Jini™ Distributed Event Specification

The Distributed Event and Notification system defines a set of interfaces and associated conventions and protocols that allow objects in different Java™ virtual machines, perhaps located on different physical machines, to identify state changes that could be of interest to other objects, allow registration of interest in those state changes, and send notifications when those state changes occur to all who have registered interest. Along with the interfaces and conventions are a set of classes that allow programmers to use the interfaces to construct distributed programs using the event and notification model.

THE NETWORK IS THE COMPUTER™

901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300
fax 415 969-9131

Revision 1.0
January 25, 1999

Contents

1. Introduction 1

1.1 Distributed Events and Notifications. 1

1.2 Goals and Requirements 2

1.3 Dependencies 3

1.4 Comments 3

2. The Basic Interfaces 5

2.1 Entities Involved. 6

2.2 Overview of the Interfaces and Classes. 7

2.3 Details of the Interfaces and Classes 9

2.4 Sequence Numbers, Leasing and Transactions. 15

2.5 Serialized Forms 16

3. Third-party objects 19

3.1 Store-and-Forward agents. 20

3.2 Notification Filters 22

3.3 Notification Mailboxes. 23

3.4 Compositionality 24

4. Integration with JavaBeans™ Components 27

4.1 Differences with the JavaBeans Component Event Model 28

4.2 Making a JavaBeans Component Event out of a Distributed Event 30

Introduction 1

The purpose of the distributed event interfaces specified in this document is to allow an object in one Java™ virtual machine (JVM) to register interest in the occurrence of some event occurring in an object in some other JVM, perhaps running on a different physical machine, and to receive a notification when an event of that kind occurs.

1.1 Distributed Events and Notifications

Programs based on an object reacting to a change of state somewhere outside the object are common in a single address space. Such programs are often used for interactive applications in which user actions are modeled as events to which other objects in the program react. Delivery of such *local events* can be assumed to be well-ordered, very fast, predictable and reliable. Further, the entity interested in the event can be assumed to always want to know about the event as soon as the event has occurred.

The same style of programming is useful in distributed systems, where the object reacting to an event is in a different JVM, perhaps on a different physical machine, from the one on which the event occurred. Just as in the single-JVM case, the logic of such programs is often reactive, with actions occurring in response to some change in state that has occurred elsewhere.

A distributed event system has a different set of characteristics and requirements than a single-address-space event system. Notifications of events from remote objects may arrive in different orders on different clients, or may not arrive at all. The time it takes for a notification to arrive may be long (in

comparison to the time for computation at either the object that generated the notification or the object interested in the notification). There may be occasions in which the object wishing the event notification does not wish to have that notification as soon as possible, but only on some schedule determined by the recipient. There may even be times when the object that registered interest in the event is not the object to which a notification of the event should be sent.

Unlike the single address space notion of an event, a distributed event cannot be guaranteed to be delivered in a timely fashion. Because of the possibilities of network delays or failures, the notification of an event may be delayed indefinitely and even lost in the case of a distributed system.

Indeed, there are times in a distributed system where the object of a notification may actively desire that the notification be delayed. In systems that allow object activation (such as is allowed by Java Remote Method Invocation (RMI) in the Java™ Development Kit, version 1.2 (JDK1.2), an object may wish to be able to find out if an event occurred, but not want that notification to cause an activation of the object if it is otherwise quiescent. In such cases, the object receiving the event may wish the notification to be delayed until the object requests notification delivery, or until the object has been activated for some other reason.

Central to the notion of a distributed notification is the ability to place a third-party object between the object that generates the notification and the party that ultimately wishes to receive the notification. Such third parties, which can be strung together in arbitrary ways, allow ways of off-loading notifications from objects, implementing various delivery guarantees, storing of notifications until needed or desired by a recipient, and the filtering and re-routing of notifications. In a distributed system in which full applications are made up of components assembled to produce an overall application, the third party may be more than a filter or storage spot for a notification; in such systems it is possible that the third party is the final intended destination of the notification.

1.2 Goals and Requirements

The requirements of this set of interfaces are to:

- Specify an interface that can be used to send a notification of the occurrence of the event
- Specify the information that must be contained in such a notification

In addition, the fact that the interfaces are designed to be used by objects in different virtual machines, perhaps separated by a network, imposes other requirements, including:

- Allowing various degrees of assurance on delivery of a notification
- Support for different policies of scheduling notification
- Explicitly allowing the interposition of objects that will collect, hold, filter, and forward notifications

Notice that there is no requirement for a single interface that can be used to register interest in a particular kind of event. Given the wide variety of kinds of events, the way in which interest in such events can be indicated may vary from object to object. This document will talk about a model that lies behind the system's notion of such a registration, but the interfaces used to accomplish such a registration are not open to general description.

1.3 Dependencies

This document relies on the following other specifications:

- Java Remote Method Invocation Specification
- Jini™ Distributed Leasing Specification

1.4 Comments

Please direct comments to `jini-comments@java.sun.com`.

The Basic Interfaces 2

The basic interfaces presented in this chapter define a protocol that can be used by one object to register interest in a kind of state change in another object, and to receive a notification of an occurrence of that kind of state change, either directly or through some third-party, that is specified by the object at the time of registration. The protocol is meant to be as simple as possible. No attempt is made to indicate the reliability or the timeliness of the notifications; such guarantees are not part of the protocol but instead are part of the implementation of the various objects involved.

In particular, the purpose of these interfaces is:

- To show the information needed in any method that allows registration of interest in the occurrence of a kind of event in an object
- To provide an example of an interface that allows the registration of interest in such events
- To specify an interface that can be used to send a notification of the occurrence of the event

Implicit in the event registration and notification is the idea that events can be classified into *kinds*. Registration of interest indicates the kind of event that is of interest, while a notification indicates that an instance of that kind of event has occurred.

2.1 Entities Involved

An *event* is something that happens in an object, corresponding to some change in the abstract state of the object. Events are abstract occurrences that are not directly observed outside of an object, and may not correspond to a change in the *actual* state of the object that advertises the ability to register interest in the event. However, an object may choose to export an identification of a kind of event and allow other objects to indicate interest in the occurrence of events of that kind; this indicates that the *abstract* state of the object includes the notion of this state changing. The information concerning what kinds of events occur within an object can be exported in a number of ways, including identifiers for the various events or methods allowing registration of interest in that kind of event.

An object is responsible for identifying the kinds of events that can occur within that object, allowing other objects to register interest in the occurrence of such events, and generating `RemoteEvent` objects that are sent as notifications to the objects that have registered interest when such events occur.

Registration of interest is not temporally open ended, but is limited to a given duration using the notion of a lease. Full specification of the way in which leasing is used is contained in the *Jini™ Distributed Leasing Specification*.

The basic, concrete objects involved in a distributed event system are:

- The object that registers interest in an event
- The object in which an event occurs (referred to as the event generator)
- The recipient of event notifications (referred to as a remote event listener)

An *event generator* is an object that has some kinds of abstract state changes that might be of interest to other objects, and allows other objects to register interest in those events. This is the object that will generate notifications when events of this kind occur, sending those notifications to the event listeners that were indicated as targets in the calls that registered interest in that kind of event.

A *remote event listener* is an object that is interested in the occurrence of some kinds of events in some other object. The major function of a remote event listener is to receive notifications of the occurrence of an event in some other object (or set of objects).

A *remote event* is an object that is passed from an event generator to a remote event listener to indicate that an event of a particular kind has occurred. At a minimum, a remote event contains information about the kind of event that has occurred, a reference to the object in which the event occurred, and a sequence number allowing identification of the particular instance of the event. A notification will also include an object that was supplied by the object that registered interest in the kind of event as part of the registration call.

2.2 Overview of the Interfaces and Classes

The event and notification interfaces introduced here define a single basic type of entity, a set of requirements on the information that needs to be handed to that entity, and some supporting interfaces and classes. All of the classes and interfaces defined in this specification are in the `net.jini.core.event` package.

The basic type is defined by the interface `RemoteEventListener`. This interface requires certain information to be passed in during the registration of interest in the kind of event that the notification is indicating; while there is no single interface that defines how to register interest in such events the ways in which such information could be communicated will be discussed.

The supporting interfaces and classes define a `RemoteEvent` object, an `EventRegistration` object used as an identifier for registration, and a set of exceptions that can be generated.

The `RemoteEventListener` is the receiver of `RemoteEvent`s, which signal that a particular kind of event has occurred. A `RemoteEventListener` is defined by an interface that contains a single method, `notify`, which informs interested listeners that an event has occurred. This method returns no value, and has parameters that contain enough information to allow the method call to be idempotent. In addition, this method will return information that was passed in during the registration of interest in the event, allowing the *registrant*, the object that registered interest with the event generator, to associate arbitrary information or actions with the notification.

The `RemoteEventListener` interface extends from the `Remote` interface, so the methods defined in `RemoteEventListener` are remote methods and objects supporting these interfaces will be passed by RMI, by reference. Other objects defined by the system will be local objects, passed by value in the remote calls.

The first of these supporting classes is `RemoteEvent`, which is sent to indicate that an event of interest has occurred in the event generator. The basic form of a `RemoteEvent` contains:

- An identifier for the kind of event in which interest has been registered
- A reference to the object in which the event occurred
- A sequence number identifying the instance of the event type
- An object that was passed in, as part of the registration of interest in the event by the registrant

These `RemoteEvent` notification objects are passed to a `RemoteEventListener` as a parameter to the `RemoteEventListener notify` method.

The `EventRegistration` class defines an object that returns the information needed by the registrant, and is intended to be the return value of remote event registration calls. Instances of the `EventRegistration` class contain an identifier for the kind of event, the current sequence number of the kind of event, and a `Lease` object for the registration of interest.

While there is no single interface that allows for the registration of event notifications, there are a number of requirements that would be put on any such interface if it wished to conform with the remote event registration model. In particular, any such interface should reflect:

- Event registrations are bounded in time, in a way that allows those registrations to be renewed when necessary. This can easily be reflected by returning, as part of an event registration, a lease for that registration.
- Notifications need not be delivered to the entity that originally registered interest in the event. The ability to have third-party filters greatly enhances the functionality of the system. The easiest way to allow such functionality is to allow the specification of the `RemoteEventListener` to receive the notification as part of the original registration call.
- Notifications can contain a `MarshalledObject` supplied by the original registrant, allowing the passing of arbitrary information (including a closure that is to be run on notification) as part of the event notification; so the registration call should include a `MarshalledObject` that is to be passed as part of the `RemoteEvent`.

2.3 Details of the Interfaces and Classes

2.3.1 The `RemoteEventListener` Interface

The `RemoteEventListener` interface needs to be implemented by any object that wants to receive a notification of a `RemoteEvent` from some other object. The object supporting the `RemoteEventListener` interface does not have to be the object that originally registered interest in the occurrence of an event. To allow the notification of an event's occurrence to be sent to an entity other than the one that registered with the event generator, the registration call needs to accept a destination parameter, that indicates the object to which the notification should be sent. This destination must be an object which implements the `RemoteEventListener` interface.

The `RemoteEventListener` interface extends the `Remote` interface (indicating that it is an interface to a `Remote` object) and the `java.util.EventListener` interface. This latter interface is used in the Java Abstract Window Toolkit (AWT) and JavaBeans™ components to indicate that an interface is the recipient of event notifications. The `RemoteEventListener` interface consists of a single method, `notify`:

```
public interface RemoteEventListener extends Remote,
                    java.util.EventListener
{
     void notify(RemoteEvent theEvent)
        throws UnknownEventException, RemoteException;
}
```

The `notify` method has a single parameter of type `RemoteEvent` that encapsulates the information passed as part of a notification. The `RemoteEvent` base class extends the class `java.util.EventObject` that is used in both JavaBeans components and AWT components to propagate event information.The `notify` method returns nothing, but can throw exceptions.

2.3.2 The `RemoteEvent` Class

The public part of the `RemoteEvent` class is defined as:

```
public class RemoteEvent extends java.util.EventObject {
    public RemoteEvent(Object source,
                       long   eventID,
                      long   seqNum,
                      MarshalledObject handback)
    public Object getSource();
    public long getID();
    public long getSequenceNumber();
    public MarshalledObject getRegistrationObject();
}
```

The abstract state contained in a `RemoteEvent` object includes: a reference to the object in which the event occurred, a `long` which identifies the kind of event relative to the object in which the event occurred, a `long` which indicates the sequence number of this instance of the event kind, and a `MarshalledObject` that is to be handed back when the notification occurs.

The combination of the event identifier and the object reference of the event generator obtained from the `RemoteEvent` object should uniquely identify the event type. If this type is not one in which the `RemoteEventListener` has registered interest (or in which someone else has registered interest on behalf of the `RemoteEventListener` object), an `UnknownEventException` may be generated as a return from the remote event listener's `notify` method[1].

On receipt of an `UnknownEventException`, the caller of the `notify` method is allowed to cancel the lease for the combination of the `RemoteEventListener` instance and the kind of event that was contained in the `notify` call.

The sequence number obtained from the `RemoteEvent` object is an increasing value that can act as a hint to the number of occurrences of this event relative to some earlier sequence number. Any object that generates a `RemoteEvent` is required to insure that for any two `RemoteEvent` objects with the same event identifier, the sequence number of those events differ if and only if the `RemoteEvent` objects are a response to different events. This guarantee is required to allow notification calls to be idempotent. A further guarantee is

1. There are cases where the `UnknownEventException` may not be appropriate, even when the notification is for a combination of an event and a source that is not expected by the recipient. Objects that act as event mailboxes for other objects, for example, may be willing to accept any sort of notification from a particular source until explicitly told otherwise.

that if two RemoteEvents, *a* and *b*, come from the same source and have the same event identifier, then *a* occurred before *b* if and only if the sequence number of *a* is lower than the sequence number of *b*.

A stronger guarantee can be made by a service that generates RemoteEvents, in which sequence numbers are guaranteed to not only be unique but strictly increasing. This guarantees that, if RemoteEvent *a* has sequence number `m` and RemoteEvent *b* has sequence number `m + 1`, with *a* and *b* coming from the same source and having the same event identifier, then the receiver can be assured that there was no event of the same type from that source that occurred between the events that caused *a* and *b*. This guarantee is referred to as “strict ordering of the sequence numbers”. Note that this guarantee does not tell the receiver anything about the number of events that could have occurred between the events that caused *a* and *b* if there is a gap of more than 1 between the sequence numbers.

An even stronger guarantee is possible for those generators of RemoteEvents that can support it. This guarantee states that not only do sequence numbers increase, but they are not skipped. In such a case, if RemoteEvent *a* and *b* have the same source and the same event identifier, and *a* has sequence number `m` and *b* has sequence number `n` where `m` is not equal to `n`, then if `m < n` there were exactly `n-m-1` events of the same event type between the event that triggered *a* and the event that triggered *b*. Such sequence numbers are said to be “fully ordered”.

There are interactions between the generation of sequence numbers for a RemoteEvent object and the ability to see events that occur within the scope of a transaction. Those interactions are discussed later in section 2.5.

The common intent of a call to the `notify` method is to allow the recipient to find out that an occurrence of a kind of event has taken place. The call to the `notify` method is synchronous to allow the party making the call to know if the call succeeded. However, it is not part of the semantics of the call that the notification return can be delayed while the recipient of the call reacts to the occurrence of the event. Simply put, the best strategy on the part of the recipient is to note the occurrence in some way and then return from the `notify` method as quickly as possible.

2.3.3 The `UnknownEventException`

The `UnknownEventException` is thrown when the recipient of a `RemoteEvent` does not recognize the combination of the event identified and the source of the event as something in which it is interested. Throwing this exception has the effect of asking the sender to not send further notifications of this kind of event from this source in the future. This exception is defined as:

```
public class UnknownEventException extends Exception{
    public UnknownEventException(){
        super();
    }
    pubic UnknownEventException(String reason){
        super(reason);
    }
}
```

2.3.4 An Example `EventGenerator` Interface

Registering interest in an event can take place in a number of ways, depending on how the event generator identifies its internal events. There is no single way of identifying the events that are reasonable for all objects and all kinds of events, and so there is no single way of registering interest in events. Because of this, there is no single interface for registration of interest.

However, the interaction between the event generator and the remote event listener does require that some initial information be passed from the registrant to the object that will make the call to its `notify` method.

The `EventGenerator` interface is an example of the kind of interface that could be used for registration of interest in events that can (logically) occur within an object. This is a remote interface that contains one method.

```
public interface EventGenerator extends Remote
{
    public EventRegistration register(long evId,
                                      MarshalledObject handback,
                                      RemoteEventListener toInform,
                                      long leaseLength)
        throws UnknownEventException, RemoteException;
}
```

The one method, `register`, allows registration of interest in the occurrence of an event inside the object. The method takes an `evID` which is used to identify the class of events, an object that is handed back as part of the notification, a reference to an `RemoteEventListener` object, and a `long` integer indicating the leasing period for the interest registration.

The `evID` is a long integer that is obtained by a means that is not specified here. It may be returned by other interfaces or methods, or be defined by constants associated with the class or some interface implemented by the class. If an `evID` is supplied to this call that is not recognized by the `EventGenerator` object, an `UnknownEventException` is thrown. The use of an `long` to identify kinds of events is used only for illustrative purposes--objects may identify events by any number of mechanisms, including identifiers, using separate methods to allow registration in different events, or allowing various sorts of pattern matching to determine what events are of interest.

The second argument of the `register` method is a `MarshalledObject` that is to be handed back as part of the notification generated when an event of the appropriate type occurs. This object is known to the remote event listener, and should contain any information that is needed by the listener to identify the event and to react to the occurrence of that event. This object will be passed back as part of the event object that is passed as an argument to the notify method. By passing a `MarshalledObject` into the `register` method, the re-creation of the object is postponed until the object is needed.

The ability to pass a `MarshalledObject` as part of the event registration should be common to all event registration methods. While there is no single method for identifying events in an object, the use of the pattern in which the remote event listener passes in an object that is passed back as part of the notification is central to the model of remote events presented here.

The third argument of the `EventGenerator` interface's `register` method is a `RemoteEventListener` implementation that is to receive event notifications. The listener may be the object that is registering interest, or it may be some other `RemoteEventListener`, such as a third-party event handler or notification "mailbox." The ability to specify some third party object to handle the notification is also central to this model of event notification, and the capability of specifying the recipient of the notification is also common to all event registration interfaces.

The final argument to the `register` method is a long indicating the requested duration of the registration. This period is a request, and the period of interest actually granted by the event generator may be different. The actual duration of the registration lease is returned as part of the `Lease` object included in the `EventRegistration` object.

The return value of the `register` method is an object of the `EventRegistration` class. This object contains a long identifying the kind of event in which interest was registered (relative to the object granting the registration), a reference to the object granting the registration, and a `Lease` object.

2.3.5 The `EventRegistration` Class

Objects of the class `EventRegistration` are meant to encapsulate the information needed by the client to identify a notification as a response to a registration request and to maintain that registration request. It is not necessary for a method that allows event interest registration to return an object of type `EventRegistration`. However, the class does show the kind of information that needs to be returned in the event model.

The public parts of this class look like

```
public class EventRegistration implements java.io.Serializable {
  public EventRegistration(long eventID,
           Object eventSource,
           Lease eventLease,
           long seqNum);
  public long getID();
  public Object getSource();
  public Lease getLease();
  public long getSequenceNumber();
}
```

The `getID` method returns the identifier of the event in which interest was registered. This, combined with the return value returned by `getSource`, will uniquely identify the kind of event. This information is needed to hand off to third-party repositories to allow them to recognize the event and route it correctly if they are to receive notifications of those events.

The result of the `EventRegistration getID` method should be the same as the result of the `RemoteEvent getID` method, while the result of the `EventRegistration getSource` method should be the same as the `RemoteEvent getSource` method.

The `getSource` method returns a reference to the event generator, which is used in combination with the result of the `getID` method to uniquely identify an event.

The `getLease` returns the `Lease` object for this registration. It is used in lease maintenance.

The `getSequenceNumber` method returns the value of the sequence number on the event kind that was current when the registration was granted, allowing comparison with the sequence number in any subsequent notifications.

2.4 Sequence Numbers, Leasing and Transactions

There are cases in which event registrations are allowed within the scope of a transaction, in such a way that the notifications of these events can occur within the scope of the transaction. This means that other participants in the transaction may see some events whose visibility is hidden by the transaction from entities outside of the transaction. This has an effect on the generation of sequence numbers and the duration of an event registration lease.

An event registration that occurs within a transaction is considered to be scoped by that transaction. This means that any occurrence of the kind of event of interest that happens as part of the transaction will cause a notification to be sent to the recipients indicated by the registration that occurred in the transaction. Such events must have a separate event identification number (the `long` returned in the `RemoteEvent getID` method) to allow third-party store-and-forward entities to distinguish between an event that happens within a transaction and those that happen outside of the transaction. Notifications of these events will not be sent to entities that registered interest in this kind of event outside the scope of the transaction until and unless the transaction is committed.

Because of this isolation requirement of transactions, notifications sent from inside a transaction will have a different sequence number than the notifications of the same events would have outside of the transaction. Within a transaction, all `RemoteEvent` objects for a given kind of event are given a sequence number relative to the transaction, even if the event that triggered the

`RemoteEvent` occurs outside of the scope of the transaction (but is visible within the transaction). One counter-intuitive effect of this is that an object could register for notification of some event, E, both outside a transaction and within a transaction, and receive two distinct `RemoteEvent` objects with different sequence numbers for the same event. One of the `RemoteEvent` objects would contain the event with a sequence number relative to the transaction, while the other would contain the event with a sequence number relative to the source object.

The other effect of transactions on event registrations is to limit the duration of a lease. A registration of interest in some kind of event that occurs within the scope of a transaction should be leased in the same way as other event interest registrations. However, the duration of the registration is the minimum of the length of the lease and the duration of the transaction. Simply put, when the transaction ends (either because of a commit or a rollback) the interest registration also ends. This is true even if the lease for the event registration has not expired and no call has been made to `cancel` the lease.

It is still reasonable to lease event interest registrations, even in the scope of a transaction, because the requested lease may be shorter than the transaction in question. However, no such interest registration will survive the transaction in which it occurs.

2.5 Serialized Forms

The `serialVersionUID` of `RemoteEvent` is 1777278867291906446. The serialized fields are:

- `Object source` - the event source
- `long eventID` - the event id
- `long seqNum` - the event sequence number
- `MarshalledObject handback` - the registration object

The `serialVersionUID` of `UnknownEventException` is 5563758083292687048. There are no serialized fields.

The `serialVersionUID` of `EventRegistration` is 4055207527458053347. The serialized fields are:

- `Object source` - the event source

- `long eventID` - the event id
- `Lease lease` - the granted lease
- `long seqNum` - the current event sequence number

Third-party objects 3

One of the basic reasons for the design presented in the previous chapter was to allow the production of third-party objects, or “agents”, that could be used to enhance a system, built using distributed events and notifications. In this chapter we will look at three examples of such agents, which allow various forms of enhanced functionality without changing the basic interfaces. Each of these agents may be thought of as *distributed event adapters*.

The first example we will look at is a *store-and-forward agent*. The purpose of this object is to act on behalf of the event generator, allowing the event generator to send the notification to one entity (the store-and-forward agent) that will forward the notification to all of the event listeners, perhaps with a particular policy that allows a failed delivery attempt to be re-tried at some later date.

The second example, which we will call a *notification filter*, is an object that may be local to either the event generator or the event listener. This agent gets the notification, and spawns a thread that will respond to the notification, using a method supplied by the object which originally registered interest in events of that kind.

The final object is a *notification mailbox*. This mailbox will store notifications for another object (a remote event listener) until the that object requests that the notifications be delivered. This design allows the listener object that registered interest in the event type to select the times at which a notification can be delivered, without losing any notifications that would have otherwise have been delivered.

3.1 Store-and-Forward agents

A store-and-forward agent enables the object generating a notification to hand off the actual notification of those who have registered interest to a separate object.

This agent can implement various policies for reliability. For example, the agent could try to deliver the notification once (or a small number of times) and, if that call fails, not try again. Or the agent could try and, on notification failure, try again at a pre-set or computed interval of time for some known period of time. Either way, the object in which the event occurred could avoid worrying about the delivery of notifications, only needing to notify the store-and-forward agent (which might be on the same machine and hence more reliably available).

From the point of view of the remote event listener, there is no difference between the notification delivered by a store-and-forward agent and one delivered directly from the object in which the event that generated the original notification occurred. This transparency allows the decision to use a store-and-forward agent to be made by the object generating the notification, independent of the object receiving the notification. There is no need for distributed agreement; all that is required is that the object using the agent know about the agent.

A store-and-forward agent is used by an object that generates notifications. When an object registers interest in receiving notifications of a particular event type, the object receiving that registration will pass the registration along to the store-and-forward agent. This agent will keep track of which objects need to be notified of events that occur in the original object.

When an event of interest occurs in the original object, it need send only a single notification to the store-and-forward agent. This notification can return immediately, with processing further happening inside the store-and-forward agent. The object in which the event of interest occurred will now be freed from informing those who registered interest in the event.

Notification is taken over by the store-and-forward agent. This agent will now consult the list of entities that have registered interest in the occurrence of an event, and send a notification to those entities. Note that these might not be the same as the objects that registered interest in the event; the object that should receive the event notification is specified during the event interest registration.

The store-and-forward agent might be able to make use of network-level multicast (assuming that the `RemoteEvent` object to be returned is identical for multiple recipients of the `notify` call), or might send a separate notification to each of the entities that have registered interest. Different store-and-forward agents could implement different levels of service, from a simple agent that sends a notification and doesn't care if the notification is actually delivered (for example, one that simply caught `RemoteException`s and discards them) to agents that will repeatedly try to send the notification, perhaps using different fall-back strategies, until the notification is known to be successful or some number of tries have been attempted.

The store-and-forward agent does not need to know anything about the kinds of events that are triggering the notifications that it stores and forwards. All that is needed is that the agent implement the `RemoteEventListener` interface and some interface that allows the object producing the initial notification to register with the agent. This combination of interfaces allows such a service to be offered to any number of different objects, without having to know anything about the possible changes in abstract state that might be of interest in those objects.

Note that the interface used by the object generating the original notifications to register with the store-and-forward agent does not need to be standard. Different qualities of service concerning the delivery of notifications may require different registration protocols. Whether or not the relationship between the notification originator and the store-and-forward agent is leased or not is also up to the implementation of the agent. If the relationship is leased, lease renewal requests would need to be forwarded to the agent.

In fact, an expected pattern of implementation would be to place a store-and-forward agent on every machine on which objects were running that could produce events. This agent, which could be running in a separate JVM (on hardware that supported multiple processes) could off-load the notification-generating objects from the need to send those notifications to all objects that had registered interest. It would also allow for consistent handling of delivery guarantees across all objects on a particular machine. Since the store-and-forward agent is on the same machine as those objects using the agent, the possibilities of partial failure brought about by network problems (which wouldn't effect communication between objects on the same machine) and server machine failure (which would induce total, rather than partial, failure in

this case) are limited. This allows the reliability of notifications to be off-loaded to these agents instead of being a problem that needs to be solved by all of the objects using the notification interfaces.

A store-and-forward agent does require an interface that allows the agent to know what notifications it is supposed to send, the destinations of those notifications, and on whose behalf those notifications are being sent. Since it is the store-and-forward agent that is directing notification calls to the individual recipients, the agent will also need to hold the `Object` (if any) that was passed in during interest registration to be returned as part of the `RemoteEvent` object.

In addition, the store-and-forward agent could be the issuer of `Lease` objects to the object registering interest in some event. This could off-load any lease renewal calls from the original recipient of the registration call, which would only need to know when there were no more interest registrations of a particular event kind remaining in the store-and-forward agent.

3.2 Notification Filters

Similar to a store-and-forward agent is a notification filter, which can be used by either the generator of a notification or the recipient to intercept notification calls, do processing on those calls, and act in accord with that processing (perhaps forwarding the notification, or even generating new notifications).

Again, such filters are made possible because of the uniform signature of the method used to send all notifications, and because of the ability of an object to indicate the recipient of a notification when registering for a notification. This uniformity and indirection allow the composition of third-party entities. A filter could receive events from a store-and-forward agent without the client of the original registration knowing about the store-and-forward agent or the server in which the notifications are generated knowing about the filter. This composition can be extended further; store-and-forward agents could use other store-and-forward agents, and filters can themselves receive notifications from other filters.

3.2.1 Notification Multiplexing

One example of such a filter is one that can be used to concentrate notifications in a way to help minimize network traffic. If a number of different objects on a single machine are all interested in some particular kind of event, it could

make sense to create a notification filter that would register interest in the event. When a notification was received by the filter, it would forward the notification to each of the (machine local) objects that had expressed interest.

3.2.2 Notification Demultiplexing

Another example of such a filter is an object that generates an event in response to a series of events that it has received. There might be an object that is only interested in some particular sequence of events in some other object or group of objects. This object could register interest in all of the different kinds of events, asking that the notifications be sent to a filter. The purpose of the filter is to receive the notifications and, when the notifications fit the desired pattern (as determined by some class passed in from the object that has asked the notifications be sent to the filter), generate some new notification that is delivered to the client object.

3.3 Notification Mailboxes

The purpose of a notification mailbox is to store the notifications sent to an object until such time as the object for which the notifications were intended desires delivery.

Such delivery can be in a single batch, with the mailbox storing any notifications received after the last request for delivery until the next request is received. Alternatively, a notification mailbox can be viewed as a faucet, with notifications turned on (delivering any that have arrived since the notifications were last turned off) and then delivering any subsequent notifications to an object immediately, until told by that object to hold the notifications.

The ability to have notification mailboxes is important in a system that allows objects to be deactivated (for example, to be saved to stable storage in such a way that they are no longer taking up any computing resource) and re-activated. The usual mechanism for activating an object is a method call. Such activation can be expensive both in time and computing resources; it is often too expensive to be justified for the receipt of what would otherwise be an asynchronous event notification. An event mailbox can be used to insure that an object will not be activated merely to handle an event notification.

Use of a mailbox is simple; the object registering interest in receiving an event notification simply gives the mailbox as the place to send the notifications. The mailbox can be made responsible for renewing leases while an object is

inactive, and for storing all (or the most recent, or the most recent and the count of other) notifications for each type of event of interest to the object. When the object indicates that it wishes to receive any notifications from the mailbox, those notifications can be delivered. Delivery can continue until the object requests storage to occur again, or storage can resume automatically.

Such a mailbox is a type of filter. In this case, however, the mailbox filters over time rather than over events. A pure mailbox need not be concerned with the kinds of notifications that it stores. It simply holds the `RemoteEvent` objects until they are wanted.

It is because of mailboxes and other client-side filters that the information returned from an event registration needs to include a way of identifying the event and the source of the event. Such client-side agents need a way of distinguishing between the events they are expected to receive and those that should generate an exception to the sender. This distinction cannot be made without some simple way of identifying the event and the object of origin.

3.4 Compositionality

All of the above third-party entities work because of two simple features of the `RemoteEventListener` interface:

- There is a single method, `notify`, that passes a single type of object, `RemoteEvent` (or a subtype of that object) for all notifications
- There is a level of indirection in delivery allowed by the separate specification of a recipient in the registration method that allows the client of that call to specify a third-party object to contact for notifications

The first of these features allows the composition of notification handlers to be “chained,” beginning with the object that generates the notification. Since the ultimate recipient of the event is known to be expecting the event through a call to the single `notify` method, other entities can be composed and interposed in the call chain as long as they produce this call with the right `RemoteEvent` object (which will include a field indicating the object at which the notification originated). Because there is a single method call for all notifications, third-party handlers can be produced to accept notifications of events without having to know the kind of event that has occurred, or any other detail of the event.

Compositionality in the other direction (driven by the recipient of the notification) is enabled by allowing the object registering interest to indicate the first in an arbitrary chain of third-parties to receive the notification. Thus the recipient can build a chain of filters, mailboxes, and forwarding agents to allow any sort of delivery policy that object desires, and then register interest with an indication that all notifications should be delivered to the beginning of that chain. From the point of view of the object in which the notification originates, the series of objects the notification then goes through is unknown and irrelevant.

Integration with JavaBeans™ Components

4

As noted in the second chapter, distributed notification differs from local notification (like the notification used in user interface programming) in a number of ways. In particular, a distributed notification may be delayed, dropped, or otherwise fail between the object in which the event occurred and the object that is the ultimate recipient of the notification of that event. Additionally, a distributed event notification may require handling by a number of third-party objects between the object interested in the notification and the object that generates the notification. These third-party objects need to be able to handle arbitrary events, and so from the point of view of the type system all of the events must be delivered in the same fashion.

While this model differs from the event model used for user interface tools such as the AWT or Java™ Foundation Classes (JFC), such a difference in model is to be expected. The event model for such user interface toolkits was never meant to allow the components that communicate using these local event notifications to be distributed across virtual or physical machines; indeed, such systems assume that the event delivery will be fast, reliable, and not open to the kinds of partial failures or delays that are common in the distributed case.

In between the requirements of a local event model and the distributed event model presented here is the event model used by software components to communicate changes in state. The delegation event model, which is the event model for JavaBeans™ components, written in the Java programming language, is built as an extension of the event model used for AWT and JFC. This is completely appropriate, as most JavaBeans components will be located

in a single address space, and can assume the communication of events between components will meet the reliability and promptness requirements of that model.

However, it is also possible that JavaBeans components will be distributed across virtual, and even physical, machines. In such a case, the assumption that the event propagation will be either fast or reliable can lead to subtle program errors that will not be found until the components are deployed (perhaps on a slow or unreliable network). In such cases, an event and notification model such as that found in this specification is more appropriate.

One approach would be to add a second event model to the JavaBeans component specification, which dealt only with distributed events. While this would have the advantage of exporting the difference between local and remote components to the component builder, it would also complicate the JavaBeans component model unnecessarily.

In this chapter, we show how the current distributed event model can be fit into the existing Java platform's event model. While the mapping is not perfect (nor can it be, since there are essential differences between the two models), it will allow the current tools used to assemble JavaBeans components to be used when those components are distributed.

4.1 Differences with the JavaBeans Component Event Model

The JavaBeans component event model is derived from the event model used in the AWT in JDK 1.2. The model is characterized by:

- Propagation of event notifications from sources to listeners by Java technology method invocations on the target listener objects
- Identification of the kind of event notification by using a different method in the listener being called for each kind of event
- Encapsulation of any state associated with an event notification in an object that inherits from `java.util.EventObject` and which is passed as the sole argument of the notification method
- Identification of event sources by the convention of those sources defining registration methods, one for each kind of event in which interest can be registered, that follow a particular design pattern

The distributed event and notification model we have defined is similar in a number of ways:

- Distributed event propagation is accomplished by the use of `Remote` methods
- State passed as part of the notification is encapsulated in an object that is derived from `java.util.EventObject` and is passed as the sole argument of the notification method
- The `RemoteEventListener` interface extends the more basic `java.util.EventListener` interface

However, there are also differences between the JavaBeans component event model and the distributed event model proposed here:

- Identification of the kind of event is accomplished by passing an identifier from the source of the notification to the listener; the combination of the object in which the event occurred and the identifier uniquely identify the kind of event
- Notifications are accomplished through a single method, `notify`, defined in the `RemoteEventListener` interface rather than by a different method for each kind of event
- Registration of interest in a kind of event is for a (perhaps renewable) period of time, rather than being for a period of time bound by the active cancellation of interest
- Objects registering interest in an event can, as part of that registration, include an object that will be passed back to the recipient of the notification when an event of the appropriate type occurs

Most of these differences in the two models can be directly traced to the distributed nature of the events and notifications defined in this specification.

For example, reliability and recovery of the distributed notification model is based on the ability of creating third-party objects, such as those discussed in the last chapter, which can provide those guarantees. However, for those third-party objects to be able to work in general cases, the signature for a notification must be the same for all of the event notifications that are to be handled by that third-party. If we were to follow the JavaBeans component model of having a different method for each kind of event notification, third party objects would need to support every possible notification method including those that had not yet been defined when the third-party object was implemented. This is clearly impossible.

Note that this is not a weakness in the JavaBeans component event model, merely a difference required by the different environments in which the event models are assumed to be used. The JavaBeans component event model, like the AWT model on which it is based, assumes that the event notification is being passed between objects in the same address space. Such notifications do not need various delivery and reliability guarantees—delivery can be considered to be (virtually) instantaneous, and can be assumed to be fully reliable.

Being able to send event notifications through a single `Remote` method also requires that the events be identified in some way other than the signature of the notification delivery method. This leads to the inclusion of an event identifier in the event object. Since the generation of these event identifiers cannot be guaranteed to be globally unique across all of the objects in a distributed system, they must be made relative to the object in which they are generated, thus requiring the combination of the object of origin and the event identifier to completely identify the kind of event.

The sequence number being included in the event object is also an outgrowth of the distributed nature of the interfaces. Since no distributed mechanism can guarantee reliability, there is always the possibility that a particular notification will not be delivered, or could be delivered (by some notification agent) more than once. This is not a problem in the single address-space environment of AWT and JavaBeans components, but requires the inclusion of a sequence number in the distributed case.

4.2 Making a JavaBeans Component Event out of a Distributed Event

To translate between the event models is fairly straightforward. All that is required is:

- Allow an event listener to map from a distributed event listener to the appropriate call to a notification method
- Allow creation of a `RemoteEvent` from the event object passed in the JavaBeans component event notification method
- Allow creation of a JavaBeans component event object from a `RemoteEvent` object without loss of information

Each of these is fairly straightforward, and can be accomplished in a number of ways.

More complex matings of the two systems could be undertaken, including third-party objects that kept track of the interest registrations made by remote objects, and implemented the corresponding JavaBeans component event notification methods by making the remote calls to the `RemoteEventListener notify` method with properly constructed `RemoteEvent` objects. Such objects would need to keep track of the event sequence numbers, and would need to deal with the additional failure modes that are inherent in distributed calls. However, their implementation would be fairly straightforward, and would fit into the JavaBeans component model of event adaptors.