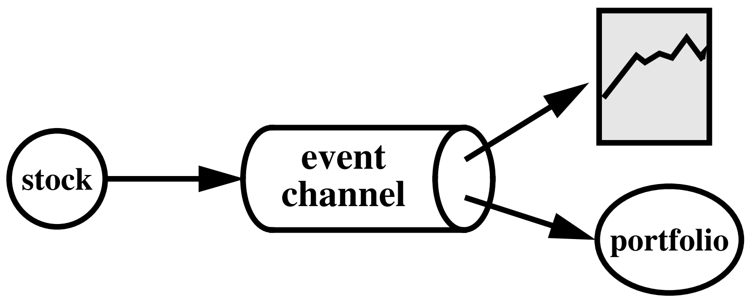

Figure 5. An event channel with multiple consumers.

channel independently communicates with the chart object and with the portfolio object in either push or pull styles.

The event channel makes the communication *anonymous*. The stock object is unaware of the consumers. Structuring a distributed application around anonymous event communication makes it easily extensible. The portfolio functionality was added to the application in figure 5 without modifying the stock object. The stock object continues to supply events, unaware of the consumers.

An event channel can also consume events from multiple suppliers. As illustrated in figure 6, multiple stocks can independently supply events to the same event channel; all consumers receive events from all suppliers.

When there are multiple suppliers, the suppliers of an event are anonymous. If the consumers need to distinguish the multiple suppliers, it can be done in event data.

3.2 Scoping events

So how many event channels are there? Should the cloud labelled “distributed object system” in figure 1 just be replaced with a single event channel? Clearly not; a single event channel for all events in the distributed object system becomes a bottleneck, does not scale and does not federate. Furthermore, all consumers would receive all of the events, most of which are not of interest.

An event channel provides a *scope* for events. There is no need for unrelated application domains to share an event channel. By definition, the consumers of one event channel will not receive the events supplied by another.

In practice, we have found scoping events by supplier to be an effective technique for organizing event communication.

Figure 7 illustrates scoping events by supplier in the stock application. Each stock object supplies events to

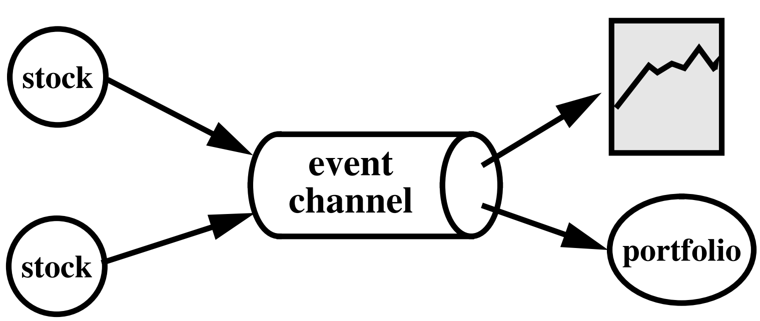

Figure 6. An event channel with multiple suppliers and multiple consumers.

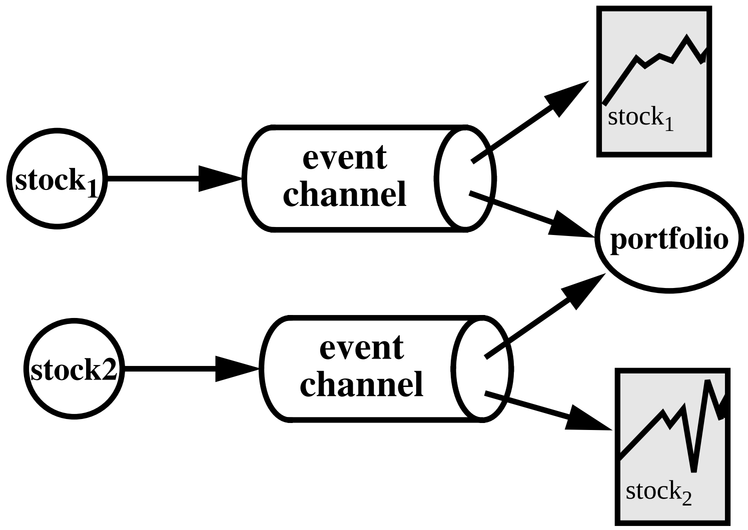

Figure 7. Scoping events by supplier

its own event channel; stock objects do not share event channels. A chart object consumes events from a single event channel since it charts a single stock. The portfolio object, on the other hand, consumes events from multiple event channels, since it makes trading decisions based on multiple stocks.

3.3 Filtering events

Even within the scope of a single event channel, not all consumers want to consume all events. In the stock example, the chart object wants to consume all changes in a stock's price, but suppose the portfolio object wants to consume stock price changes only when the price of a stock reaches a certain price. This can achieved with a special event channel called a *filter*. A filter is a specialized event channel that consumes events from another event channel but only supplies events that satisfy some condition. Events that do not satisfy the condition are simply discarded by the filter.

In Figure 8, only stock prices that are above (or below) a certain price are forwarded to the portfolio object; those that are not are discarded by the filter.

Filters are usually lightweight event channels that just evaluate the event data. A typical implementation of a push-style filter only supports a single consumer and makes no attempt to store and retry supplying the event if it has difficulties communicating with its consumer. The filter itself simply fails. The event channel that supplies events to the filter detects the failure and attempts to supply the event to the filter later.

Filters are best configured to be near the event channels they are filtering. This reduces communication costs when the filter consumes but then discards an event.

3.4 Chaining event channels

For a particular event, there might be numerous consumers residing at a remote location. In this case, chaining of two or more event channels can be used to

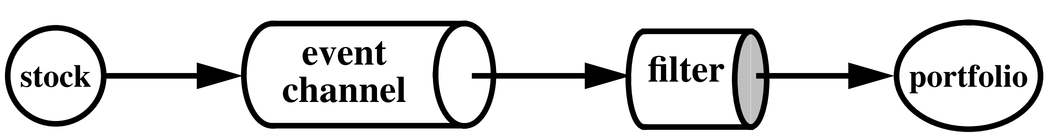

Figure 8. An event channel that filters events for its consumer

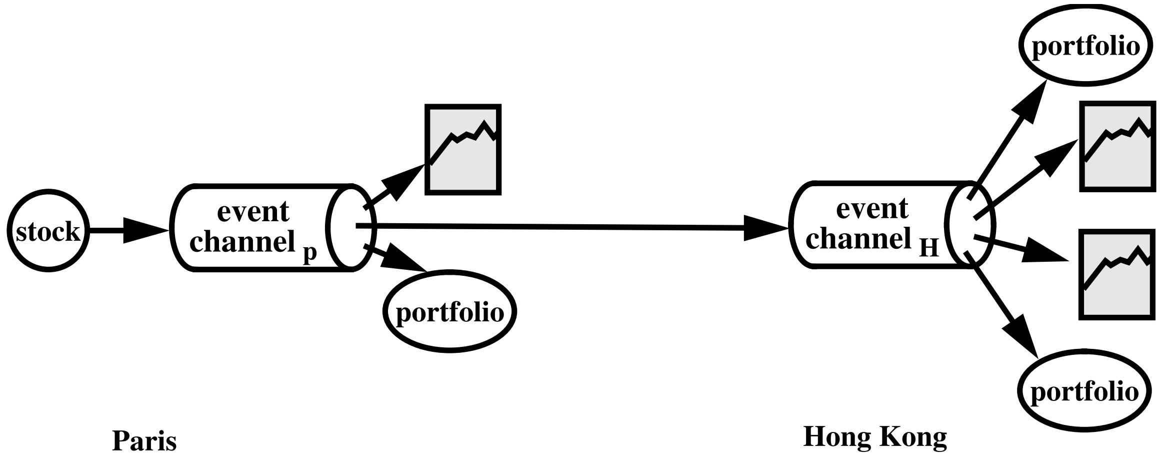

Figure 9. Chaining event channels to minimize communication costs

optimize the delivery of events. For example, in figure 9, there are several consumers of a stock event who are all in Hong Kong. Instead of delivering the same stock event from Paris to each of these consumers across the continents, an event is sent once to the event channel in Hong Kong which then forwards the event to the local consumers.

Event channels support an administrative interface that allows them to be chained. The interface supports operations to obtain consumer and supplier "proxy" objects. Two event channels can be chained in either push or pull styles.

In figure 9 chaining the event channel P in Paris as a push-style supplier and the event channel H in Hong Kong as a push-style consumer is achieved in three steps:

- request a push supplier proxy, p1, from event channel P. The push supplier proxy is a push-style supplier which also supports an operation to connect to a push consumer.
- request a push consumer proxy, p2, from H. p2 is a push-style consumer which also supports an operation to connect to a push supplier.
- connect the proxies p1 and p2 by calling their connect operations.

Event communication proceeds as follows: the stock pushes an event to the event channel P in Paris. P, in turn, pushes the event to its three consumers, one of which is the event channel H in Hong Kong. H, in turn, pushes the event to the two portfolio and two chart objects.

Alternatively, P and H can be chained in pull style by requesting and connecting pull-style proxies.

3.5 Typing events

The data associated with an event is of some programmer defined data type. The *PushConsumer* and *PullSupplier* interfaces given in figure 3 and figure 4 support the passing of the CORBA IDL `any` data type. The `any` data type specifies dynamically typed, self describing data. Any IDL data type can be passed. Event channels usually do not interpret the event data; they just pass it on to consumers. A consumer usually interprets the event data passed to it.

In order to federate multiple distributed object systems, their type spaces must be federated. In CORBA-based systems, this is accomplished by federating interface repositories. An interface repository from one system is merged with the interface repository of another system. Types can be correlated across multiple interface repositories using globally defined repository identifiers.

Rather than define its own type space, the event channel leverages the type system of the distributed object system. It leverages the distributed object type system by using standard interobject requests for communicating with its suppliers and its consumers. As such, event suppliers can communicate with event consumers, via one

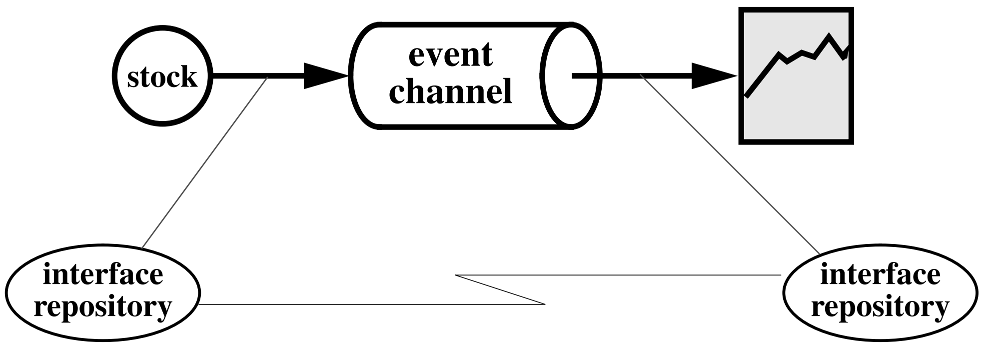

Figure 10. Federating interface repositories enables communication between suppliers and consumers across administrative boundaries.

or more channels, in a strongly typed fashion, even across federated system boundaries.

4.0 Implementing Event Channels

The IDL interfaces of the event channel define basic operations which provide support for asynchronous event communication. These interfaces are generic in the sense that they allow for a wide range of implementations in a wide range of computing environments. For example, one implementation of an event channel might choose to provide reliable delivery of events while another might provide less reliable, but faster delivery of events. One implementation might be optimized for execution in a local program, while another implementation might be designed for operation in a distributed environment of shared persistent objects.

A particular event channel implementation defines certain policies:

acquisition policy

This policy defines when an event channel obtains an event from its suppliers.

delivery policy

This policy defines which events are delivered to which consumers and when. This policy is especially of interest in a distributed environment in which consumers may be unavailable due to partial failures of the system.

discard policy

This policy defines when an event is discarded by an event channel.

persistence policy

This policy defines what part of an event channel's state is persistent and when an event channel saves changes to its persistent state.

4.1 Our implementation

At SunSoft we have implemented the event channel as part of SunSoft's NEO product[10]. NEO includes a CORBA-based, shared, persistent, multi-threaded[8] distributed object environment with location transparency and automatic activation and deactivation of objects. Our event channel implements the acquisition, delivery, discard and persistence policies in that environment as follows:

acquisition policy

An event channel accepts all events pushed at it. This eliminates the need for push-style suppliers to buffer their events by letting them leverage the event channel's buffer. Even if several events happen in the time it takes to push one event to the event channel, a multi-threaded supplier can concurrently push out all events as they occur.

An event channel pulls events from a pull-style supplier only in response to demand from consumers. The event channel dedicates a thread to each pull-style supplier. The thread invokes the *pull* operation on the pull-style supplier. The channel stays at most one event ahead of demand with respect to each pull-style supplier.

Since the event channel operates in a distributed environment, a pull-style supplier of an event may not be available. In such a case, the event channel retries with exponential back-off. If the event channel determines that the pull-style supplier no longer exists or has permanently failed, it disconnects the supplier from the channel.

Slow pull-style suppliers do not affect the service to other suppliers because the event channel isolates suppliers from each other by serving each supplier with a separate thread. A thread blocked in a *pull* request on a supplier does not impede event acquisition from other suppliers.

delivery policy

A consumer subscribes to events by connecting to and disconnecting from an event channel. When an event channel receives an event, it supplies it to all consumers that are currently connected to it. For pull-style consumers, the event is available via a *pull* operation. For push-style consumers, the event channel invokes the *push* operation on the consumer and passes the event data as an argument.

Since the event channel operates in a distributed environment, a push-style consumer may not be available. In such a case, the event channel retries with exponential back-off. If the event channel determines that the push-style consumer no longer exists or has permanently failed, it disconnects the consumer from the channel.

Slow push-style consumers do not affect the service to other consumers because the event channel isolates consumers from each other by serving each consumer with a separate thread. A thread blocked in a *push* request on a consumer does not impede event delivery to other consumers.

All consumers of an event channel receive events in the same order. Furthermore, the consumers receive the events from a single supplier in the order in which they were supplied to the channel.

Events are delivered to consumers at most once. An event may not be delivered to a consumer due to the channel's persistence and discard policies.

discard policy

An event channel has a finite buffer whose size is determined when it is created. When the buffer overflows, the oldest event in the buffer is discarded to make room for the newest.

Another seemingly reasonable approach is to discard the newest event when there is an overflow. However, this policy suffers a serious problem. Consider an event channel with several consumers, one of whom is very slow. When the buffer fills with events not yet delivered to this consumer, subsequent events will be discarded, causing a degradation of service to all other consumers. With our chosen policy, only the slow consumer will lose events. It is a good design principle to ensure where possible that service to well-behaved clients will not degrade because of ill-behaved clients.

persistence policy

Most objects in CORBA-based environments are persistent; the object request broker transparently activates objects upon client demand and deactivates objects to free up computational resources. As such, an event channel persistently remembers the object references of its suppliers and of its consumers. Thus a supplier, a consumer, or the event channel itself, can deactivate and the connection between suppliers and consumers remains valid.

In contrast, it is possible to justify a wider range of policy towards event data. Communication speed and reliability requirements of applications vary widely. Therefore, in our implementation, the creator of an event channel can choose at creation time whether the events are transient or persistent.

By definition, transient events are not saved to the event channel's persistent state. As such, transient events will be lost when an event channel deactivates. If persistent events are chosen, the creator may choose from among a range of save policies, determining the appropriate trade-off between communication speed and reliability. The most conservative policy saves the event data to the event channel's persistent state whenever a new event is supplied to the channel. Alternatively, the

TABLE 1. Push events throughput

Metric	Transient events events/sec 128 byte	4 byte	Persistent events events/sec 128 byte	4 byte
same machine push throughput. Supplier, consumer and event channel are separate processes on the same machine.	66	71	58	65
same LAN push throughput. Supplier, consumer and event channel are on separate machines on the same LAN.	100	144	90	125
same LAN multicast push throughput. Supplier, 5 consumers and event channel are on separate machines on the same LAN.	29	51	21	46

time interval between saves and/or the number of events delivered between saves can be specified by the creator of the event channel.

4.2 Performance of our implementation

The two primary performance metrics for the event channel are throughput and latency. The factors that impact these metrics are:

1. persistence of events
2. the size of data carried in the event
3. the number of consumers and suppliers
4. the location of the supplier(s), the event channel and the consumer(s)
5. push versus pull style

For the sake of brevity, we do not consider the effects of push versus pull style, nor multiple suppliers.

4.2.1 Measurement Methodology

We measure throughput by pushing events as fast as we can into an event channel which has a buffer of 100 events, without the event channel losing events. We then report the number of events delivered to the event channel per unit time in the steady state as the throughput. The latency is measured by recording the machine's high resolution time, pushing an event[1], recording the high resolution time when the event is received by the consumer and then repeating this entire measurement after a short delay. The difference in the two times is the latency. This procedure has the obvious problem that the clocks on different machines are not synchronized enough to allow accurate measurements at the millisecond level. We employ two methods to work around this problem. First, we have a test in which the supplier and consumer processes are located on one machine and the event channel is on another machine. Since the times are recorded in the supplier and consumer, and since both of these are running on the same machine and thus using the same low-level timer, the latency measurement is accurate. When the supplier and consumer are on separate machines, we use an application level echo protocol to calibrate the time skew between the supplier and each consumer to adjust the measured latency value.

All the measurements were conducted using Sparc 10/51 single processor workstations running NEO 1.0 and Solaris 2.4. When multiple workstations were used, they were interconnected with a lightly loaded 10 Mbit/sec Ethernet. The measurements used a sample size of 500 events.

4.2.2 Event Channel Throughput

Table 1 shows the throughput for push events. The multicast measurements use 5 push consumers on separate machines. Within each category of transient or persis-

1. The time required to place the event data into an "Any" datatype is not included in the measured latency. However, the time for marshalling and unmarshalling this data is included.

TABLE 2. Push events latency

Metric	Transient events millisec 128 byte	Transient events millisec 4 byte	Persistent events millisec 128 byte	Persistent events millisec 4 byte	Interobject requests millisec 128 byte	Interobject requests millisec 4 byte
same machine push latency. Supplier, consumer and event channel are separate processes on the same machine.	11	9	13	9	6.2	6.1
same LAN push-push latency. Supplier and consumer are on separate machines and the event channel is on a separate machine on the same LAN.	11.4	10	12.1	10	6.0	5.7
same LAN multicast push-push latency. Supplier and 5 consumers are on separate machines and the event channel is on a separate machine on the same LAN.	17	16	19.4	16	31	28

tent events, we show measurements for events containing 128 bytes of data and 4 bytes of data.

One interesting observation is that the throughput is higher when using a LAN than when all three participants (supplier, event channel and the consumer) are on the same machine. This is not hard to explain - on a single machine, the three participants contend for the single processor and memory whereas when using a LAN, there are 3 different machines providing three times the computing resources.

Another interesting comparison is between having a single consumer versus having 5 consumers. In a simple-minded event channel implementation, multicast throughput would be expected to be one-fifth for 5 consumers. However, since our event channel is implemented using a separate thread for each consumer, we get the advantage of concurrent operation even on a single processor running the event channel.

Since the persistent events require more processing in the event channel and also require periodic[1] saves of the persistent database, their throughput is lower than that of transient events. Also, the size of the event data also affects the throughput substantially.

1. Persistent events were configured to save event data every 100 events or 60 seconds whichever is earlier.

Transient events are more efficient than persistent events. However, because of the design of the persistent event channel, you can trade off performance against the reliability provided by persistent events. When an event channel is created, the two parameters “commit_time” and “commit_count” can be set based on the application needs. The larger these values, the faster a persistent event channel performs. However, event channels with a high commit_time or commit_count can lose more events in the case of the event server crash.

4.2.3 Event Channel Latency

Table 2 shows the latency measurements for the same factors as in table 1. The latency does not vary much between the same-machine and same-LAN cases. This is because for the latency measurements, we issue events one at a time so the three participants are not active simultaneously. Again, persistent events require slightly more time as do events containing more data bytes. The benefits of a multi-threaded event channel are obvious since the latency for delivering an event to 5 consumers is only slightly larger than the latency for a single consumer.

It is interesting to compare the latency through the event channel with the time required to make direct interobject requests. (The latter time is shown in the columns labeled “Interobject requests in table 2.) The event channel implementation takes about twice the time needed to make a direct interobject request which is quite reasonable since it does take two interobject requests (supplier to event channel and event channel to consumer) to deliver the event. When sending an event to 5 consumers, the event channel has a lower latency compared to the supplier making 5 interobject requests one after another to the 5 consumers. This clearly shows the performance gains obtained due to the multithreaded implementation of the event channel.

4.3 Experience using event channels

To date, several distributed applications have been built at SunSoft that use the event channel we have described here, including the stock trading application we have used to exemplify the event channel. Typically, we run this application with a few hundred stocks, a few hundred portfolios and dozens of concurrent presentations. Because of the scoping and scalability properties of the event service and the distributed object system, the application has operated on tens of thousands of stock objects simply by providing more computational and storage resources to the objects.

5.0 Conclusions

We have described how interobject communication in a distributed object system is decoupled using an *event channel*. Communication with an event channel is achieved via standard interobject requests.

The event channel implements the decoupling. In particular, it implements:

- asynchronous communication

  An event channel does not block a supplier while delivering an event to a consumer. Instead, it buffers the data, returns control to the supplier and asynchronously delivers the event to the consumers.

- failure handling

  Because event communication is asynchronous, the event channel handles communication failures by retrying to deliver (or acquire) the event at a later point in time. This relieves the supplier from having to include complicated error handling logic.

- many-to-many communication

  Many suppliers of events can share the same event channel. Similarly, many consumers of events can share the same event channel.

- anonymous communication

  A supplier does not direct request to a particular consumer, but rather it supplies events to an event channel. Anonymous communication makes a distributed application more extensible. Additional consumer objects that extend the functionality of the application can be added without disturbing the supplier objects.

We have also described how event channels provide a scope for events in a distributed objects system, how event channels can be chained to optimize communication costs, how events can be filtered and how strongly typed events are supported.

We have described SunSoft’s implementation of the event channel, its performance and our use of it in a simple stock trading application.

6.0 References

[1] C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohn and William E. Sommerfeld, “The Zepher Notification Service,” In *Proceedings of the Winter USENIX conference*, Winter, 1988.

[2] Bruce E. Martin, Claus H. Pedersen and James Bedford-Roberts, “An Object-Based Taxonomy of Distributed Computing Systems.” In *Readings in Distributed Computing Systems*, published by IEEE Computer Society Press. edited by Mukesh Singhal and Thomas L. Casavant. Also in *IEEE Computer* special issue on distributed computing systems, edited by Mukesh Singhal and Thomas L. Casavant, August, 1991.

[3] Bruce E. Martin, “The Separation of Interface and Implementation in C++.” In *The Evolution of C++*, edited by Jim Waldo, published by MIT press. Also in *Proceedings of the 3rd USENIX C++ Conference*, April, 1991, Washington, D.C.

[4] Bruce E. Martin and R.G.G. Cattell, “Relating Distributed Objects.” In The Proceedings of the 20th VLDB Conference, September, 1994, Santiago, Chile.

[5] John R. Nicol, C. Thomas Wilkes and Frank A. Manola, “Object Orientation in Heterogeneous Distributed Computing Systems,” *IEEE Computer*, June, 1993.

[6] Object Management Group, “The Common Object Request Broker: Architecture and Specification, version 2”, March, 1995

[7] Object Management Group, “CORBAservices: Common Object Services Specification”, OMG Document Number 95.3.31, March 31, 1995.

[8] M. L. Powell, S. R. Kleinman, S. Barton, D. Shah, D. Stein and M. Weeks, “Sun{OS} Multi-thread Architecture”, In *Proceedings of Usenix Winter conference*, Winter, 1991.

[9] Alan Snyder. “Encapsulation and Inheritance in Object-oriented Programming Languages”, In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA). Association of Computing Machinery, 1986.

[10] SunSoft, Solaris NEO Operating Environment, Product Overview, 1995.

[11] Jim Waldo, Ann Wollrath, Geoff Wyant and Samuel C. Kendall. “Events in an RPC Based Distributed System.” In *Proceedings of the 1995 USENIX Annual Technical Conference*.