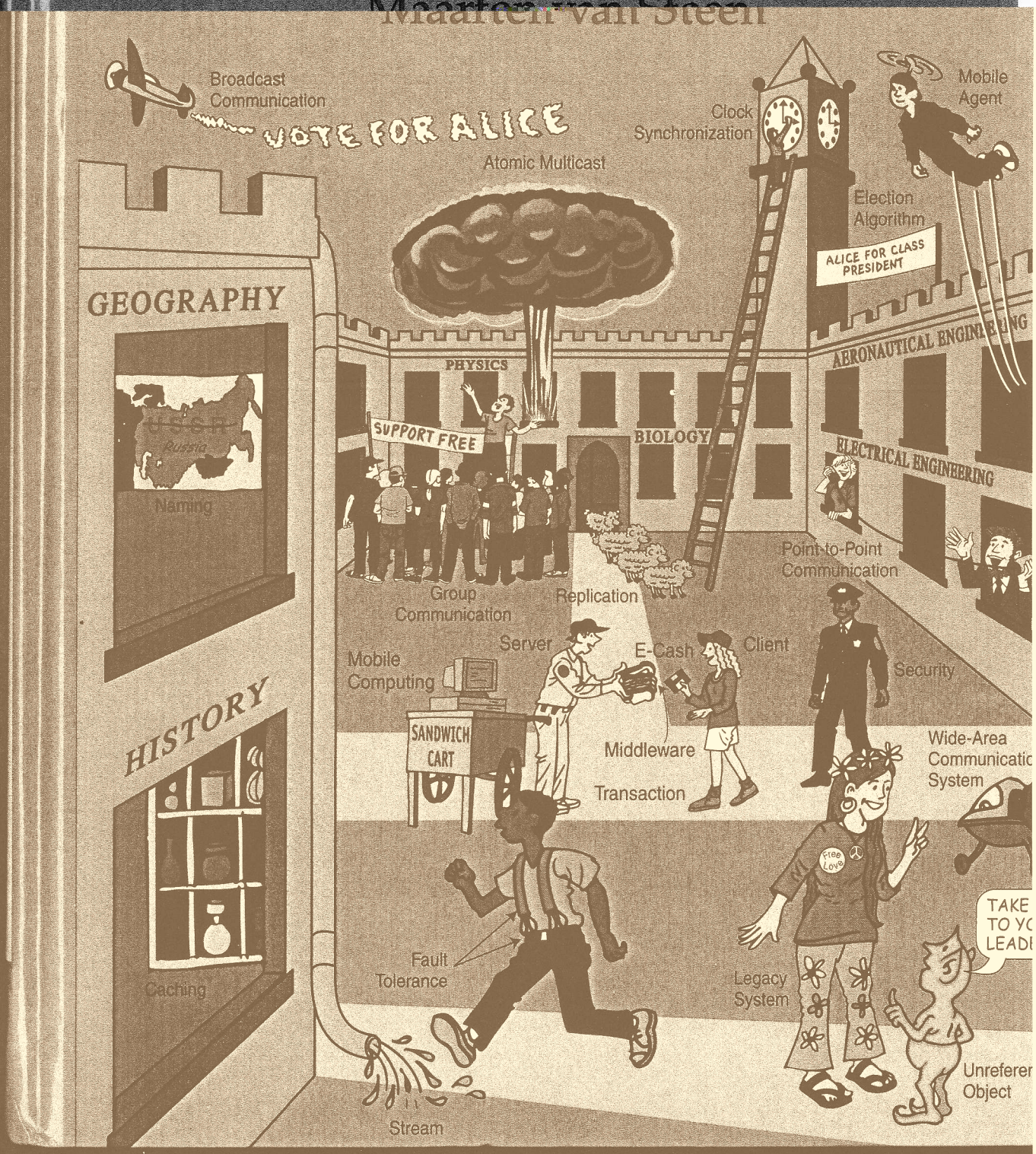
DISTRIBUTED SYSTEMS

Principles and Paradigms

Andrew S. Tanenbaum
Maarten van Steen



Library of Congress Cataloging-in-Publication Data
on file

Vice President and Editorial Director, ECS: ***Marcia Horton***
Publisher: ***Alan Apt***
Associate Editor: ***Toni D. Holm***
Vice President and Director of Production and Manufacturing, ESM: ***David W. Riccardi***
Executive Managing Editor: ***Vince O'Brien***
Managing Editor: ***David A. George***
Assistant Managing Editor and Production Coordinator: ***Camille Trentacoste***
Director of Creative Services: ***Paul Belfanti***
Creative Director: ***Carole Anson***
Art Director: ***Heather Scott***
Assistant to Art Director: ***John Christiana***
Cover Art and Design: ***Don Martinetti***
Cover Illustration Concept: ***Andrew S. Tanenbaum*** and ***Maarten van Steen***
Interior Design and Typesetting: ***Andrew S. Tanenbaum***
Interior Illustrations: ***Maarten van Steen***
Manufacturing Manager: ***Trudy Pisciotti***
Manufacturing Buyer: ***Lisa McDowell***
Marketing Manager: ***Jennie Burger***

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-088893-1

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Pearson Education Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

once semantics, implying that the invoked method may have been invoked once, or not at all. Note that it is up to an implementation to provide these semantics.

Synchronous invocation is therefore useful when the client expects an answer. If a proper response is returned, CORBA guarantees that the method has been invoked exactly once. However, in those cases where no response is needed, it would be better for the client to simply invoke the method and continue with its own execution as soon as possible. This type of invocation is similar to the asynchronous RPCs we discussed in Chap. 2.

In CORBA, this form of asynchronous invocation is called a **one-way request**. A method can be specified as being one-way only if it is specified to return no results. However, unlike the guaranteed delivery of asynchronous RPCs, one-way requests in CORBA offer only a best-effort delivery service. In other words, no guarantees are given to the caller that the invocation request is delivered to the object's server.

Besides one-way requests, CORBA also supports what is known as a **deferred synchronous request**. Such a request is, in fact, a combination of a one-way request and letting the server asynchronously send the result back to the client. As soon as the client sends its request to the server, it continues without waiting for any response from the server. In other words, the client can never know for sure whether its request is actually delivered to the server.

These three different invocation models are summarized in Fig. 9-4.

Request type	Failure semantics	Description
Synchronous	At-most-once	Caller blocks until a response is returned or an exception is raised
One-way	Best effort delivery	Caller continues immediately without waiting for any response from the server
Deferred synchronous	At-most-once	Caller continues immediately and can later block until response is delivered

Figure 9-4. Invocation models supported in CORBA.

Event and Notification Services

Although the invocation models offered by CORBA should normally cover most of the communication requirements in an object-based distributed system, it was felt that having only method invocations was not enough. In particular, there was a need to provide a service that could simply signal the occurrence of an event. Clients amenable to that event could take appropriate action.

The result is the definition of an **event service**. The basic model for events in CORBA is quite simple. Each event is associated with a single data item, generally represented by means of an object reference, or otherwise an application-

specific value. An event is produced by a **supplier** and received by a **consumer**. Events are delivered through an **event channel**, which is logically placed between suppliers and consumers, as shown in Fig. 9-5.

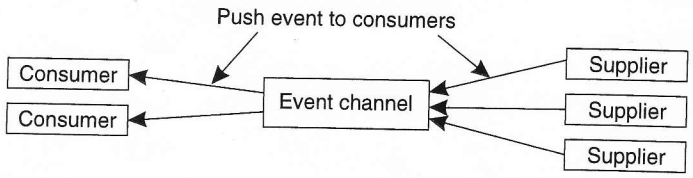


Figure 9-5. The logical organization of suppliers and consumers of events, following the push-style model.

Fig. 9-5 shows what is referred to as the **push model**. In this model, whenever an event occurs, the supplier produces the event and pushes it through the event channel, which then passes the event on to its consumers. The push model comes closest to the asynchronous behavior that most people associate with events. In effect, consumers passively wait for event propagation, and expect to be interrupted one way or the other when an event happens.

An alternative model, also supported by CORBA, is the **pull model** shown in Fig. 9-6. In this model, consumers poll the event channel to check whether an event has happened. The event channel, in turn, polls the various suppliers.

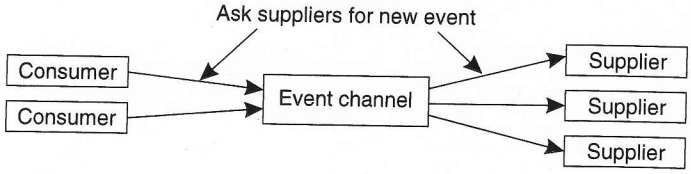


Figure 9-6. The pull-style model for event delivery in CORBA.

Although the event service provides a simple and straightforward way for event propagation, it has a number of serious drawbacks. In order to propagate events, it is necessary that suppliers and consumers are connected to the event channel. This also means that when a consumer connects to an event channel after the occurrence of an event, that afterward, the event will be lost. In other words, CORBA's event service does not support persistence of events.

More serious is that consumers have hardly any means to filter events; each event is, in principle, passed to every consumer. If different event types need to be distinguished, it is necessary to set a separate event channel for each type. Filtering capabilities have been added to an extension called the **notification service** (OMG, 2000a). In addition, this service offers facilities to prevent event propagation when no consumers are interested in a specific event.

Finally, event propagation is inherently unreliable. The CORBA specifications state that no guarantees need to be given concerning the delivery of events.

As we will discuss in Chap. 12, applications exist for which reliable event propagation is important. Such applications should not use the CORBA event service, but should resort to other means of communication.

Messaging

All communication in CORBA as described so far is transient. This means that a message is stored by the underlying communication system only as long as both its sender and its receivers are executing. As we discussed in Chap. 2, there are many applications that require persistent communication so that a message is stored until it can be delivered. With persistent communication, it does not matter whether the sender or receiver is executing after the message has been sent; in all cases, it will be stored as long as necessary.

A well-known model for persistent communication is the message-queuing model. CORBA supports this model as an additional **messaging service**. What makes messaging in CORBA different from other systems is its inherent object-based approach to communication. In particular, the designers of the messaging service needed to retain the model that all communication takes place by invoking an object. In the case of messaging, this design constraint resulted in two additional forms of asynchronous method invocations.

In the **callback model**, a client provides an object that implements an interface containing callback methods. These methods can be called by the underlying communication system to pass the result of an asynchronous invocation. An important design issue is that asynchronous method invocations do not affect the original implementation of an object. In other words, it is the client's responsibility to transform the original synchronous invocation into an asynchronous one; the server is presented a normal (synchronous) invocation request.

Constructing an asynchronous invocation is done in two steps. First, the original interface as implemented by the object is replaced by two new interfaces that are to be implemented by client-side software only. One interface contains the specification of methods that the client can call. None of these methods returns a value or has any output parameter. The second interface is the callback interface. For each operation in the original interface, it contains a method that will be called by the client's ORB to pass the results of the associated method as called by the client.

As an example, consider an object implementing a simple interface with just one method:

```
int add(in int i, in int j, out int k);
```

Assume that this method (which we expressed in CORBA IDL) takes two nonnegative integers i and j and returns $i + j$ as output parameter k. The operation is assumed to return -1 if the operation did not successfully complete. Transforming the original (synchronous) method invocation into an asynchronous one with