will not be longer. The value of the `leaseDuration` argument must be positive, `Lease.FOREVER`, or `Lease.ANY`; otherwise, an `IllegalArgumentException` will be thrown. Two calls to the `createLeaseRenewalSet` method will never return objects that are equal. The set's lease is obtained through a method provided by the set.

`LeaseRenewalSet` defines the interface to the sets created by the lease renewal service. This interface is not a remote interface. Each implementation of the renewal service exports proxy objects that implement the `LeaseRenewalSet` interface local to the client and use an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal RMI remote interface semantics except where explicitly noted. The proxy objects for two sets are equal (using the `equals` method) if they are proxies for the same set created by the same renewal service. Any method that communicates with the remote server should throw a `NoSuchObjectException` if the set no longer exists. If a client receives a `NoSuchObjectException` from one of the operations on a lease renewal set, the client can infer that the set has been destroyed; however, it should *not* infer that the renewal service has been destroyed.

```
package net.jini.lease;

public interface LeaseRenewalSet {
    final public static long RENEWAL_FAILURE_EVENT_ID = 0;
    final public static long EXPIRATION_WARNING_EVENT_ID = 1;

    public void renewFor(Lease leaseToRenew,
                         long  desiredDuration,
                         long  renewDuration)
        throws RemoteException;

    public void renewFor(Lease leaseToRenew,
                         long  desiredDuration)
        throws RemoteException;

    public EventRegistration setExpirationWarningListener(
            RemoteEventListener listener,
            long                minWarning,
            MarshalledObject    handback)
        throws RemoteException;

    public void clearExpirationWarningListener()
        throws RemoteException;
```

```
    public EventRegistration setRenewalFailureListener(
            RemoteEventListener listener,
            MarshalledObject    handback)
        throws RemoteException;

    public void clearRenewalFailureListener()
        throws RemoteException;

    public Lease remove(Lease leaseToRemove)
        throws RemoteException;

    public Lease[] getLeases()
        throws LeaseUnmarshalException, RemoteException;

    public Lease getRenewalSetLease();
}
```

Leases can be added to the set through the `renewFor` methods. There are two forms of this method: a three-argument form and a two-argument form. The three-argument form will be described first. The `leaseToRenew` argument specifies the lease to be renewed. An `IllegalArgumentException` will be thrown if the lease has not expired and was granted by the renewal service itself. An `IllegalArgumentException` will also be thrown if the lease is currently a member of another set allocated by the same renewal service. If `leaseToRenew` is `null`, a `NullPointerException` will be thrown.

The `desiredDuration` parameter is the number of milliseconds that the client would like for the client lease to remain in the set. It is used to calculate the client lease's desired expiration by adding `desiredDuration` to the current time (as viewed by the service). If this causes an overflow, a desired expiration of `Long.MAX_VALUE` will be used. Unlike a lease duration, the desired duration is unilaterally specified by the client, not negotiated between the client and the service. Note that a negative value for `desiredDuration` (including `Lease.ANY`) will result in a desired expiration that is in the past. This will cause the client lease to be dropped immediately from the set and will not result in an exception. A renewal failure event will be generated if and only if the client's actual expiration is before its desired expiration.

If the actual expiration time of the client lease being added to the set is before both the current time (as viewed by the renewal service) and the client lease's desired expiration time, the method will return normally. However, the client lease will be dropped from the set, and a renewal failure event will be generated. If the

actual expiration time is before the current time and equal to or after the desired expiration time, the method will return normally, the client lease will be dropped from the set, and no event will be generated.

A `desiredDuration` of `Long.MAX_VALUE` does not imply that the client lease will remain in the set forever. The client lease will be ejected from the set if the set is destroyed, the client lease itself expires, the client lease is removed from the set, or the renewal service makes a renewal attempt on the client lease that results in a definite exception.

The `renewDuration` is the renewal duration to associate with the client lease (in milliseconds). If `desiredDuration` is exactly `Long.MAX_VALUE`, the `renewDuration` may be any positive number or `Lease.ANY`; otherwise it must be a positive number. If these requirements are not met, the renewal service will throw an `IllegalArgumentException`.

Calling `renewFor` with a lease that is equivalent to a client lease already in the set will associate the existing client lease in the set with the new desired duration and renew duration. The original copy of the client lease is not replaced with the new one. These semantics also allow `renewFor` to be used in an idempotent fashion.

The two-argument form of `renewFor` is equivalent to

```
renewFor(leaseToRenew, desiredDuration, Lease.FOREVER)
```

Client leases get returned to clients in a number of ways (via `remove` and `getLeases` calls, as components of events, etc.). The serial format of client leases returned to clients may be either `Lease.DURATION` or `Lease.ABSOLUTE`. In particular it may be necessary to use the `Lease.ABSOLUTE` format if the implementation has access to the client lease only in marshalled form and is unable to unmarshal the client lease before sending it to the client.

Whenever a client lease gets returned to a client, its actual expiration should reflect either:

- The result of the last recorded successful renewal of the client lease performed by the renewal service; or
- The expiration time the client lease originally had when it was added to the set, if the renewal service has been unable to successfully renew the client lease and record the result

Although it is impossible for a renewal service to guarantee that all renewal attempts will be recorded, persistent implementations should attempt to keep the interval between the renewal of a client lease and the logging of the result to a minimum.

Client leases are removed from the set by using the `remove` method. Removal from the set will not cause the lease to be cancelled. The method will return the lease that is being removed. If the lease is not in the set, `null` will be returned; and this call will not be blocked by in-progress renewal attempts. As a result, a client lease removed by this method might be renewed after the method has returned. Implementations should keep the window where renewals of removed leases could occur as small as possible.

The `getLeases` method returns all the client leases in the set at the time of the call, as an array of type `Lease`. If one or more of the `Leases` in the array cannot be deserialized, a `LeaseUnmarshalException` is thrown.

```
package net.jini.lease;

public class LeaseUnmarshalException extends Exception {
    public LeaseUnmarshalException(
            Lease[]            leases,
            MarshalledObject[] marshalledLeases,
            Throwable[]        exceptions) {…}
    public LeaseUnmarshalException(
            Lease[]            leases,
            MarshalledObject[] marshalledLeases,
            Throwable[]        exceptions,
            String             message) {…}

    public Lease[] getLeases() {…}
    public MarshalledObject[] getMarshalledLeases() {…}
    public Throwable[] getExceptions() {…}
}
```

The leases that could be successfully deserialized will be returned by the `getLeases` method of the exception. If no leases could be deserialized, a zero-length array will be returned. The leases that could not be deserialized will be returned in the form of `MarshalledObjects` by the `getMarshalledLeases` method of the exception. For each element of the array returned by the `getMarshalledLeases` method, the corresponding element of the array returned by the `getExceptions` method will hold a `Throwable` that indicates why the given lease could not be deserialized.

Throwing a `LeaseUnmarshalException` represents a (possibly transient) failure in the ability to unmarshal one or more client leases in the set; it does not necessarily imply anything about the state of the renewal service or the set that threw the exception.

The `getRenewalSetLease` method of `LeaseSet` returns the lease associated with the set itself. This method does not make a remote call.

LR.2.1 Events

The lease renewal service does not support multiple simultaneous event listener registrations for the same kind of event. Although it would be useful in some limited circumstances, to do so would require event registrations to be leased separately from the set they are associated with. For the average client of the lease renewal service, this ability would increase the number of leases that it would have to manage. Since the renewal service is based on the premise that some clients have difficulty managing their own leases, increasing the number of leases that a client would need to manage could significantly complicate the implementation of those clients. Because there can be at most one listener for each kind of event, a given set provides a `set/clear` interface instead of the more common `addListener/removeListener` or `addListener/lease.cancel` interfaces.

The source field of each event generated by a lease renewal service is the renewal set that the event is associated with. In the case of an expiration warning event, this is the set that is about to expire. In the case of a renewal failure event, this is the set the client lease was in when the event occurred. Note that the value of the source field will in general be a copy of the set in question, the `equals` method will return `true` for any other copies of the set the client has in its possession, but in general it will not be the same object (that is, comparing two sets using == will usually return `false`).

The event ID `LeaseRenewalSet.EXPIRATION_WARNING_EVENT_ID` is used for all expiration warning events. One event ID is used because there is only one kind of expiration warning event. Similarly, all renewal failure events will have the event ID `LeaseRenewalSet.RENEWAL_FAILURE_EVENT_ID`.

Because all of the expiration warning events generated by a given set will have the same source and event ID, the sequence number of any given expiration warning event generated by the set will be different from the sequence number of any other expiration warning event generated by the set. Similarly, the sequence number of any renewal failure event generated by a given set will be different from the sequence number of any other renewal failure event generated by the set. Two different events with the same source and event ID will have different sequence numbers even if different event registration were in effect when each event was generated.

If a `RemoteEventListener` registered for a renewal failure or expiration warning event throws an `UnknownEventException`, this action will only clear the specific event registration. It will not cancel the lease on the renewal set or affect

any other event registration on the set. If the listener throws a bad object exception, the renewal service may clear that specific event registration; it will not clear any registration associated with other listeners, nor will it cancel the lease on the associated renewal set.

If an event listener is replaced and one or more event delivery attempts on the original listener failed, implementations may choose to send some or all of these events to the new listener.

Event listeners may receive notification of events that they are no longer registered to receive, if those events occurred before they were unregistered. Implementations should keep the window where such notifications could occur as small as possible.

The `setExpirationWarningListener` method of `LeaseRenewalSet` allows the client to register for notification of the approaching expiration of the *set's* lease. Expiration warning events are not generated for client leases. The `listener` argument specifies which listener should be notified when the set's lease is about to expire. The `minWarning` argument specifies the minimum number of milliseconds before set lease expiration that the first event delivery attempt should be made by the service. The service may also make subsequent delivery attempts if the first and any subsequent attempts resulted in an indefinite exception. The `minWarning` argument must be zero or a positive number; if it is not, an `IllegalArgumentException` must be thrown. If the current expiration of the set's lease is less than `minWarning` milliseconds away, the event will occur immediately (though it will take time to propagate to the handler).

The `handback` argument to `setExpirationWarningListener` specifies an object that will be part of the expiration warning event notification. This mechanism is detailed in *The Jini Technology Core Platform Specification, "Distributed Events"*.

The `setExpirationWarningListener` method returns the event registration for this event. The `Lease` object associated with the registration will be equivalent (in the sense of the `equals` method) to the `Lease` on the renewal set. Because the event registration shares a lease with the set, clients that want to just remove their expiration warning registration without destroying the set should use the `clearExpirationWarningListener` method described below, instead of cancelling the registration's lease. The event ID returned with the registration will be `LeaseRenewalSet.EXPIRATION_WARNING_EVENT_ID`. The source of the registration will be the set. The method will throw a `NullPointerException` if the `listener` argument is `null`. If an event handler has already been specified for this event, the current registration is replaced with the new one. Because both registrations are for the same kind of event, the events sent to the new registration must be in the same sequence as the events sent to the old registration.

The `clearExpirationWarningListener` method of `LeaseRenewalSet` removes the event registration currently associated with the approaching expiration of the set's lease. It is acceptable to call this method even if there is no active registration.

The `setRenewalFailureListener` method of `LeaseRenewalSet` allows the client to register for the event associated with the failure to renew a client lease in the set. These events are generated when a client lease in the set reaches its actual expiration before its desired expiration or when the service attempts to renew a client lease and gets a definite exception. The `listener` argument specifies the listener to be notified if a client lease could not be renewed.

The `handback` argument to `setRenewalFailureListener` specifies an object that will be part of the renewal failure event notification. This mechanism is detailed in *The Jini Technology Core Platform Specification, "Distributed Events"*.

The `setRenewalFailureListener` method returns the event registration for this event. The `Lease` object associated with the registration will be equivalent (in the sense of the `equals` method) to the `Lease` on the renewal set. Because the event registration shares a lease with the set, clients that want to just remove their expiration warning registration without destroying the set should use the `clearRenewalFailureListener` method (described below) instead of cancelling the registration's lease. The registration ID returned with the registration will be `LeaseRenewalSet.RENEWAL_FAILURE_EVENT_ID`. The source of the registration will be the set. The method will throw `NullPointerException` if the `listener` argument is `null`. If an event handler has already been specified for this event, the current registration is replaced with the new one. Because both registrations are for the same kind of event, the events sent to the new registration must be in the same sequence as the events sent to the old registration.

The `clearRenewalFailureListener` method of `LeaseRenewalSet` removes the event registration currently associated with the failure to renew client leases. It is acceptable to call this method even if there is no active registration.

```
package net.jini.lease;

public class ExpirationWarningEvent extends RemoteEvent {
    public ExpirationWarningEvent(
            LeaseRenewalSet  source,
            long             seqNum,
            MarshalledObject handback) {…}
    public Lease getRenewalSetLease() {…}
}
```

`ExpirationWarningEvent` objects are passed to the event handlers specified in calls to the `LeaseRenewalSet` method, `setExpirationWarningListener`. The `ExpirationWarningEvent` is a subclass of `RemoteEvent` and adds no additional state. Because the source of a `ExpirationWarningEvent` is the set that is about to expire, the lease that needs to be renewed can be obtained by: calling `getSource`, casting the result to a `LeaseRenewalSet` and then invoking the set's `getRenewalSetLease` method. The convenience method `getRenewalSetLease` in `ExpirationWarningEvent` uses this technique to retrieve the lease on the set. The `Lease` object returned will be equivalent (in the sense of the `equals` method) to other `Lease` objects associated with the set but may not be the same object. One notable consequence of having two different objects is that the `getExpiration` method of the `Lease` object returned by the event's `getRenewalSetLease` method may return a different time than the `getExpiration` methods of other `Lease` objects granted on the same set.

The expiration time associated with the `Lease` object returned by the `getRenewalSetLease` method will reflect the expiration the lease had when the event occurred. Renewal calls may have changed the expiration time of the underlying lease between the time when the event was generated and when it was delivered.

Other aspects of the event's state are described in *The Jini Technology Core Platform Specification, "Distributed Events"*. Sequence numbers for a given event ID are increasing. If there is no gap between two sequence numbers, no events have been missed; if there is a gap, events might (but might not) have been missed.

```
package net.jini.lease;

public abstract class RenewalFailureEvent
    extends RemoteEvent
{
    public RenewalFailureEvent(LeaseRenewalSet  source,
                               long             seqNum,
                               MarshalledObject handback) {…}
    abstract public Lease getLease()
        throws IOException, ClassNotFoundException;
    abstract public Throwable getThrowable()
        throws IOException, ClassNotFoundException;
}
```

`RenewalFailureEvent` objects are passed to the event handlers specified in calls to the `LeaseRenewalSet` method, `setRenewalFailureListener`. The `RenewalFailureEvent` is a subclass of `RemoteEvent`, adding two additional

items of abstract state: the client lease that could not be renewed before expiration and the `Throwable` object that was thrown by the last recorded renewal attempt (if any). The client lease is returned by the `getLease` method, and the `Throwable` object is returned by the `getThrowable` method. If the `Throwable` object is `null`, it can be assumed that during the time between the last-recorded, successful renewal (or when the client lease was added to the set if there have been no renewals) and the actual expiration time of the client lease the renewal service was either unable to attempt a renewal of the client lease, or that it attempted a renewal but was unable to record the result.

Both the `getLease` and `getThrowable` methods may throw `IOException` or `ClassNotFoundException.` This declaration allows implementations to delay unmarshalling this state until it is actually needed. Once either method of a given `RenewalFailureEvent` object returns normally, future calls on that method must return the same object and may not throw an exception.

If the renewal service was able to renew the client lease and record the result before the event occurred, the expiration time of the `Lease` object returned by the event's `getLease` method will reflect the result of the last-recorded successful renewal call. Note that this time may be distorted by clock skew between hosts if it is currently set to use the `Lease.ABSOLUTE` serial format. If the `Lease` object is using the `Lease.DURATION` serial format, and the event only unmarshals the lease when `getLease` is called, the expiration time may be distorted if a long time has passed between the time the event was generated by the renewal service and when the client called `getLease`. When a renewal failure event is generated for a given lease, that lease is removed from the set.

The event's other state is described in *The Jini Technology Core Platform Specification, "Distributed Events"*. Sequence numbers for a given event ID are increasing. If there is no gap between two sequence numbers, no events have been missed; if there is a gap, events might (but might not) have been missed.

LR.2.2 Serialized Forms

Class	serialVersionUID	Serialized Fields
`RenewalFailureEvent`	889145704195932943L	*none*
`ExpirationWarningEvent`	-2020487536756927350L	*none*

`LeaseUnmarshalException`	-6736107321698417489L	`Lease[]unmarshalledLeases`
		`MarshalledObject[]`
		`stillMarshalledLeases`
		`Throwable[] exceptions`

EM

Jini Event Mailbox Service Specification

EM.1 Introduction

THE *The Jini Technology Core Platform Specification,* "Distributed Events" states the ability to interpose third-party objects, or "agents," into an event notification chain as one of its design goals. This specification also describes a notification mailbox object, which stores and forwards event notifications on behalf of other objects, as an example of a useful third-party agent. These mailbox objects can be particularly helpful for objects that need more control over how and when they receive event notifications.

For example, it would be impossible to send event notifications to a transient entity that has detached itself from a system of Jini technology-enabled services and/or devices (*Jini system*). In such a situation an entity could employ the services of an event mailbox to store event notifications on its behalf before leaving the system. Upon rejoining the Jini system, the entity could then contact the event mailbox to retrieve any collected events that it would otherwise have missed. Similarly, an entity that wishes to deactivate could use an event mailbox to collect event notifications on its behalf while dormant.

Like other Jini technology-enabled services (*Jini services*), the event mailbox service will grant its services only for a limited period of time without an active expression of continuing interest. Therefore, event mailbox clients still need to renew their leases if they intend to maintain the mailbox's services beyond the initially granted lease period. Any resources (for example, remote objects or storage space) associated with a particular client can be freed once the client's lease has expired or been cancelled. In the previous usage scenarios, it might also benefit a transient or deactivatable entity to employ the services of a lease renewal service

(see the *Jini Lease Renewal Service Specification)* to help mitigate the issue of lease maintenance.

The remainder of this specification defines the requirements, interfaces, and protocols of the event mailbox service.

EM.1.1 Goals and Requirements

The requirements of the set of interfaces specified in this document are:

- To define a service that is capable of storing event notifications on behalf of its clients and capable of delivering stored event notifications to those clients upon request
- To provide this service in such a way that it can be used by entities that are temporarily unable or unwilling to receive event notifications
- To provide a service that complies with the policies embodied in the Jini technology programming model

The goals of this specification are:

- To describe the event mailbox service
- To provide guidance in the use and deployment of the event mailbox service

EM.1.2 Other Types

The types defined in the specification of the event mailbox service are in the `net.jini.event` package. This specification assumes knowledge of *The Jini Technology Core Platform Specification,* "Distributed Events" and *The Jini Technology Core Platform Specification, "Distributed Leasing"*. The following object types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
java.rmi.NoSuchObjectException
java.rmi.RemoteException
net.jini.core.event.RemoteEvent
net.jini.core.event.RemoteEventListener
net.jini.core.lease.Lease
net.jini.core.lease.LeaseDeniedException
```

EM.2 The Interface

THE `EventMailbox` defines the interface to the event mailbox service. Through this interface, other Jini services and clients may request that event notification management be performed on their behalf. This interface belongs to the `net.jini.event` package, and any service implementing this interface must comply with the definition of a Jini service. This interface is not a remote interface; each implementation exports a proxy object that implements this interface local to the client, using an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal Java Remote Method Invocation (RMI) interface semantics and can therefore be implemented directly using RMI (except where explicitly noted). Two proxy objects are equal (using the `equals` method) if they are proxies for the same event mailbox service.

```
package net.jini.event;

public interface EventMailbox
{
    MailboxRegistration register(long leaseDuration)
        throws RemoteException, LeaseDeniedException;
}
```

Event mailbox clients wishing to use the mailbox service first register themselves with the service using the `register` method. Clients then use the methods of the returned `MailboxRegistration` object (a *registration*) in order to:

- Manage the lease for this particular registration
- Obtain a `RemoteEventListener` reference that can be registered with *event generators* (that is, objects that support event notification for changes in their abstract state). This listener will store any received notifications for this particular registration.
- Enable or disable the delivery of any stored notifications for this particular registration

EM.3 The Semantics

To employ the event mailbox service, a client must first register with the event mailbox service by invoking the `EventMailbox` interface's only method, `register`. Each invocation of the `register` method produces a new registration.

The `register` method may throw a `RemoteException` or a `LeaseDeniedException`. Typically, a `RemoteException` occurs when there is a communication failure between the client and the event mailbox service. If this exception does occur, the registration may or may not have been successful. A `LeaseDeniedException` is thrown if the event mailbox service is unable or unwilling to grant the registration request. It is implementation specific as to whether or not subsequent attempts (with or without the same argument) are likely to succeed.

Each registration with the event mailbox service is persistent across restarts or crashes of the event mailbox service, until the lease on the registration expires or is cancelled.

The `register` method takes a single parameter of type `long` that represents the requested initial lease duration for the registration, in milliseconds. This duration value must be positive (except for the special value of `Lease.ANY`). Otherwise, an `IllegalArgumentException` is thrown.

Every method invocation on an event mailbox service (whether the invocation is directly on the service, or indirectly on a `MailboxRegistration` that the service has created) is atomic with respect to other invocations.

EM.4 Supporting Interfaces and Classes

THE `register` method returns an object that implements the interface `MailboxRegistration`. It is through this interface that the client controls its registration and notification management with the event mailbox service.

```
package net.jini.event;

public interface MailboxRegistration
{
    Lease getLease();
    RemoteEventListener getListener();
    void enableDelivery(RemoteEventListener target)
        throws RemoteException;
    void disableDelivery() throws RemoteException;
}
```

The `MailboxRegistration` interface is not a remote interface. Each implementation of the event mailbox service exports proxy objects that implement this interface local to the client. These proxies use an implementation-specific protocol to communicate with the remote server. All of the remote proxy methods obey normal RMI interface semantics and can therefore be implemented using RMI. Two proxy objects are equal (using the `equals` method) if they are proxies for the same registration, created by the same event mailbox service.

Each remote method of this interface may throw a `RemoteException`. Typically, this exception occurs when there is a communication failure between the client and the event mailbox service. Whenever a method invocation results in a `RemoteException`, the method may or may not have successfully completed.

Any invocation of a remote method defined in this interface will result in a `NoSuchObjectException` if the client's registration with the event mailbox service has expired or has been cancelled. Note that upon receipt of a `NoSuchObjectException`, the client can assume that the registration no longer exists; the client cannot assume that the event mailbox service itself no longer exists.

EM.4.1 The Semantics

The `getLease` method returns the `Lease` object associated with the registration. The client can renew or cancel the registration with the mailbox service through the `Lease` object returned by this method (see *The Jini Technology Core Platform Specification, "Distributed Leasing"*). This method is not remote and takes no arguments.

The `getListener` method returns an object that implements the interface `RemoteEventListener`. This object, referred to as a *mailbox listener*, can then be submitted as the `RemoteEventListener` argument to an event generator's registration method(s) (see *The Jini Technology Core Platform Specification,* "Distributed Events"). Subsequent calls to this method will return equivalent objects (in the `equals` sense). Note that mailbox listeners generated by different registrations will not be equal. This method is not remote and takes no arguments.

The valid period of use for a mailbox listener is tied to the associated registration's lease. A `NoSuchObjectException` will be thrown if an attempt is made to invoke the `notify` method on a mailbox listener whose associated lease has terminated.

Mailbox listener references, just like their associated registrations, are persistent across server restarts or crashes until their associated registration's lease terminates.

The `enableDelivery` method allows a client to initiate delivery of event notifications (received on its behalf by this particular registration) to the client-specified listener, referred to as the *target listener*. This method takes a single argument of type `RemoteEventListener`. Subsequent calls to this method simply replace the registration's existing target listener, if any, with the specified target listener. Passing `null` as the listener argument has the same effect as disabling delivery (see below).

Resubmitting a mailbox listener back to the same mailbox service that generated it will result in an `IllegalArgumentException` being thrown. This is necessary to prevent a recursive event notification chain. Therefore, the event mailbox service must keep track of any listener objects that it generates and reject the resubmission of those objects.

Once enabled, event delivery remains enabled until it is disabled. Any events received while delivery is enabled will also be scheduled for delivery.

Event delivery guarantees with respect to exception handling, ordering, and concurrency are implementation specific and are not specified in this document. However, implementations are encouraged to support the following functionality. If an event delivery attempt produces an indefinite exception, then reasonable efforts should be made to successfully redeliver the event until the associated registration's lease terminates. On the other hand, if an event delivery attempt pro-

duces a definite exception, then event delivery should be disabled for the associated registration until it is explicitly enabled again.

Also, implementations may concurrently deliver event notifications to the same target listener, which implies that events may be sent in a different order than the order in which they were originally received. Hence, it is the target listener's responsibility to guard against potential concurrent, out-of-order event delivery.

Similarly, implementations are encouraged to support this method's intended semantics regarding listener replacement. That is, a mailbox client can reasonably assume that listener replacement has occurred upon successful return from this method and can therefore safely unexport the previous listener object. This also implies that any in-progress delivery attempts to the previous listener are either successfully cancelled before returning from this method (blocking), or subsequently retried using the replacement listener after returning from this method (non-blocking). Note that the non-blocking case can potentially allow the previous listener to be notified after successfully returning from this method.

The `disableDelivery` method allows the client to cease event delivery to the existing target listener, if any. It is acceptable to call this method even if no target listener is currently enabled. This method takes no arguments.

Again, event delivery guarantees are implementation specific and are not specified in this document. Implementations are encouraged to support the method's intended semantics regarding delivery suspension. That is, a mailbox client can reasonably assume that event delivery has been suspended upon successful return from this method and can therefore safely unexport the previously enabled listener object if desired. This also implies that any in-progress delivery attempts to the previously enabled listener are either successfully cancelled before returning from this method (blocking), or subsequently retried using the next enabled listener after returning from this method (non-blocking). Note that the non-blocking case can potentially allow the previously enabled listener to be notified after successfully returning from this method.

The event mailbox service does not normally concern itself with the attributes of the `RemoteEvent`s that it receives. The one circumstance about which it must concern itself is when a target listener throws an `UnknownEventException` during an event delivery attempt. The event mailbox service must maintain a list, on a per-registration basis, of the particular combinations of event identifier and source reference (obtained from the offending `RemoteEvent` object) that produced the exception. The event mailbox must then propagate an `UnknownEventException` back to any event generator that attempts to deliver a `RemoteEvent` with an identifier-source combination held in a registration's unknown exception list. The service will also skip the future delivery of any stored events that have an identifier-source combination held in this list.

A registration's unknown exception list is cleared upon re-enabling delivery with any target listener. This list is persistent across service restarts or crashes, until the associated registration's lease terminates.

Note that the act of comparing event source objects for equality poses a security risk because source objects are potentially given references to other source objects that are currently using the mailbox. If security is a concern, then care should be taken to prevent independent event sources from obtaining information about each other.

Again, although implementation details are not specified in this document, service implementations need to carefully weigh the trade-offs of taking a particular security approach. For example, a low-security implementation could simply compare source objects using the `equals` method. This approach assumes well-behaved `equals` methods that pose no security risk. A more secure implementation might compare only source objects (using `equals`) that have the same codebase on the assumption that classes from the same codebase are trusted. Unfortunately, this approach will not work for services that evolve by changing their codebase (presumably to the location of the upgraded class files).

The event mailbox does not support multiple, concurrent notification targets per registration. As a result, the interface supports only a set/clear model rather than the more common add/remove model.

Event persistence guarantees are not specified in this document because no single policy can cover all the possible design trade-offs between reliability, efficiency, and performance. It is expected that operational parameters—controls for how the event mailbox deals with issues such as persistence guarantees, storage quotas, and low space behavior—will be exposed through an administration interface, which can vary across different event mailbox implementations.