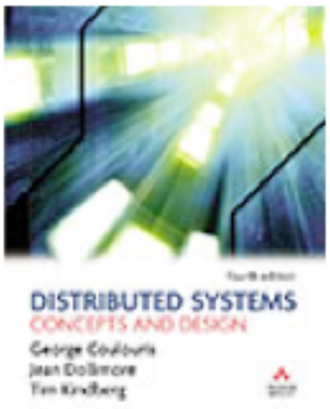

the subscriber is ready to receive them. The subscriber should be able to turn delivery on and off as required. The subscriber sets up a notification mailbox when it registers with an object of interest by specifying the notification mailbox as the place to send notifications.

5.4.2 Case study: Jini distributed event specification

The Jini distributed event specification described by Arnold *et al.* [1999] allows a potential subscriber in one Java Virtual Machine (JVM) to subscribe to and receive notifications of events in an object of interest in another JVM, usually on another computer. A chain of observers may be inserted between the object of interest and the subscriber. The main objects involved in the Jini distributed event specification are:

Event generators: An event generator is an object that allows other objects to subscribe to its events and generates notifications.

Remote event listeners: A remote event listener is an object that can receive notifications.

Remote events: A remote event is an object that is passed by value to remote event listeners. A remote event is the equivalent of what we called a notification.

Third-party agents: Third-party agents may be interposed between an object of interest and a subscriber. They are the equivalent of our observers.

An object subscribes to events by informing the event generator about the type of event and specifying a remote event listener as the target for notifications.

Java RMI is used to send notifications from the event generator to the subscriber, possibly via one or more first third-party agents. The designers state that event listeners should reply to notification calls as soon as possible to avoid delaying event generators. They can process a notification after the return. Java RMI is also used to subscribe to events. Jini events are provided by means of the following interfaces and classes:

RemoteEventListener: This interface provides a method called *notify*. Subscribers and third-party agents implement the *RemoteEventListener* interface so that they can receive notifications when the *notify* method is invoked. An instance of the *RemoteEvent* class represents a notification and is passed as argument to the *notify* method.

RemoteEvent: This class has instance variables that hold:

- a reference to the event generator in which the event occurred;
- an event identifier, which specifies the type of event at that event generator;
- a sequence number, which applies to events of that type. The sequence number should increase as events occur over time. It can be used to enable recipients to order events of a particular type from a given source or to avoid applying the same event twice;
- a marshalled object. This is supplied when the recipient subscribes to that type of event and may be used by a recipient for any purpose. It generally holds any information needed by the recipient to identify the event and react to its occurrence. For example, it could include a closure that is to be run when it is notified.

EventGenerator: This interface provides a method called *register*. Event generators implement the *EventGenerator* interface, whose *register* method is used to subscribe to events at the event generator. The arguments of *register* specify:

- an event identifier, which specifies the type of event;
- a marshalled object to be handed back with each notification;
- a remote reference to an event listener object – the place to send notifications;
- a requested leasing period. The lease period specifies the duration of lease required by the subscriber, but the actual lease granted is returned with the results of *register*. Time limits on subscriptions avoid the problem of event generators holding stale event subscriptions. Subscriptions can be renewed whenever the time limit in the lease expires.

The Jini specification says that the *EventGenerator* interface is just an example of the kind of interface that might be used by subscribers to register interest in events at an object of interest. Some applications may require a different interface.

Third-party agents ◊ The third-party agents that are interposed between an event generator and a subscriber may play a variety of useful roles, including all of those described above.

In the simplest case, a subscriber registers interest in a particular type of event at an event generator and specifies itself as the remote event listener. This corresponds to case 1 illustrated in Figure 5.11.

Third-party agents can be set up by an event generator or by a subscriber.

An event generator can interpose one or more third-party agents between itself and a subscriber. For example, the event generators on each computer could make use of a shared third-party agent that is responsible for reliable delivery of notifications.

A subscriber can build a chain of third-party agents in order to produce whatever delivery policy it requires. It then registers interest with an event generator, specifying the first in the chain of third-party agents as the place to send notifications. For example, a subscriber may arrange for its notifications to be stored by a third-party agent until such time as it is ready to receive them. The third-party agent can take responsibility for renewing leases.

5.5 Case study: Java RMI

Java RMI extends the Java object model to provide support for distributed objects in the Java language. In particular, it allows objects to invoke methods on remote objects using the same syntax as for local invocations. In addition, type checking applies equally to remote invocations as to local ones. However, an object making a remote invocation is aware that its target is remote because it must handle *RemoteException*s; and the implementor of a remote object is aware that it is remote because it must implement the *Remote* interface. Although the distributed object model is integrated into Java in a natural way, the semantics of parameter passing differ because invoker and target are remote from one another.

The programming of distributed applications in Java RMI should be relatively simple because it is a single-language system – remote interfaces are defined in the Java language. If a multiple-language system such as CORBA is used, the programmer needs to learn an IDL and to understand how it maps onto the implementation language. However, even in a single-language system, the programmer of a remote object must consider its behaviour in a concurrent environment.

In the remainder of this introduction, we give an example of a remote interface, then discuss the parameter-passing semantics with reference to the example. Finally, we discuss the downloading of classes and the binder. The second section of this case study discusses how to build client and server programs for the example interface. The third section is concerned with the design and implementation of Java RMI. For full details of Java RMI, see the tutorial on remote invocation [java.sun.com I].

In this case study and the CORBA case study in Chapter 20 as well as in the discussion of web services in Chapter 19, we use a *shared whiteboard* as an example. This is a distributed program that allows a group of users to share a common view of a drawing surface containing graphical objects, such as rectangles, lines and circles, each of which has been drawn by one of the users. The server maintains the current state of a drawing by providing an operation for clients to inform it about the latest shape their users have drawn and keeping a record of all the shapes it has received. The server also provides operations allowing clients to retrieve the latest shapes drawn by other users by polling the server. The server has a version number (an integer) that increments each time a new shape arrives and attaches to the new shape. The server provides operations