Federated Management Architecture (FMA) Specification

Version 1.0

Revision 0.0
November 12, 1999

Table Of Contents

Table of Figures

About This Document

1 Status

This document is a draft for public review, as defined by the Java™ Community Process (JCP).

2 Annotations

Note – In this document you will notice several paragraphs appear in this style. These are areas where we specifically invite comment. Consider them as “notes to reviewers”.

Terms in ***bold Italics*** are particularly important and are defined in the glossary.

3 Changes in this Version

This version is a major revision.

4 Comments

Please direct comments to core-ri@thor.central.sun.com.

Introduction

5 Audience

The readers of this document are assumed to be technical and versed in object oriented design, the Unified Modeling Language (UML), Jini™ technology, and Java technology. The audience is assumed to be implementers of this specification or of components which are deployed on such an implementation. In the latter case, this specification is intended as a reference rather than a guide.

6 Document Goals

This specification defines the Federated Management Architecture (FMA) sufficiently for vendors providing implementations of the specification. As the scope of the platform includes the interactions between an implementation and deployed components, this specification also places constraints on the behavior of components in their contracts with the implementation. However, this specification is not intended as a guide for vendors writing or using management components. Design guidelines for management components and their use of supporting technologies such as Web Based Enterprise Management (WBEM) is the subject of other related documents.

7 Specification Boundaries

In leading the development of this specification, Sun has placed boundary conditions that must not be violated and must remain part of the JCP. In particular, architecture is to be based on Java technology and Jini technology. Java technology is used as the primary mechanism for achieving platform neutrality. Given platform neutrality, one may derive other forms of neutrality such as protocol neutrality. Language neutrality is not a goal for the initial specification, but can be approached later through other means. While this specification is Java technology centric, it is not Solaris operating environment centric. The primary validation platforms are NT and the Solaris operating environment.

Section 1: Analysis Model

This section presents an analysis model of management applications as assemblies of management services, management clients, and managed resources. The model illustrates how management services can use other management services, the interface between management services and clients, as well as the interface between management services and managed resources such as storage devices and applications. When management services are assembled in hierarchies, complex storage systems can be made to appear simple because users of the system interact only with the top-levels of the hierarchy at a high level of abstraction. In particular, it is desirable that a given storage system be managed at the same level of abstraction as the provided data. For example, when managing a database appliance, an administrator would ideally manage the performance, size, and other characteristics of tables rather than manipulating the disks and volumes on which the database runs.

The analysis model describes the form of a solution to the management problem. The solution model, described after the analysis model, specifies the infrastructure designed to support such solutions.

8 Three Tiered Architecture

The three tiered architecture has been validated in many application domains and has well known properties. As applied to management, the first tier is the management client tier, the second tier is the management services tier, and the third tier is the managed resource tier. Clients are hosted by Java Virtual Machines (JVMs), services by JVMs enabled as management servers, and resources by any appropriate host machine including a JVM.

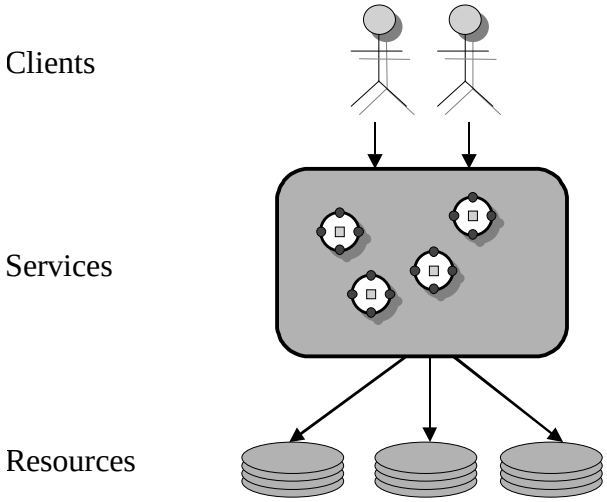

Figure 1. The Three Tiered Architecture of Management Applications.

The client communicates with management services, which ensures that the third tier, the resource tier, is manipulated in a controlled and consistent manner.

8.1 Client

The client locates and communicates with management services. Often the client is the user interface for an administrative user, but this is not always the case. Clients are

considered transitory. Objects associated exclusively with the client are only expected to live as long as the client process, even if the client terminates abnormally or becomes unreachable. This is also true for client objects that have been transferred to a management server. For the purposes of defining expected high availability, it is acceptable to restart the client in order to reestablish management capabilities in response to the failure of a management server.

Java clients can locate and communicate with management services directly. Outside of this specification, there can be bridges to connect non-Java clients. An example bridge would be a servlet that allows using a browser for management. The servlet would communicate with the browser using HTTP/HTML and with the services tier using Java Remote Method Invocation (RMI).

Other than specifying how clients communicate with management services, this specification will not define the architecture or design of management clients.

8.2 Services

Management logic is comprised of services hosted by management servers. Management services are classified in a number of ways including whether they are transient, persistent, static, or dynamic.

Management is divided into disjoint domains. Each management domain has a single management server, called the ***shared management server***, representing the domain as a whole. There may be more than one shared server for the purposes of redundancy, but the entire replication group is treated as a single logical server.

Appliances, such as encapsulated file servers, can also have embedded management servers to host services that are private to the appliance. This class of server is called the ***private management server***. The union of shared and private management servers within a single domain is called an ***intradomain federation***.

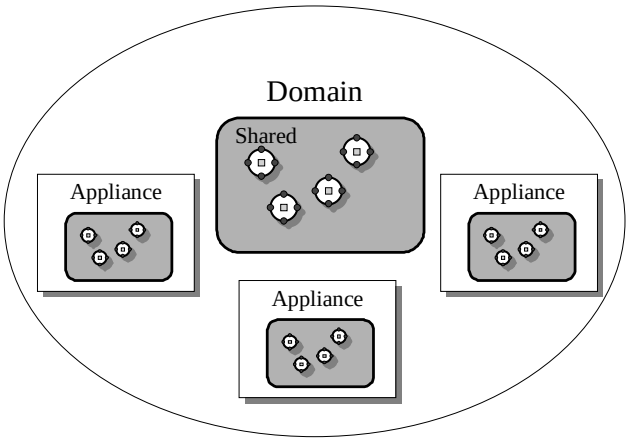

Figure 2. Intradomain Federation.

The union of the shared management servers of each domain forms the strictly hierarchical ***interdomain federation***.

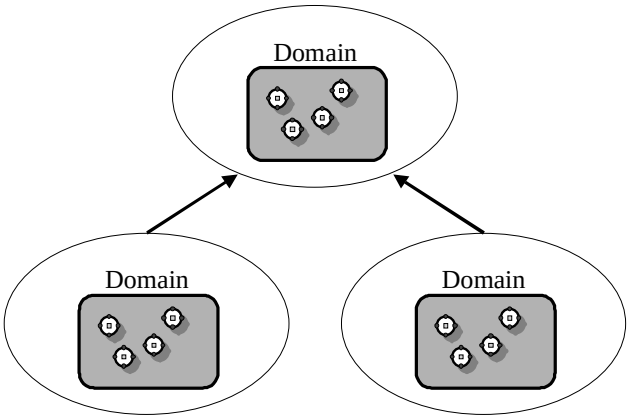

Figure 3. Interdomain Federation.

Shared management servers of different domains may communicate with one another. Private management servers may not communicate across domain boundaries.

8.3 Managed Resources

The resource tier includes a mix of heterogeneous managed resources such as devices, appliances, systems, and applications. Unlike business applications, most of the state that is interesting to management resides not in a database but in managed resources. A number of standards exist or are emerging, such as Web Based Enterprise Management (WBEM), for communicating with managed resources.

While management servers will provide built-in support for WBEM, the architecture is protocol neutral. If the managed resource is capable of hosting a JVM, it can choose to embed a private management server and be managed using inter-service communication, in which case the managed resource is called an ***appliance***. This technique has the advantage of propagating the features of management services to the appliance. Thus, management services can be dynamically installed, updated, and otherwise manipulated within an appliance.

9 High Level Requirements

9.1 Automate Management

Management to date has been dominated by monitoring. Moving from monitoring to controlling and, finally, to automated or policy based management, requires infrastructure support, such as control arbitration, not found in the current generation of management products.

9.2 Provide High Integrity Management

As managed systems become more automated and complex, it becomes essential for the platform to provide some guarantees about the integrity of the management activities. This requirement drives such features as security, transactions, and the control arbitration as the set of mechanisms that protect management integrity.

9.3 Provide a Simple Programming Model

Vendors providing management components will generally not be experts in distributed Java programming. The specification should be biased towards simplicity rather than completeness or performance to minimize the cost of creating services by vendors who are not Java technology centric. The simplicity can be achieved using a mix of development tools, class factoring, and any other applicable techniques.

9.4 Remote Management

Management shall be possible from remote locations, including outside firewalls and possibly over unsecured networks.

9.5 Centralized Management

It shall be possible to manage an entire management domain from a single location.

9.6 Provide Highly Available Management

The management services of highly available systems should themselves be highly available. Highly available means that one can proceed with management tasks following the loss of a management server. The continuation is not necessarily transparent, just possible.

9.7 The Management Infrastructure Should Not Be A Management Headache

The solution to the management problem should not itself be a management problem. This requirement drives a simple management solution compared to similar technologies such as application servers.

10 Aspect Requirements

The implementation of this specification must support the following aspects applied to management services.

10.1 Controller

An important objective of the specification is providing the infrastructure to support control arbitration. The primitive required for arbitration is called the controller aspect of the management services model and must support durable (long term) exclusive locking of resources.

10.2 Logical Thread

As the specification is intended to support active, autonomous management applications, it must be able to support concurrent and reentrant conditions with respect to threads. Management applications are made of distributed components, so the services model introduces the concept of a logical thread that spans processes. Thus, behavior with respect to threads can be specified with respect to *logical* threads instead of language threads.

10.3 Transaction

Most distributed object models provide some form of transaction support to aid in protecting the integrity of the resource layer. The specified transactions are inherited from the Jini programming model and focus on supporting large numbers of heterogeneous resources, rather than a single large resource (database). In many respects they may be thought of as a distributed form of try/catch rather than the more classic transaction model supported by transaction monitors and application servers.

11 Installation Requirements

11.1 Dynamic Installation

The specification must provide for the dynamic installation and updating of management services without requiring that management servers be restarted. Installation shall support both temporary installs as well as durable installs.

12 Registration Requirements

12.1 Management Server Lookup

Management servers shall registered with a well known lookup service where they may be located by clients and other management servers. Management servers shall be well good Jini technology citizens with respect to registration.

12.2 Service Lookup

Management services shall registered with a well known lookup service where they may be located by clients and other management services. Management services shall be well good Jini technology citizens with respect to registration.

Section 2: Dynamic Services

While the analysis model describes the problem domain, which in the case of infrastructure is a solution to a higher order problem, the solution model describes the form of a solution. The problem is providing the infrastructure needed to support three tiered management applications as described by the analysis model. The specified solution to this problem provides a component model based on Jini technology services.

The specification classifies management services as static or dynamic. Static services, called base services, include the transaction manager, logging, and other services considered always present in a management domain as part of the environment. These services are supplied as part of an implementation of this specification. As such, the deployment of base services and the hosting environment are implementation rather than specification issues. For example, the logging service could be implemented as an Enterprise Java Beans (EJB) or even a native implementation exposed through a Java facade.

Dynamic services are supplied independent of a management server implementation. Since a vendor boundary exists between dynamic services and the management server implementation on which they run, this boundary must be specified so that dynamic services may be portable between management server implementations. The dynamic services model specifies the involved contracts and compromises the majority of this specification.

The dynamic services model extends Java RMI to support a higher level (application level) of abstraction appropriate for management applications. The added abstractions include the following.

1) The propagation of contextual information including security and controller information.
2) Reference fault rebinding to allow management servers to be recovered on a different host than the one on which they were started.
3) Management aspects (security, transaction, controller)
4) Transactional persistence
5) Remote class method (procedural) invocations.
6) Remote object instantiation.

Management servers, called stations in the context of dynamic services, host management services that, in turn, communicate with other management services or managed resources. Resources may be accessed using Web Based Enterprise Management (WBEM), Simple Network Management Protocol (SNMP), or any other means appropriate to a particular situation. Stations are Jini technology services and registered with the lookup services serving the management domain to which the station belongs. One can consider stations as services that host dynamic services.

13 Extended RMI Semantics

The dynamic services model adds application layer semantics to complement RMI remote communication semantics for usage patterns common in the management domain.

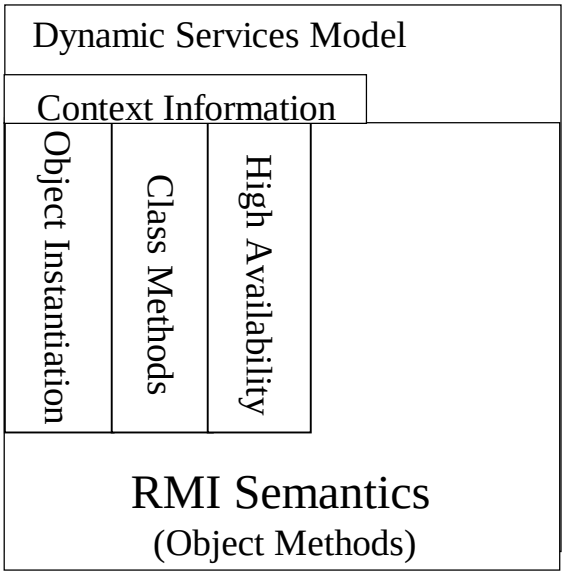

Figure 4. Architectural Layering of RMI Semantics with Dynamic Services Semantics.

13.1 Remote Object Instantiation

Stations support remote instantiation of objects without the need to create explicit remote factories by providing a reflective remote instantiation service that may be used directly as a low-level interface or indirectly through Proxies (described later).

13.2 Remote Class Method Invocation

Stations also support remote invocation of class methods by providing a reflective class method invocation service that may be used directly as a low-level interface or indirectly through Proxies (described later).

13.3 High Availability

To support highly available stations, the dynamic services model defines a reference faulting/rebinding scheme. A failed station may be restarted on another host and communications with the objects hosted by that station will fail over to the new location. This mechanism is distinct from the RMI activation reference faulting for the purposes of activation within the bounds of a single host.

13.4 Context Information

The management architecture described in this specification is an explicit three tiered architecture. There is a source of activity (client, resource, service, etc.), an arbitrarily deep chain of logic that is largely stateless, and finally, the managed resources themselves at the end of the logic chain. The resources must be guarded against inappropriate access. Some such access can be malicious and must be guarded against using a security mechanism. Other inappropriate access can include accessing the resource using multiple, concurrent threads or transactions.

The information needed to guard a resource is passed implicitly in context. Thus, the context information includes the following.

1) Security (unauthorized access).
2) Transactions (concurrent access under more than one transaction).
3) Logical Thread (concurrent access under more than one thread).
4) Controller (concurrent access by more than one controlling entity).

14 Programming Interfaces vs. Implementations

This specification defines the programming interfaces that station implementations must support. Separating programming interfaces from implementation is done using several mechanisms to handle abstractions of object methods, class methods, and constructors.

Object methods are abstracted using Java interfaces. The specification defines and a number of such interfaces in the javax.sxi package and sub-packages. Implementations provide concrete classes that implement these interfaces. Java interfaces, however, do not provide a way of abstracting class (static) methods or constructors, both of which require

an abstraction mechanism in order to cleanly separate the specification from the implementation.

Constructors are class operations much like class (static) methods. When a constructor must be abstracted for the specification, it is replaced with a class (static) factory method. This reduces the problem of interface/implementation separation to object methods and class (static) methods.

Static methods are abstracted using ***implementation forwarding***. For example, consider a class `A` with a class method `foo()`. The specification provides an *abstract* class A in a specification package (javax.sxi…) with the class method `foo()`. The implementation of the method fetches a reference to the *implementation* class `A` and invokes `foo()` on the implementation class. This class has the same unqualified name, `A`, but resides in an implementation package. A system property, "javax.sxi.implementation", provides the implementation package. Thus, all such implementation classes reside, for convenience, in the same package as defined by the system property "javax.sxi.implementation". If the implementation package is not provided, "com.sun.sxi.implementation" is used as the default. Note that this property is static and cannot be changed at runtime.

15 The Structure of Dynamic Services

Dynamic services all have a common structure and deployment. The structure is dominated by the requirement to be a good, network loadable Jini citizens.

15.1 Service Proxy

The service proxy is a Jini proxy that is registered by value with the lookup services serving a particular management domain. Remote operations invoked on the proxy are forwarded to the remote point objects - the remote entry point to the service. Generally, there is a single point object that implements the same interface as the service proxy, but this is not required.

15.2 Point Objects

Point objects are the entry points into a service. Other objects that comprise the interface of the service are exposed, directly or indirectly, by the point objects. Objects may also be exposed through remote instantiation and class method invocations. These operations do not require access to the service through the service proxy.

15.3 Public Interface

The public interface of a service is the set of all objects, classes, and interfaces that may be exposed to clients of the service. Not all of these entities may be statically determined. For example, consider a service defining a method that returns an object of interface I. The class of the actual object returned may be anything that implements I. This implementation is part of the public interface because a client of the service would need to load this class in order to communicate with the service. This kind of problem may be reduced by using final classes and JDK classes as the arguments, return values, and exceptions of remote operations when allowable by good design.

The public interface does not include remote objects: just the Proxies (or stub in the case of RMI) to the remote objects. Since remote objects reside in the JVM hosting the service, they do not need to be loaded by a client of the service and, therefore, or not considered part of the public interface. The client will, however, need to load the client side representation (proxy) of the remote object. Thus, Proxies and RMI stubs are considered part of the public interface. The service proxy itself belongs to the public interface.

15.4 Service Implementation

The parts of the service that are not the public interface are considered the service implementation.

15.4.1 Point Objects

Point objects, previously described, are part of the service implementation as a special kind of remote object.

15.4.2 Hidden Objects and Classes

Many, if not most, of the service implementation is composed of hidden objects and classes. Hidden objects and classes are not exposed in any way to the client of the service. Clients never communicate directly, or apparently directly, with hidden objects and classes as they do remote objects and classes.

15.4.3 Remote Objects and Classes

Remote objects and classes, which reside in the JVM hosting the service, are referred to remotely using Proxies, and are considered part of the implementation. Remote objects and classes are known collectively as remote referents.

15.5 Service Packaging

15.5.1 JAR Files

Services are packaged into two JARs for deployment: the implementation JAR and the interface JAR. The two JAR files are known collectively as a deployment group. The classes and resources needed to support the implementation and public interface shall be contained in the implementation JAR. Only the classes and resources needed to support the public interface shall be placed in the interface JAR. If the implementation JAR is named x.jar, then the interface JAR must be named x-dl.jar in accordance with Jini technology naming conventions. 'dl' is case insensitive. Each JAR shall be self sufficient in that it contains all of the classes and resources needed to load any of the contained classes with the exception that the following infrastructure classes may be omitted.

1) JDK classes
2) Jini classes
3) Java extensions

4) Classes defined in this specification

This is similar to applet packaging except that the result is a deployment group (two JARs) rather than a single JAR. The second JAR, the interface JAR, is a strict subset of the first.

15.5.2 Signing

The deployment JAR files shall be signed to enable security. Stations are encouraged not to grant any permission to anonymous code.

15.5.3 Versioning

The deployment JAR files are required to contain Java package version information in the manifest according to the Java Package Versioning specification.

16 Remote Referents

Remote referents are the targets of remote operations and include referent classes and referent objects. Referent objects may be stateful and are further classified as transient or persistent. Referent classes are stateless: only constant static fields are permitted.

16.1 Referent Classes

Stations, by providing generic factory and invocation services, permit class operations including class method invocation and instantiation. These operations obey extended RMI semantics as if the class was treated as a remote object.

16.2 Referent Objects

Referent objects are remote objects that support extended RMI semantics. The three aspects (logical thread, transaction, and controller) may be applied to referent objects. Referent objects are either transient or persistent.

16.3 Exclusion of RMI Remote Objects

The station security model depends on passing all remote access to the station through a well controlled gateway. The use of RMI remote objects from within a station would circumvent the security model and is therefore prohibited. Remote objects should instead be proxied and participate in the extended RMI semantics of this specification.

17 Proxies

Remote operations are performed on referents, class or object, by invoking local operations on a Proxy object or class. Proxy classes are created during development, preferably with a wizard type tool, and often packaged as part of both the implementation and interface JARs. The developer can choose to expose all public operations of the referent to the Proxy, or just a subset. Operations on a Proxy *class* are forwarded to the referent *class* while operations on a Proxy *object* are forwarded to the referent *object*.

A Proxy refers to a single referent. However, a single referent can have many Proxies just as a local object can have many references.

A client can obtain a Proxy object in one of three ways:

1) remote instantiation,
2) receiving the Proxy as a result of a remote operation, or
3) receiving a Proxy as an argument to a remote operation invoked on the client.

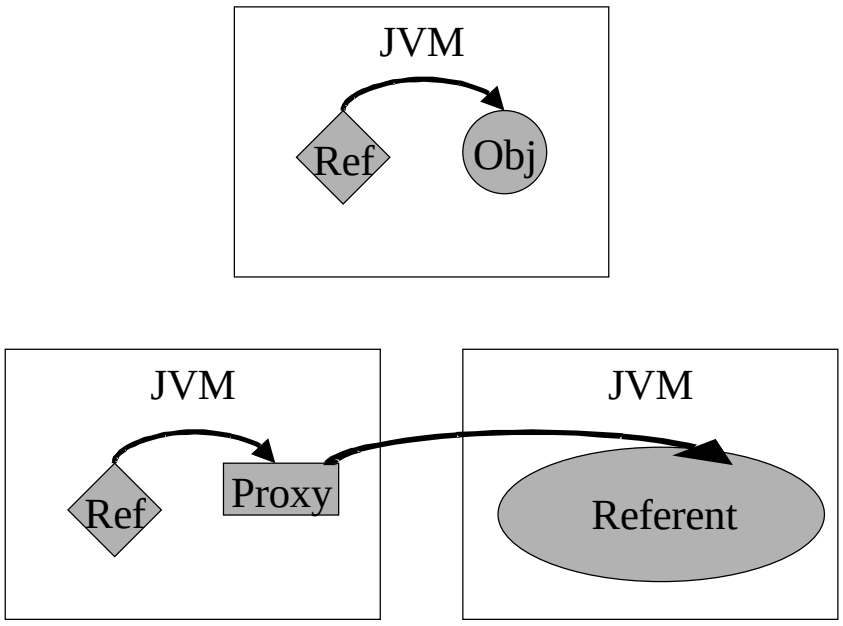

Figure 5. A Comparison of Local Java Programming (top) and Remote Programming using Proxies and referents.

As shown above, when a reference and object coexist in the same JVM, the reference points directly to the object: in effect the referent. In contrast, when the reference exists in a different JVM than the referent, the reference points to a Proxy, which communicates, through the station infrastructure, with the referent. When the Proxy and the referent implement the same interfaces, the client is largely unaware of whether the referent exists locally or remotely. That is not to say that referent always appears to be local, but rather that it always appears to be remote and may be local.

It is important to note that while local and remote operations appear similar, they have different behaviors, particularly with respect to failure modes and latency. The intent in making remote programming appear similar to local programming is to minimize the learning curve, not to hide the fact the remote operations must be treated differently than local operations.

Proxy classes are independent of station implementations and, therefore, can be used against any station implementation. The neutrality of the implementation is achieved by defining an interface between the Proxy and the ***station proxy*** for communicating with a station. The station proxy is a Jini proxy (the station is a Jini service), rather than a Proxy in the sense of Proxies and referents. In this specification, proxy is used in the generic sense of the Proxy Pattern and in the specific senses of referents and of Jini proxies. When a proxy is referred to with respect to referents, the word Proxy is capitalized to provide differentiation.

Proxies are durable in that they can be serialized for the purposes of persistence. When Proxies are not live, however, they do not participate in the distributed garbage collector. Thus, it is possible that persisting a Proxy allows a transient referent to be prematurely garbage collected.

18 Context

Certain contextual information can be associated with a thread of execution, both locally and across remote operations. With the exception of security, this context information is accessed on both the client and server side using methods on the `java.sxi.common.Context` class. The contextual information includes `LogicalThreadID`, `Transaction`, and `Controller`. The security context is accessed using the Subject class of the Java Authorization and Authentication Service (JAAS) and is described fully the JAAS specification.

18.1 Logical Thread Identifiers

When the Proxy and its referent lie in different JVMs, they execute in different language threads. This can cause reentrancy problems when the logical thread of execution spans JVMs and thread concurrency control is based on language threads. To permit reentry and support other thread related constructs, the concept of a logical thread is introduced. During a remote operation that spans JVMs, both the caller and the called threads belong to the same logical thread. This allows concurrency control to be based on logical threads rather than language threads, if so desired.

Each logical thread is uniquely identified, with respect to the universe, by its `LogicalThreadID`. A logical thread is assigned to a language thread when 1) the language thread first invokes a remote operation or 2) when servicing a remote operation. Thus, the infrastructure is the only entity allowed to set the `LogicalThreadID` of a language thread.

One can query the `LogicalThreadID` of the current thread using the `Context` class. The returned ID is opaque, but can be compared for equality using `equals()`.

18.2 Transactions

Transactions are issued by a Jini transaction service: one of the base management services. The semantics and transaction interfaces are more fully described in the *Jini Transaction Specification*. One can query the transaction associated with the current thread using the `javax.sxi.common.Context` class.

18.3 Controller

While logical threads and transactions are considered short-lived, bounded by the lives of one or more processes, controllers are long lived. Controllers are assigned to each controller object or client. Clients must obtain a controller directly from the controller service. Stations hosting controller objects obtain controllers for these objects on their behalf.

Clients obtain a controller from the controller service and retain the context for the life of the client by maintaining the associated lease. Clients must cancel the lease at the end of their lives. The controller service will cancel locks held by a controller in response to lease expiration, presumably indicating that the client has unexpectedly failed or otherwise become irrelevant. Controllers can exclusively lock resources for the life of the controller. The locking mechanism is covered further in the controller aspect chapter.

```
import javax.sxi.services.controller.ControllerService;
import javax.sxi.services.controller.ClientController;
import javax.sxi.services.ServiceFinder;

ControllerService controller =
   ServiceFinder.getControllerService();

ClientController aController =
   controller.newClientController( 5*60*1000 );
```

```
package javax.sxi.common;

import java.io.Serializable;
import javax.sxi.services.controller.*;
import net.jini.core.transaction.server.*;

/**Contextual information associated with a thread of
 *  execution. This information is propagated implicitly
 *  during a remote operation.
 */
public class Context implements Serializable
{
   /** The empty context */
   static public final Context EMPTY_CONTEXT;

   /** Logical thread of this context.*/
   public LogicalThreadID    getLogicalThread();

   /** Transaction associated this context. */
   public ServerTransaction  getTransaction();

   /** Controller of this context. */
   public Controller            getController();
```

```
    /**Construct a new Context object.
     * @param Controller
     * controller to associate with current thread
     * @param transaction
     *     transaction to associate with current thread
     */
    public    Context(
          Controller controller,
          ServerTransaction transaction );

    /**Return the Context associated with the current
     *  thread of execution. If the current thread has no
     *  associated Context, the default Context is
     *  returned.
     * @return Context context associated with the current
     *     thread.
     */
    public static Context    getContext();

    /**Associate this context with the calling thread. The
     *  Context previously associated will be returned
     *  (default Context, if no Context was associated). A
     *  logical thread ID will be set for the Context when
     *  this method is invoked (unless a logical thread ID
     *  is already established for the thread).
     * @return Context context previously associated
     *     with current thread (may be the default Context)
     */
    public static Context     setContext(
                                    Context context
                                );

    /**Set the default Context. This object will be
     * returned by getContext() for any calling
     *  thread that has no associated Context.
     * @return Context context previously set as default
     *     (EMPTY_CONTEXT if none)
     */
    public static Context     setDefaultContext(
                                    Context context
                                );

    /**If called within a referent controller object while
     *  servicing a remote operation, this method cancels
     *  any locks held by the associated controller.
     *  @throws RemoteException If not able to contact the
     *     controller service.
     */
    public static void        cancelLocks()
       throws RemoteException;
}
```

19 The Station Interface

Stations are Jini technology services for hosting dynamic management services. The primary responsibility of a station is providing means of introducing services into the

station: instantiation and installation. A secondary responsibility is providing a mechanism for invoking methods on referents.

19.1 Method Signatures

Method signatures, as String objects, are used to specify methods and constructors. Signatures consist of the method name followed by the method descriptor, as specified by the Java Virtual Machine Specification. For example, the signature of the method '`void foo(Integer i, int j)`' is `foo(Ljava/lang/Integer;I)V`. Constructor signatures are method signatures with the special name of `<init>`.

19.2 Station Registration

Stations must register themselves with the all Jini lookup services servicing the management domain to which the station belongs. The group name for a management domain is the management domain name. The registered service item must contain a proxy that implements the Station interface and a single item of type `javax.sxi.common.StationAddress`. Stations shall monitor the existence of lookup services and register with any new relevant lookup services that join the network. In short, stations shall be good Jini citizens.

19.3 Station Lookup

The station proxy is a Jini proxy and is looked up using a `javax.sxi.common.StationAddress`, a specialization of the `net.jini.lookup.entry.ServiceInfo` class. The `net.jini.lookup.entry.ServiceInfo` class provides the following fields.

1) Manufacturer
2) Model
3) Name
4) Serial number
5) Vendor
6) Version

`javax.sxi.common.StationAddress` adds an additional field, role. All fields are public, so it is possible to base lookups on any of the fields. In accordance with the Jini specification, empty fields are treated as wild cards for the purposes of lookup.

19.4 The Station Interface

All station proxies must implement the Station interface.

```
package javax.sxi.common;

import net.jini.core.lookup.ServiceID;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.UnknownObjectException;

public interface Station extends Remote
{
   /**Invoke a static method.
    *  Requires security permission as described in
    *  javax.sxi.security.AccessPermission.
    */
    Object   invokeStaticMethod(
                  String className,
                  String methodSignature,
                  Object[] arguments,
                  Context context
               )
      throws RemoteException,
         InvocationTargetException,
         IllegalAccessException,
         IllegalArgumentException;

   /**Invoke a constructor. The result may be an object
    *  passed by value or BindInformation depending on the
    *  target of the invocation. If the target has an
    *  available Proxy class, BindInformation is returned.
    *  Otherwise the newly constructed object will be
    *  returned by value. In the latter case, if the
    *  object is not serializable a RemoteException will
    *  be thrown.
    *  Requires security permission as described in
    *  javax.sxi.security.AccessPermission.
    */
    Object   invokeConstructor(
                  String className,
                  String constructorSignature,
                  Object[] arguments,
                  Context context
               )
      throws RemoteException,
         InvocationTargetException,
         IllegalAccessException,
         IllegalArgumentException,
         InstantiationException;

   /**Rebind to a referent. Proxies, after relocating
    *  a station, can rebind to the referent.
    *  @param cookie As returned during remote
    *     constructor invocation.
    */
   BindInformation rebind( Object cookie )
      throws RemoteException,
         UnknownObjectException;
```

```
    /**Low cost roundtrip communication check.
     * @throws RemoteException if ping fails.
     */
    void         ping()
       throws RemoteException;

    static public final class BindInformation
    {
       /** Information needed to relocate this
        *  particular station.
        */
       public final ServiceID stationID;

       /** Information needed to relocate referent. */
       public final Object    cookie;

       /** Invocation path to referent */
       public final Acceptor  referent;
    }
}
```

20 Deployment

Stations that are shared management servers must implement the `javax.sxi.common.DeploymentStation` interface. Stations that are private management servers may optionally implement the `javax.sxi.common.DeploymentStation` interface if they support deployment.

20.1 Deployment Definition

Deployment, as used in this specification, is the process of giving classes and resources, packaged as JARs, to a station. Deployment is generally part of an installation process. A single deployment is one deployment operation. If the same deployment group is deployed multiple times, even to the same station, each is considered a distinct deployment.

20.2 Class Loaders and Deployment

Deployments and class loaders have a one-to-one relationship. To remain compatible with RMI class loading, this mapping implies that each deployment is given a unique code base. This code base shall consist of two ordered URLs. The first is a URL that may be used to load the public interface JAR and the second is a URL that may be used to load the implementation JAR. Note that the URLs are generated by the station to ensure uniqueness of the code base and may not have any resemblance, in name, to the JARs of the deployment group presented for deployment. Only HTTP is allowed as a protocol for code base URLs.

A class that is loaded from a deployment group must be annotated with the code base of the deployment according to RMI class loader semantics. This requirement helps ensure that when objects of that class are passed outside of the originating JVM, network class loading will work as outlined in the RMI specification.

```
package javax.sxi.common;

import java.rmi.RemoteException;
import java.net.URL;
import net.jini.core.lease.Lease;

/**The proxies for stations that support deployment
 * must implement the DeploymentStation interface.
 */
public interface DeploymentStation extends Station
{
   /**Deploy a deployment group. If the lease duration
    * is specified as Lease.FOREVER, the lease does not
    * need to be maintained; however, the installing
    * entity must guarantee that the deployment group
    * will be recalled when appropriate. Other lease
    * values will result in a lease that must be
    * maintained. A cancelled or expired lease releases
    * the deployment group for garbage collection.
    * <P>This version of install should be used when
    * possible as it permits the JARs to be pulled rather
    * than pushed. Note that the provide URLs may not
    * have any relationship to the code base resulting
    * from the installation.
    * Requires security permission as described in
    * javax.sxi.security.AccessPermission.
    * @param implementationJar JAR containing the
    *     implementation resources.
    * @param interfaceJar JAR containing the
    *     public interface resources.
    * @param leaseDuration Requested lease duration
    *     for the deployment. May be Lease.NO_LEASE.
    * @RemoteException Error communicating with the
    *     station or an unexpected exception.
    * @throws DeploymentException Unable
    *     to deploy for reasons nested within the
    *     DeploymentException.
    * @throws IllegalArgumentException null argument,
    *     invalid URL, or invalid JAR file.
    */
   Lease            deploy(
                       URL implementationJar,
                       URL interfaceJar,
                       long leaseDuration,
                       Context context
                    )
      throws RemoteException,
         InstallationException;
```

```
/**Install a deployment group. If the lease duration
 * is specified as Lease.FOREVER, the lease does not
 * need to be maintained; however, the installing
 * entity must guarantee that the deployment group
 * will be uninstalled when appropriate. Other lease
 * values will result in a lease that must be
 * maintained. A cancelled or expired lease releases
 * the deployment group for garbage collection.
 * <P>This version of install is used when installing
 * from a location that will not accept http
 * connections. Thus, the JAR files must be pushed
 * during the call rather than pulled. The URL form of
 * install should be used when ever possible.
 * Requires security permission as described in
 * javax.sxi.security.AccessPermission.
 * @param implementationJar JAR containing the
 *     implementation resources.
 * @param interfaceJar JAR containing the
 *     public interface resources.
 * @param leaseDuration Requested lease duration
 *     for the deployment. May be Lease.NO_LEASE.
 * @RemoteException Error communicating with the
 *     station or an unexpected exception.
 * @throws DeploymentException Unable
 *     to deploy for reasons nested within the
 *     DeploymentException.
 * @throws IllegalArgumentException null argument,
 *     or invalid JAR file.
 */
Lease            deploy(
                    byte[] implementationJar,
                    byte[] interfaceJar,
                    long leaseDuration,
                    Context context
                 )
   throws RemoteException,
      InstallationException;
```

```
    /**Remove the deployment group identified by the given
     *  code base. There is no assurance that the code base
     *  being recalled is not in use.
     *  Requires security permission as described in
     *  javax.sxi.security.AccessPermission.
     *  @param codeBase Code base to recall. null results
     *     in an IllegalArgumentException.
     *  @RemoteException Error communicating with the
     *     station or an unexpected exception.
     *  @throws UnknownCodeBaseException The code base
     *     is not a known code base.
     *  @throws DeploymentException Unable
     *     to recall for reasons nested within the
     *     DeploymentException.
     *  @throws IllegalArgumentException null argument,
     *     invalid URL, or invalid JAR file.
     */
    void             recall(
                        String codeBase,
                        Context context
                     )
       throws RemoteException,
          UnknownCodeBaseException,
          DeploymentException;

    /**List all of the installed deployment groups.
     *  Returns an empty list if no deployment groups
     *  have been installed.
     *  Requires security permission as described in
     *  javax.sxi.security.AccessPermission.
     *  @RemoteException Error communicating with the
     *     station or an unexpected exception.
     *  @return A list of inventory records. The list
     *     is empty if the inventory is empty.
     */
    Deployment[] getInventory( Context context )
       throws RemoteException;
```

```
    /**Get the code base for the latest version of the
     *  given package that is compatible with the supplied
     *  version. Compatibility between versions is defined
     *  in java.lang.Package.
     *  Requires security permission as described in
     *  javax.sxi.security.AccessPermission.
     *  @param packageName The fully qualified package
     *     name such as "com.sun.x.y".
     *  @param version Requested version with which the
     *     the returned code base should be compatible.
     *  @RemoteException Error communicating with the
     *     station or an unexpected exception.
     *  @throws UnknownPackageException The package
     *     is not known.
     *  @throws IllegalArgumentException null argument.
     */
    String       getCodeBase(
                        String packageName,
                        String version,
                        Context context
                    )
       throws RemoteException
          UnknownPackageException;

    /**Get the code base for the latest version of the
     *  given package.
     *  Requires security permission as described in
     *  javax.sxi.security.AccessPermission.
     *  @throws IllegalArgumentException null argument.
     */
    String       getCodeBase(
                        String packageName,
                        Context context
                    )
       throws RemoteException
          UnknownPackageException;

    /**Each deployment has a record of type Deployment
     *  in the inventory.
     *  @see getInventory()
     */
    public static final class Deployment
    {
       /** Fully qualified package name. */
       public final String packageName;

       /** Version as defined by java.lang.Package. */
       public final String version;

       /** Code base as a space delimited ordered list
        *   of URLs.
        */
       public final String codeBase;
    }
}
```

The Dynamic Services Model

21 Specifying a Type of Referent Object

Each referent object must declare its type. The declaration is done by initializing the constant field `ReferentType` with one of three constants.

```
package javax.sxi.server;

public interface ReferentType
{
   String TRANSIENT           = “Transient”;
   String PERSISTENT             = “Persistent”;
   String PERSISTENT_LOGIC    = “PersistentLogic”;
}
```

```
public class MyRemote implements ReferentType
{
   public static final String ReferentType = PERSISTENT;
}
```

In this example, the `MyRemote` is declaring itself as persistent. If the station does not support the specified type of referent, the referent cannot be used in the station. Simple stations are only required to support transient referents. Shares management servers are required to support all kinds of referents.

Persistent logic indicates a stateless persistent referent object. In comparison to stateful persistent objects, the station can make certain optimizations if it knows the referent to be stateless. This optimization is described in a later chapter.

22 Acceptors

The view of "`proxy->referent`", though useful for initial explanations, is not sufficient to fully specify the remote communications between a Proxy and its referent object. In particular, because Proxies and stations are potentially supplied by different vendors, the interface between Proxies and station must be specified. A Proxy does not see a referent object directly, but rather an interface representing the referent. This interface is the ***acceptor*** for referent object, a one-to-one mapping.

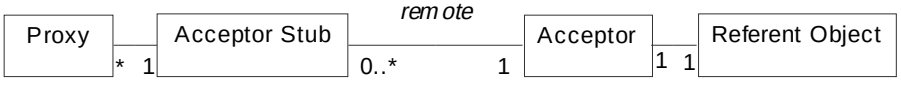

Figure 6. **Proxies, Acceptors and Referent Objects.**

```
package javax.sxi.server;

import java.rmi.RemoteException;
import javax.sxi.common.Context;

public interface Acceptor extends Remote
{
   /**Invoke a method on a referent object.
    */
    ObjectinvokeMethod(
                String methodSignature,
                Object[] arguments,
                Context context
             )
       throws RemoteException,
           InvocationTargetException,
           IllegalAccessException,
           IllegalArgumentException;
}
```

The acceptor presents RMI semantics to the Proxy. Context information is made explicit by the Proxy. In implementation, the proxy has a reference to a RMI stub which refers to the remote acceptor residing in the same JVM as the referent object.

23 Proxy Binding

A Proxy must acquire an appropriate acceptor at such time as the Proxy acquires an associated referent object. This happens in two scenarios: Proxy instantiation and Proxy wrapping, described later. The process of Proxy to acceptor association is called ***Proxy binding***. The acceptor bound to a Proxy possibly becomes invalid when a station is restarted on a different machine, such as when reacting to a failed host in a high availability scenario.

23.1 Proxy Binding During Proxy Instantiation

The Proxy constructor obtains a station proxy from a lookup service using the station address passed to the Proxy constructor, by convention as the last argument to the constructor. The Proxy then invokes `invokeConstructor()` on the station proxy, which returns an acceptor for the newly constructed referent object.

23.2 Proxy Binding During Proxy Wrapping

In the second case, Proxy wrapping, a Proxy is instantiated with a single argument, the referent object itself, using a special constructor called the ***wrapping constructor***. The Proxy is instantiated in the station containing the referent object; thus, Proxy wrapping is not a remote operation. In the wrapping constructor, the Proxy invokes the `javax.sxi.server.LocalStation.export()` method to retrieve an acceptor for the referent object to be wrapped. The newly instantiated Proxy object may then be passed remotely as a remote reference to the referent object.

```
package javax.sxi.server;

import javax.sxi.common.Station;
import javax.sxi.common.Station.BindInformation;

public final class LocalStation
{
    /**Return the local station.
     */
    static public Station  getStation();

    /**Provide an acceptor for the given referent object.
     *  Should only be called by Proxies and then only from
     *  a wrapping constructor.
     */
    static public BindInformation export( Object object );
}
```

24 Proxy Rebinding

The acceptor bound to a Proxy possibly becomes invalid when a station is restarted on a different machine, such as when reacting to a failed host in a high availability scenario. The Proxy must be able to ***rebind*** to the acceptor in such cases. The information required to rebind includes the service ID of the station, to uniquely identify it among other stations, and a cookie, issued by the station as part of the binding information, to uniquely identify the referent object within the station. All of this information is provided by the station during initial binding and must be retained by the proxy.

When a Proxy is unable to communicate with a referent object and the Proxy determines that this is likely because the hosting station is no longer reachable (`javax.sxi.common.Station.ping()`), then the Proxy should initiate rebinding. The rebinding involves relocating the station using the service ID and, if successful, requesting a fresh acceptor for the referent object using the `javax.sxi.common.Station.rebind()` operation.

25 Proxy to Referent Overviews

The following sequence diagrams are summaries of end-to-end remote communication. They elide exceptional and minor flows for the sake of clarity.

25.1 Referent Object Method Invocation

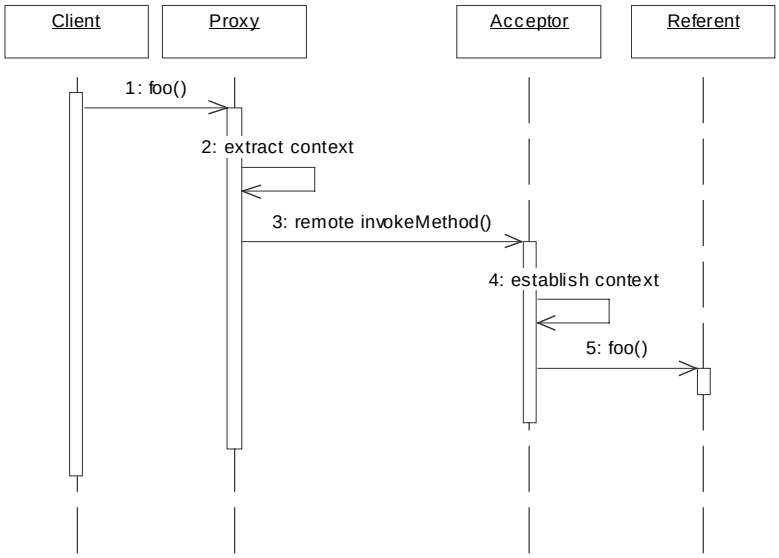

Figure 7. Referent Object Method Invocation.

1) The client invokes the object operation `foo()` on the Proxy. The Proxy will already have been bound to an acceptor when the Proxy was instantiated.
2) The Proxy extracts context (transaction, etc.) information to be passed explicitly to the acceptor.
3) The Proxy forwards the invocation request to the acceptor. This is a remote operation.
4) The acceptor uses the explicitly passed context information to establish a thread local context.
5) Finally, the acceptor invokes `foo()` on the referent object.

25.2 Referent Class Method Invocation

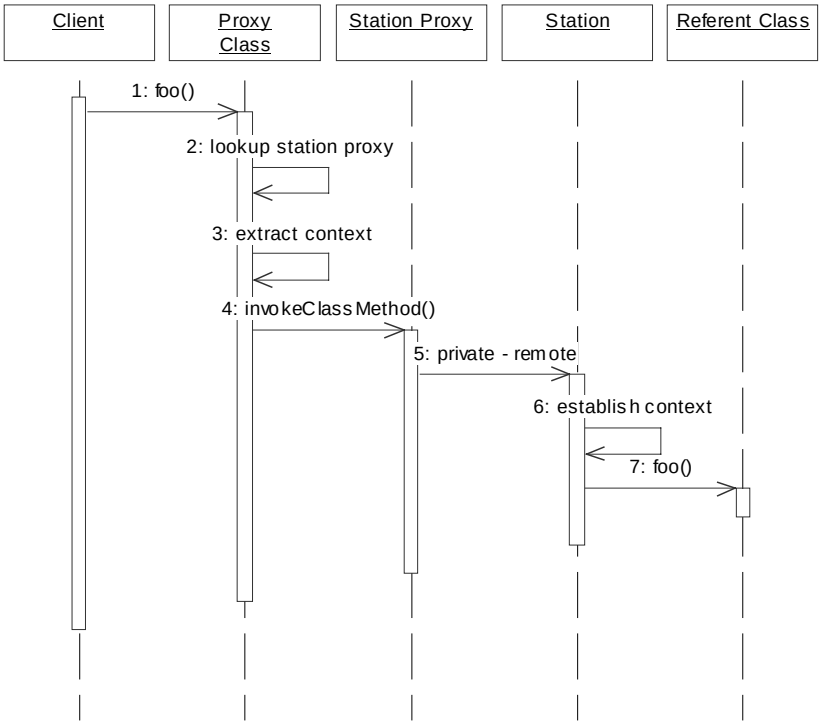

Figure 8. Referent Class Method Invocation.

1) The client invokes the class operation `foo()`, a static method, on the Proxy class.
2) The Proxy class will need to lookup an appropriate station proxy by querying a lookup service with the station address supplied as the last argument to the static method call.
3) The Proxy extracts context (transaction, etc.) information to be passed explicitly to the station proxy.
4) The Proxy class passes the operation request to the station proxy.
5) The station proxy forwards the request to the station.
6) The station uses the explicitly passed context information to establish a context.
7) Finally, the station invokes `foo()` on the referent class.

25.3 Referent Object Instantiation

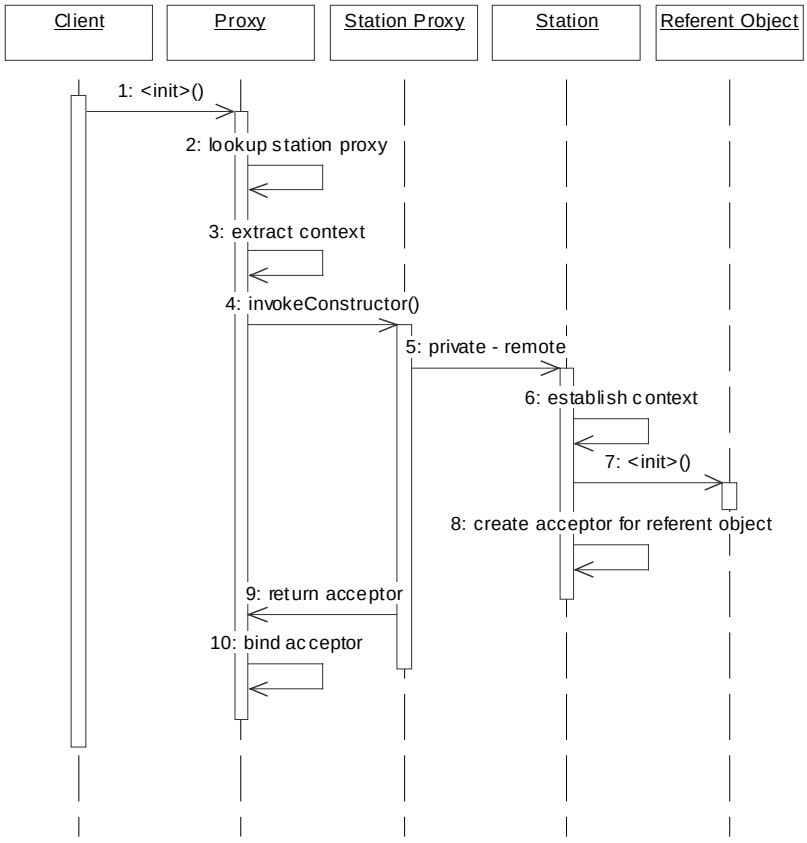

Figure 9. Referent Object Instantiation.

1) The client instantiates a Proxy using any one of the available constructors. The last argument of the constructor is the station address of the station that is to host the referent object.
2) The Proxy class will need to lookup an appropriate station proxy by querying a lookup service with the station address supplied as the last argument to the constructor.
3) The Proxy extracts context (transaction,etc.) information to be passed explicitly to the station proxy.
4) The Proxy class passes the operation request to the station proxy.
5) The station proxy forwards the request to the station.
6) The station uses the explicitly passed context information to establish a context.

7) The station locally instantiates the referent object.
8) The station must now create an acceptor for the new referent object.
9) The newly creating acceptor is eventually returned to the Proxy as an acceptor embedded in a binding information object.
10) The Proxy then binds the returned acceptor. All remote method invocations on the Proxy will now be forwarded through the acceptor to the referent object.

25.4 Wrapping a Referent Object with a Proxy

To pass a referent object by reference during a remote operation, the referent must be wrapped with a Proxy. Each Proxy class provides a wrapper constructor for this purpose. Note that the wrapping is local to the referent object being wrapped: no remote calls are involved. The wrapping sequence is similar to object instantiation except that the referent already exists and the Proxy communicates with the local station rather than a remote station.

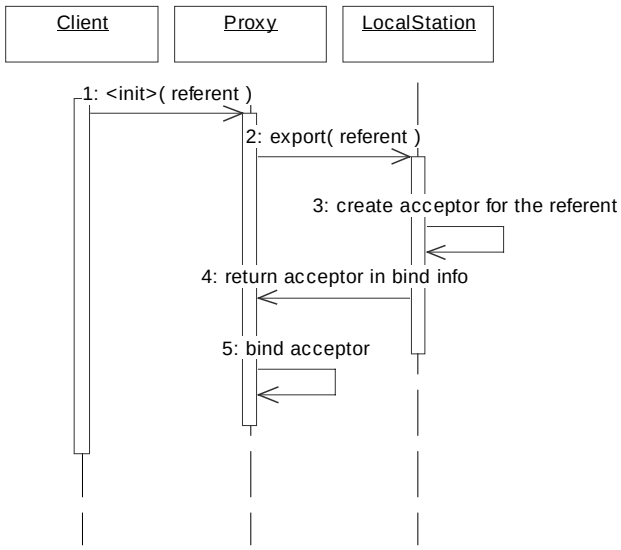

Figure 10. Wrapping a Referent Object with a Proxy.

1) The client invokes the Proxy wrapper constructor, passing the referent object as the only argument. In this sense, a client is simply the entity invoking the wrapping constructor of the Proxy.
2) The Proxy, from within the wrapping constructor, requests an acceptor from the local station by doing an export. The export provides a binding information object which contains the acceptor.

3) The station creates an acceptor for the referent object.

4) The station returns a binding information object, which contains and acceptor, to the Proxy. The acceptor should already be substituted with an acceptor stub so that RMI semantics will be followed when invoking methods on the Proxy.

5) The Proxy then binds the returned acceptor. All remote method invocations on the Proxy will now be forwarded through the acceptor to the referent object.

25.5 Proxy Rebinding

In highly available installations, a station may move to a failover host if the original host fails. This move can cause all acceptor stubs issued by the original incarnation of the station to become invalid. Proxies, upon failure of the original acceptors, proceed through a rebinding process to retrieve fresh acceptor stubs from the station at its new location. The following sequence diagram shows the rebinding in the context of a failed object method invocation.

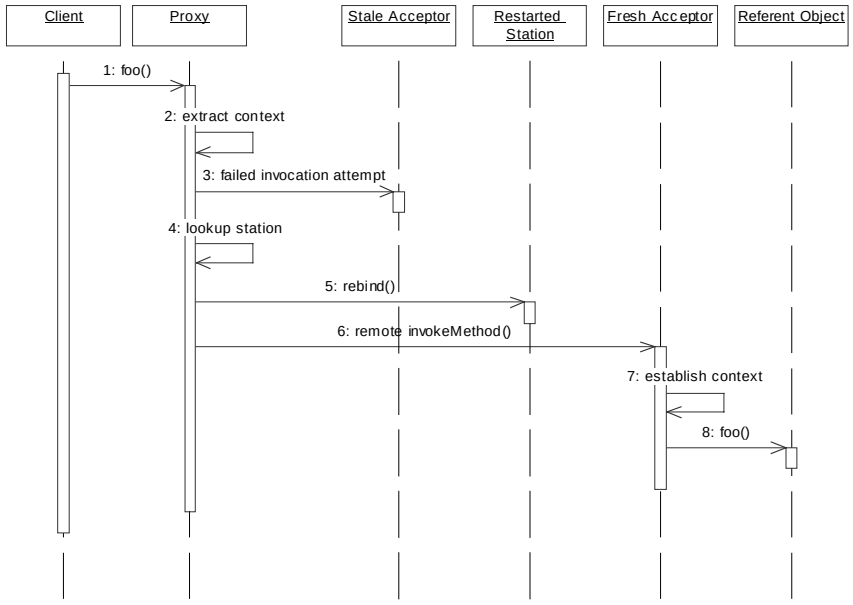

Figure 11. Proxy Rebinding.

1) The client invokes the object operation `foo()` on the Proxy.

2) The Proxy extracts context (transaction, etc.) information to be passed explicitly to the acceptor.

3) The Proxy attempts a remote call to the stale acceptor, no longer in existence, resulting in an exception. If the exception indicates that the station has become unreachable, the proxy shall continue with the rebinding, otherwise the exception shall be thrown to the client.
4) The Proxy looks up the restarted station using the original service ID of the station, retained during the original binding process.
5) The Proxy requests a fresh acceptor stub from the station. From this point on the invocation proceeds as with the normal object method invocation.
6) The Proxy forwards the invocation request to the acceptor.
7) The acceptor uses the explicitly passed context information to establish a context.
8) Finally, the acceptor invokes `foo()` on the referent object.

26 Adjunct Modifiers

Java defines a number of well-known modifiers, such as *static* and *public*, that may be applied to classes and methods. The dynamic services model extends this set with additional modifiers, ***adjunct modifiers,*** which have specific meanings to the station infrastructure. Because the Java language is not to be extended with application class features, adjunct modifiers are expressed as bit fields (`int`) in various well-known constant fields. These fields shall be considered immutable; however, they are not declared as final to allow for the possibility that they originate from sources other than the developer, such as a deployment policy defined by an administrator.

26.1 Class Modifiers

Modifiers are attached to a class using the private static field `classModifiers`. With the exception of the controller modifier, the modifiers do not apply to the class itself, but rather establish a set of default modifiers that apply to the class (static) methods of the class.

```
public class AClass
{
   ...
   private static int classModifiers = ...;
   ...
}
```

26.2 Object Modifiers

Modifiers are attached to an object using the private field `objectModifiers`. The modifiers establish a set of default modifiers that apply to the methods of objects of the given class.

```
public class AClass
{
    ...
    private int objectModifiers = ...;
    ...
}
```

26.3 Method Modifiers

Method modifiers attach modifiers to a class method, object method, or constructor. The method or constructor is specified as a signature, as defined in the previous Method Signature section.

```
package javax.sxi.server;

public final class MethodModifiers
{
    /**Construct a method modifier. The signature is as
     * defined by the object model, and, therefore,
     * includes constructors.
     * @param signature Signature of the method or
     *     constructor to which this modifier set applies.
     * @param modifiers Modifier bit field for this
     *     method or constructor.
     */
    public    MethodModifiers(
                  String signature, int modifiers
              );

    /**Modifier bit field for this method or constructor.
     */
    int       getModifiers();

    /**Signature of the method or constructor to which
     * this modifier set applies.
     */
    String    getSignature();
}
```

Modifiers are attached to methods (class or object) using the static table `methodModifiers`. The table has an entry for each method or constructor for which modifiers are specified. Methods without modifiers need not be present in the table.

```
public class AClass
{
    ...
    private static MethodModifiers[] methodModifiers =
    {
        new MethodModifier( ... ),
        ...
    };
    ...
}
```

26.4 Modifier Precedence

With the exception of the controller modifier, adjunct modifiers only have meaning at the method or constructor level. Modifiers attached to classes and objects simply establish default modifiers for the class or object methods. Thus, class and object modifiers can be overridden by modifiers attached to individual methods. Adjunct modifiers are overridden in *sets* of independent modifier categories. Effectively each set is an enumeration of exclusive modifiers. That is to say, that only one modifier of a given set can be applied to the class or method.

26.5 Accessing Modifiers

Adjunct modifiers shall never be accessed directly but rather through the `javax.sxi.server.AdjunctModifiers` class. The returned modifiers are the explicitly declared modifiers and do not apply the modifier precedence rules. Thus, if a class has modifiers but its methods do not, `getMethodModifiers()` will return 0 rather than the class modifiers.

26.6 Permissible Modifiers

The following modifiers are permissible. The modifiers are categorized as aspect modifiers or security modifiers. The semantics of each modifier set will be defined in a subsequent section of this specification.

```
package javax.sxi.server;

public abstract class Modifiers
{
    //
    // Controller modifier set
    //

    /**For classes only: declares object of the class
     *  to be controllers.
     */
    public static final int IS_CONTROLLER   = 0x0200;
```

```
    public static boolean  isController( int modifier );

    //
    // Security modifiers.
    //     Sensitivity Set
    //

    /**Communications need not be protected against
     * undetected tampering or third party viewing.
     */
    public static final int PUBLIC       = 0x0000;

    /** Communications need not be protected against
     *  third party viewing, but should be protected
     *  against undetected tampering.
     */
    public static final int SENSITIVE    = 0x0001;

    /**Communications should be protected against
     * third party viewing and undetected tampering.
     */
    public static final int CONFIDENTIAL = 0x0002;

    public static boolean  isPublic( int modifier );

    public static boolean  isSensitive( int modifier );

    public static boolean  isConfidential( int modifier );

    //
    // Security modifiers.
    //     Delegation set
    //

    /**Indicates that this methods uses client delegation.
     * In general, this means that the method requests the
     * client Subject and then intends to authenticate
     * itself with the client Subject.
     */
    public static final int USES_DELEGATION  = 0x0010;

    public static boolean  usesDelegation(
                                int modifier
                           );

    //
    // Component modifiers.
    //     Transaction set
    //

    /**Indicates that the method is synchronized with
     * respect to transactions.
     */
    public static final int SYNCHRONIZED_TRANSACTION
                                        = 0x0040;

    public static boolean  isSynchronizedTransaction(
                                int modifier
                           );
```

```
    //
    // Component modifiers.
    //     Logical thread set
    //

    /**Indicates that the method is synchronized with
     * respect to logical threads.
     */
    public static final int SYNCHRONIZED_LOGICAL_THREAD
                                        = 0x0080;

    public static boolean  isSynchronizedLogicalThread(
                                int modifier
                            );

    //
    // Component modifiers.
    //     Controller set
    //

    /**Indicates that the method is synchronized with
     * respect to controllers.
     */
    public static final int SYNCHRONIZED_CONTROLLER
                                        = 0x0100;

    public static boolean  isSynchronizedController(
                                int modifier
                            );

    public static int   getClassModifiers( Class clazz );

    public static int   getObjectModifiers( Object object );

    public static int   getMethodModifiers( Method method );

    public static int   getConstructorModifiers(
                            Constructor ctor
                        );
}
```

27 Proxy Class Details

The `Proxy` interface is the public interface that must be implemented by all Proxy classes.

27.1 Proxy interface

The `Proxy` interface includes a method returning the class name of the referent object's class.

```
package javax.sxi.client;

/**
 * Interface for Proxies.
 */
public interface Proxy extends Serializable, Cloneable
{
    /**Returns the name of the referent object
     * class.
     */
    String    getReferentObjectClassName();

    /**Indicates if the Proxy refers to a valid referent
     *  object. If isValid() returns false, the referent
     *  object may still exist, but has become, at least
     *  momentarily and perhaps permanently, unreachable.
     * @return Returns true if the Proxy refers to a valid
     *     referent object.
     */
    boolean   isValid();
}
```

27.2 Remotely Exposed Methods and Constructors

Implementations of the `Proxy` interface contain selected methods, constructors, and interfaces that are to be exposed through the Proxy. Signatures for exposed methods are identical to those of the referent with the following exceptions.

1) Class (static) methods have an additional last argument of type `javax.sxi.common.StationAddress`.
2) Constructors have an additional last argument of type `javax.sxi.common.StationAddress`.
3) All remote methods and constructors throw `java.rmi.RemoteException`.

27.3 Wrapper Constructor

`Proxy` classes must contain a wrapper constructor that takes a single argument of the type `java.lang.Object`, i.e., `MyProxy( Object object )`. The wrapper constructor allows a referent object to be effectively passed by value as an argument to a remote method call or returned by value from a remote method call. The referent object passed to the wrapper constructor must be an instance of the referent class or am `java.lang.IllegalArgumentException` will be thrown by the wrapper constructor.

27.4 equals() and hashCode()

`equals()` and `hashCode()` follow RMI semantics associated with stubs. Thus, two `Proxies` are considered equal if and only if they refer to the same referent object.

27.5 Clonable and Serializable

The `Proxy` interface extends `java.lang.Clonable` and `java.io.Serializable`. Proxy classes must implement `java.lang.Clonable` such that the `clone()` method returns a new Proxy that refers to the same referent object. Proxy classes must implement `java.io.Serializable`.

27.6 getReferentObjectClassName() and getReferentClassClassName()

Proxy classes contain methods for returning the class name of both the referent object's class and the referent class's class. These may differ if the referent object's class is an extension or implementation of the referent class's class.

The method `getReferentObjectClassName()`returns the class of the referent object. The referent object is the target object of remote object method invocations on the Proxy object. Thus, `getReferentObjectClassName()` is declared as an object method on the Proxy.

The method `getReferentClassClassName()` returns the class of the referent class. The referent class is the target class of remote class (static) method invocations on the Proxy class. Thus, `getReferentClassClassName()` is declared as a static method on the Proxy.

28 Network Class Loading

RMI network class loading provides a mechanism by which argument and return value (including exceptions) classes may be loaded. The dynamic services model inherits this mechanism and extends it to include network class loading of referent classes when performing remote instantiation and class method invocations.

28.1 Class Loaders and Deployments

As previously mentioned, each deployment has a single, unique code base and associated class loader. One can consider a station as containing the primordial class loader (also known as the system or null class loader) and a number of deployment class loaders, which are children of the primordial class loader. The primordial class loader loads infrastructure including the Java Runtime Environment (JRE), extensions, Jini technology classes, and the FMA implementation classes. The java.rmi.server.codebase property is the code base assigned to classes loaded by the primordial class loader. When one of these classes is passed to a remote JVM, the remote JVM will use the RMI code base property of the originating JVM to load the class. This property cannot be changed at runtime; therefore, updates to the infrastructure classes require restarting the station JVM. The value of the RMI code base property is implementation dependent.

Unlike infrastructure, classes and resources for dynamic services are not placed in the CLASSPATH or loaded by the primordial class loader. They are packaged into a deployment group and deployed on a station. The station accepts the JARs and stores them somewhere such that they may be accessed through the HTTP class server associated with or contained by the station. The station assigns a code base, containing the URLs that can be used to network load the JARs, to the deployment. The station also

inspects the manifest of the JARs in order to build an inventory map of package/version tuples to code bases. When the station needs to load a class to support remote instantiation, remote class method invocation, or activation, the station consults this map to retrieve the code base associated with the latest version of the class's package. Given this code base, the station then requests the RMI class loader to load the needed class.

28.2 Class Loading During Remote Instantiation

During remote instantiation, the station needs to load the class of the object to be instantiated. The search for the class shall proceed as follows.

1) Retrieve the code base of the latest version of the class's package using the local inventory.
2) If a code base was found, attempt to load the class using `java.rmi.server.RMIClassLoader`. The RMI class loader will always attempt to use the parent class loader before trying the code base.
3) If a code base was not found, attempt to the class using the primordial class loader (`java.lang.Class.forName(...)`).
4) If the class has still not been found, attempt to load the class using the client's RMI code base, if not null. This code base is embedded in the operation request. The client is the JVM requesting the remote operation.

28.3 Class Loading During Remote Class Method Invocations

During remote class method invocation, the station needs to load the referent class. The search for the class shall proceed as follows.

1) Retrieve the code base of the latest version of the class's package using the local inventory.
2) If a code base was found, attempt to load the class using `java.rmi.server.RMIClassLoader`. The RMI class loader will always attempt to use the parent class loader before trying the code base.
3) If a code base was not found, attempt to the class using the primordial class loader (`java.lang.Class.forName(...)`).
4) If the class has still not been found, attempt to load the class using the client's RMI code base, if not null. This code base is embedded in the operation request. The client is the JVM requesting the remote operation.

28.4 Class Loading During Activation

During activation of a persistent object, the station needs to load the class of the object being activated. The class shall be located according to the following sequence.

1) Retrieve the code base of the latest version of the class's package using the local inventory.

2) If a code base was found, attempt to load the class using `java.rmi.server.RMIClassLoader`. The RMI class loader will always attempt to use the parent class loader before trying the code base.
3) If a code base was not found, attempt to the class using the primordial class loader (`java.lang.Class.forName(...)`).
4) If the class has still not been found, attempt step 1 using the code base stored with the object. If this succeeds, activation shall continue but a warning, in the form of a log message, shall be issued indicating that an out of date class has been loaded during activation because of a backwards compatibility.

The semantics of class loading during activation is to use the latest available version of a particular class. If activation fails with the latest version, an attempt is made to recover by using the version that was in effect when the object was stored. Note that this requires the persistence of the object to include the code base, effective at the time of persistence, for the object's class. This is considered a recovery scenario and shall result in a log message to that effect.

29 JavaBeans Conventions

Referent objects shall follow the JavaBeans design patterns with respect to properties and methods. Accordingly, methods are classified as mutators (setters), accessors (getters), or complex (all others). Station implementations may use this classification of methods to map to security mechanisms. For example, an administrator may be able to access all methods while a user may only be able to access accessors.

Security

The following security mechanisms are addressed by an implementation of this security model:

1) Remote authentication and authorization
2) Delegation
3) Auditing
4) Cryptographic data protection (confidentiality, integrity,..)

As this specification is only concerned with standardizing vendor boundaries to allow interoperability, the specification standardizes only those mechanisms, such as security domain and federations, which serve as a basis with interoperability between stations, clients, managed resources, and dynamic services. This version does not attempt to standardize administration interfaces, such as key management, of stations, services, or any other entity.

The following is the list of security issues specified in this document:

1) Security topology
2) Authentication and authorization mechanism. Station authentication and authorization is based on the Java Authentication and Authorization Service (JAAS) and the standard Java security model. Although the reader is assumed to be familiar with both of these security technologies, the following sections review the JAAS while presenting additional specifications and constraints associated with this security model.
3) Role based access control
4) Key/certificate infrastructure
5) Delegation

The security architecture is one of a security domain with a trusted third party, the security service for the domain.

30 Trusted Third Party Architecture

30.1 Security Domains

Security domains are realms of trust against which subjects are authorized and Roles defined. Management domains and security domain, though separate concepts, are mapped one-to-one and share the same domain name. Thus, a management domain of name "boulder" belongs to a security domain of name "boulder". Each security domain has a single well-known security services as a trusted third party. A single security service, however, may serve more than one security domain, if supported by the particular security service implementation.

Entities in one security domain do not understand nor trust the security credentials of another security domain. When communication occurs between security domains, the party initiating the communication must join the target security domain as a client.

30.2 Federations

As a station may only belong to a single management domain, it may only belong to a single security domain. A station has an associated well known Subject representing the authentication of the station itself. Some stations, by virtue of being authenticated as a federation member, a special status with a security domain, belong to the containing security domain's federation and are completely trusted by other participants in the security domain. Clients, when discussing security, are considered to be JVMs that participate in a security domain that are not stations. Clients may participate in more than one security domain.

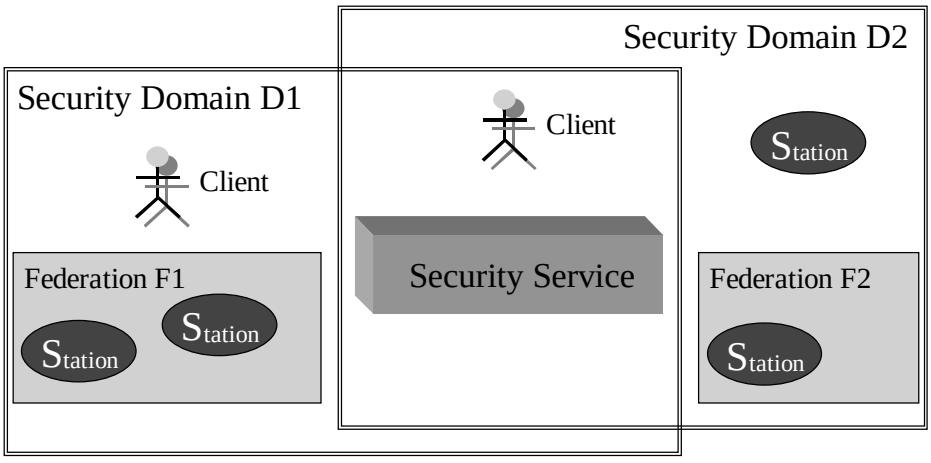

Figure 12. Security Architecture.

31 Scope of Specification

This specification standardizes only the programming interfaces at vendor boundaries. With respect to security, the main boundaries are as follows.

31.1 Client/Station to the JAAS (Authentication)

Clients and stations must authenticate themselves with the security services of the security domains in which they participate. Such authentication is performed by the authentication portion of the JAAS together with a plug in login module that communicates with the security service of the management domain.

31.2 JAAS to the Security Services

Because the proper JAAS login module and the security service can be supplied by different vendors, the interface to the security service is specified.

31.3 Service Objects to the JAAS (Authorization)

On the server side, service objects must be able to perform authorization checks. An interface for performing such checks is specified as well as a migration path from JDK1.2 to JDK1.3, the officially supported JDK for the JAAS.

31.4 Client to Proxy

A client, in the sense of something invoking operations on a Proxy, is required to present certain security information to the Proxy. This information, principally a Subject of specific composition, is specified. This point is addressed in a subsequent section entitled Client to Proxy on Page 68.

31.5 Referent Objects to Station

Referent objects inform the hosting station about the sensitivity of certain operations as well as whether a particular operation will require delegation. This point is addressed in a subsequent section entitled Referent Objects to Station on Page 68

32 Terms and Definitions

The following terms are fundamental to the security model. Additional terms and definitions are introduced as needed in the course of the chapter.

32.1 Subject

Subject is a JAAS concept that represents the source of an operation request, such as a person or service. Once authenticated, a Subject is populated with associated identities, or Principals, as explained later. A Subject may also be populated with credentials such as certificates, tickets, and keys. The public credentials of a Subject can be accessed without

restriction. Accessing private credentials, such as private keys, requires special permissions.

Subjects are associated with threads of execution and carried in context. Thus, at any point in a call stack one may retrieve the Subject associated with the current thread and perform an authorization check to verify whether the Subject has permission to perform a particular operation. This authorization based on Subject, supplied by the JAAS, augments the Java authorization model that is based on the level of trust in a class. The method of associating a subject with a thread of execution and retrieving the subject associated with the current thread is provided by the JAAS. The two security models (JAAS and Java) are fully described in the security documentation associated with the JAAS and the JDK.

32.2 Principal

Subjects are populated, during authentication, with ***Principals***, a JAAS and Java technology concept. Population of a subject requires specific permissions; thus, the association between subjects and principals is trusted to the same extent as the entity performing the population (usually a JAAS Login Module). A Principal may be thought of as one possible name for the subject. For example, the Solaris Login Module associates three Principals with a Subject during authentication:

1) SolarisUserPrincipal (user name),
2) SolarisNumericUserPrincipal,
3) SolarisNumericGroupPrincipal.

Each of the Principals has a type, given by its Java class, and a name.

If the creator, generally a Login Module, of the Principal is trusted (to create only trusted classes of Principals and give them trusted names) then the type and name of the Principal can be trusted. If a Principal is passed remotely from a source to a destination, the destination must be able to establish its own trust in the Principal (class and name). This generally involves the source proving to the destination, by the presentation of certain credentials, that it is in fact a legitimate holder of the Principal. Clearly, such credentials are sensitive information.

32.3 Stations versus JVMs

Not all parties in the security domain are stations. Parties that communicate with remote objects, rather than hosting remote objects, need only be a client JVM in which the appropriate client side classes (Proxy support) have been loaded. Only in a few instances is it necessary to treat stations as if they were different from client JVMs. In the context of security, the following classes of JVMs are referred to by this specification:

1) JVM	A client or station JVM participating in a security domain.
2) Station	A JVM enabled to support the dynamic services model: capable of hosting remote objects.
3) Client	A JVM capable of communicating with a station

4) Authenticated JVM A JVM with an authorized and well-known Subject. Authorized objects within the JVM can access and use the well-known subject of the JVM using the `javax.sxi.security.WellKnownSubject` class. If the JVM is authenticated as a server during communications, it will present its well-known Subject as its identity. Variations are ***authenticated station*** and ***authenticated client***.

32.4 Security Policy

The ***security policy*** is a Java concept manifest by a security policy "file". In this context, a "file" is usually a file in terms of the local operating system, but can be multiple files or remotely loaded data sources specified by one or more URLs. Java policy files consist of a number of entries granting permissions to classes, as specified by code base or the signer of a class. For example, the following is an entry granting code from the `/home/sysadmin` code base directory read access to the file `/tmp/abc`.

```
grant Codebase "file:/home/sysadmin/" {
    permission java.io.FilePermission "/tmp/abc", "read";
};
```

The JAAS extends the Java security policy file syntax to include entries that grant permissions to Principals and, therefore indirectly to Subjects.

```
// grant permissions to a Code/Signers/Principals triplet
grant
Codebase "www.foo.com",
SignedBy "bar",
Principal com.sun.security.auth.SolarisPrincipal "duke" {
 permission java.io.FilePermission "/duke", "read,write";
};
```

In addition to permission grants, the policy file specifies the location of the key store containing the certificates used to verify class signatures. See the JDK security documentation for further information about policy files and the policy tool that can be used to create and edit such files.

Policy files can be specified, using standard options, when starting a JVM. Implementations can provide other means of specifying security policy files such as secure remote loading. Authorization uses the policy file; thus, a policy file is loaded into the JVM controlling the resources to be protected by the policies.

32.5 Role

Role is a concept introduced to ease the burden of creating and maintaining access control lists. The JAAS service does not standardize the Principals associated with a Subject. As a result, the security policies (access control lists - described later) depend,

indirectly, on the method of authorization. For example, if the security policy granted permissions only to SolarisUserPrincipals and a NIS login module was used to authenticate a Subject, the Subject would not have any permissions because the NIS does not populate the Subject with Solaris principals. The net effect is that independent security policies would have to be maintained for each method of Subject authentication. Roles, a specialization of Principal, reduce the cost of maintaining security policies by standardizing the Principal used for management system security.

```
package javax.sxi.security;

import java.security.Principal;
import java.io.Serializable;

/**Role represents a standard, abstract kind of Principal
 *  for roles based authorization. Two roles are
 *  considered equal iff they have the same Role name,
 *  security domain name, and the same class name.
 */
public class Role implements Principal, Serializable
{
    /**Construct a new Role object. Role names and
     *  security domain names may not contain unmatched
     *  braces "{}".
     *  @param roleName Name of the Role. This is the
     *     Principal name and may be retrieved using
     *     Principal.getName()
     *  @param securityDomainName Name of the security
     *     domain against which this Role was issued.
     *  @throws IllegalArgumentException roleName or
     *     or securityDomainName are null or contain
     *     unmatched braces.
     */
    public    Role(
                  String roleName,
                  String securityDomainName
              );

    /**Get the role name of this Role.
     */
    public final String getRoleName();

    /**Get the name of the security domain against which
     *  this Role was issued.
     */
    public final String getSecurityDomainName();

    /**Get composite name of this role in the form
     *  "{roleName}{securityDomainName}".
     */
    public final String getName();
}
```

Permissions may only be granted to Roles, code bases, and signers. For example, a security policy file might contain the following entry. Note that this technique is compatible with the JAAS and Java security.

```
// Grant permissions to a Role, regardless of code base
// and signers.
grant
Principal javax.sxi.security.Role "administrator" {
 permission java.io.FilePermission "/duke", "read,write";
};
```

A Subject, when authorized, can be populated with many Principals, including more than one Role, and all of these Principals are applicable to permissions within the JVM in which the Subject was authorized (local security). However, when the Subject is passed (explicitly or implicitly) to a remote JVM, the resultant Subject in the remote JVM shall consist of a single Principal that is a Role. In the remote propagation of Subjects, the following rules apply:

1) Principals contained in the Subject are not propagated unless they are Roles.
2) A Role shall not be propagated unless it has an associated `RoleKey` credential. (Role Keys are described later in relation to secure Subjects.) This credential is the basis of the propagation.
3) Only Roles that belong to the same security domain as the remote (target) station, if it belongs to a security domain, are eligible for propagation. If the remote station does not belong to a security domain, no Roles shall be propagated.
4) If more than one eligible Role/credential pair exists, the method by which an implementation chooses which single Role to propagate is undefined.

While the use of Roles and the propagation of only a single Role have benefits (simplifying the security model, implementation, and maintenance), there are associated costs. If a user is compromised, the roles to which that user can access are also compromised. This is a result of users being authorized with respect to a role rather than respect to each user. The security model allows the end administrator, who configures the security system, to choose an appropriate tradeoff between simplicity and the size of a compromised Role scope. In one extreme, each user has a unique Role. Permissions would be maintained against each user. In the other extreme, there is a single Role that allows access to the entire system and is shared by all users.

32.6 Federations

A federation of stations is the set of authenticated stations that are considered as trusted as the security server of the security domain containing the federation. A federation member is trusted because the entity (perhaps an administrator) authenticating the station as such is trusted to make that determination. A federation is bounded by its security domain; thus, each security domain has a single, possibly empty, federation of trusted stations. The federation name and the name of the security domain containing the federation are the same.

Members of a federation trust each other because each has authenticated itself specifically as a member of the federation and, as a result, holds the private security key of the federation. Members of the federation can use the key to secure communications

among themselves. In this manner, a federation of JVMs becomes a web of trust where each member trusts the other and intra-federation communications can be secured. A consequence of this simplification is that a malicious member of the federation can compromise the federation. Because of this vulnerability, the entity providing the authentication credentials, such as a password, necessary to join a federation is responsible for ensuring that an installation (JVM, security policies, etc.) is, in fact, secure.

Within a security domain, entities outside of the federation trust the federation; however, the trust is not reciprocated: federation members do not, in general fully trust non-members.

32.7 Security Manager and Class Loaders

An implementation shall not depend on installing its own class loaders or security manager. In general, implementations shall not assume that they own the hosting JVM and shall be well behaved with existing class loaders, including the RMI class loader, and security manager.

32.8 Security Service

The primary responsibility of a security service is to provide authentication services to the participants of a security domain. The security service provides credentials, encapsulated in an authenticated Subject, that the authenticating party may use to prove its authentication to other parties that trust the security service. Clients, stations, and federations depend on the trust of the security service. The security service is, of course, quite sensitive; a compromised security services can compromise the entire security domain.

There is a single security service per security, and therefore management, domain, but security domains can share a security service. The limiting factor is the reliable reachability of the security services from the security domains and the ability of a particular security service implementation to support multiple domains, which is not required.

33 Security Topology

The following Figure shows the topology of two security domains served by a single security service. The Figure also shows the distributed of various keys and certificates at a certain point in time. The depicted client has been authenticated against two security domains and, therefore, has an authenticated subject with two different roles, one for each security domain.

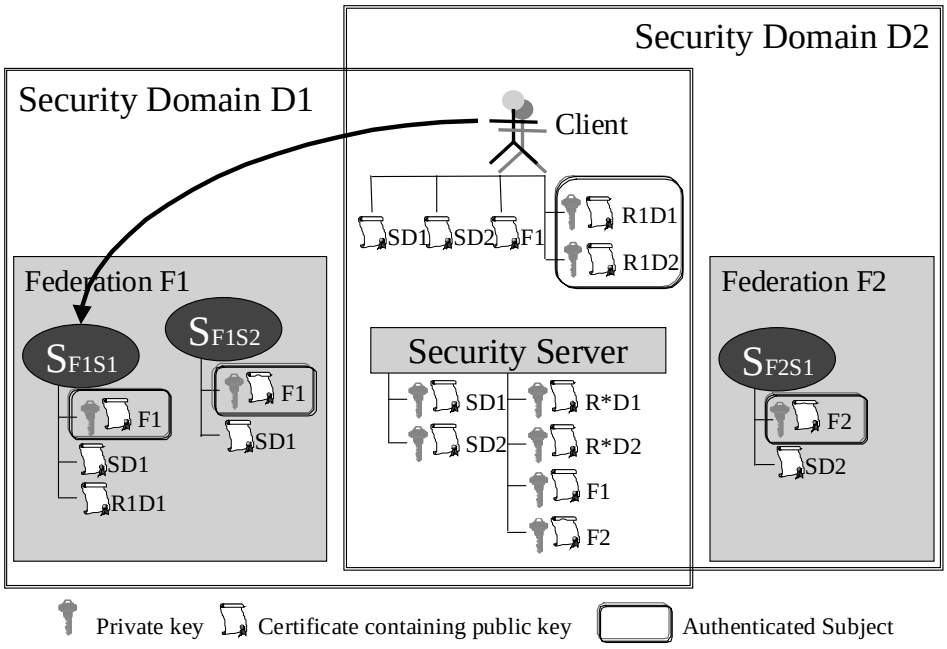

Figure 13. Security services, Security Domains, Federations, Stations, and Clients.

Private keys are shown as key icons and certificates (which contain the public keys) as certificate icons, both with annotations as follows:

1) F<d> where d is the security domain name.
2) SD<d> to represent the key pair for the security domain d.
3) R<r>D<d> to represent the Role r in the security domain d. Note that Roles are qualified by security domain. Although two security domains may have Roles with the same name, they are not considered the same role because security domains are the scope limit of Roles.

Key/certificate pairs are issued as a result of authorization against a security service and are encapsulated in a Subject. Often the authorized Subject is the well-known Subject of the particular entity.

In text, the annotations are subscripts, the private key icon is represented by K' and the certificate (public key) by K. For example, the private key for the security domain 1 is represented by K'_{SD1}. Authorized Subjects are represented by an S{principal list}, such as S{R1D1, R1D2}. Unauthorized Subjects have an empty list, S{}.

Stations are denoted by $S_{D<d>S<s>}$ where s is the identification of security domain to which the station belongs and s is the station identification.

Private keys are issued only in response to authorization and, unlike certificates, are never persisted or otherwise passed outside of the JVM to which the security service granted the key. Private keys are kept as a private credential of the authorized Subject. Thus, even within the JVM containing the Subject, access to privates keys is only

allowed to trusted code. In general, the only code trusted with this access is the Login Module, which provided the private key during authorization, and the communications infrastructure that needs the key to support secure communications. All others shall be denied access to prevent malicious code within the JVM from compromising the Role. However, none of these mechanisms can protect the private key if the JVM itself is compromised.

33.1 Certificates

The certificate for the security domain, SD<d>, boots the security mechanism by establishing trust in the security service itself. The means by which this certificate is obtained, or more generally, how trust is established with this certificate, is not part of this specification and expected to be handled as appropriate for a particular implementation and installation.

To permit implementation independence while preserving interoperability, the specification does standardize the format of all certificates to be X.509 (v1, v2, or v3). In addition, security domain certificates shall have a subject distinguished name common name equal to the name of the security domain. All role and federation certificates shall be signed by the private key of the security domain against which the certificates were issued. Thus, a holder of a trusted security domain certificate can establish trust in the role and federation certificates of that security domain. The subject distinguished name common name of Role certificates shall be the name of the Role.

The security domain certificates are the only ones that must be stored persistently. The authorization process, in particular a Login Module (described in the following section), must be able to locate the security domain certificate for a particular domain. Security domain certificates shall be made available through the Java key store facility under the alias "`javax.sxi:<security domain name>`", such as "`javax.sxi:boulder`". The `keystore` location is specified as a URL by the system property "`javax.sxi.security.keystore`". If this property is not specified, the Login Module will look for the keys stored in a file "`.keystore`" in the user's home directory as indicated by the standard system property "`user.home`". (See the JDK documentation for a precise algorithm by which the home directory is determined for various operating systems.) For conventional installations, providing the security domain certificates is a matter of storing the certificates in the default key store file using the `keytool` tool supplied with the JDK.

34 JAAS Authentication Overview

Authentication is the secure process of associated Principals and, optionally, credentials with a Subject. The association shall be done in such a manner as to be trusted by the mechanism performing authorization. Note that both the association of a Principal with a particular Subject and the class of the Principal must be trusted. The class of the Principal must be trusted to trust the Principal name, which is a factor in granting permissions.

The JAAS authentication mechanism is designed primarily for the local (within a JVM) authentication of users. In its most basic form, the JAAS authentication algorithm

performs a multi-phase login across one or more trusted Login Modules, with each Login Module being specific to a method of authorization, such as host based Solaris operating environment authentication or NIS authentication. As the Login Modules are trusted, (they have been granted certain permissions by the policy file associated with the JVM in which authentication is being performed) they have permission to add Principals and credentials to the Subject of the login. Thus, the composition of the resulting Subject is trusted to the extent that the relevant policy file (and JVM) are trusted. In general, this level of trust is sufficient to make authorization decisions when the Subject is authenticated in the same JVM in which the authorization is taking place. Further details can be found in the JAAS documentation.

35 Management Extension to JAAS Authentication

The JAAS is sufficient when the authentication and authorization occur in the same JVM. In the case of the security model of this specification, authorization and authentication can happen in different JVMs. In particular, unless other measures are taken beyond JAAS, the authorization process in a distributed environment cannot trust a Subject because the authorization cannot trust the source of the Subject, nor can it trust the communication channel by which the Subject was transferred from the authenticating JVM to the authorizing JVM. The purpose of the management extensions to JAAS are to establish the trust in foreign Subjects, that is, Subjects authenticated in a remote JVM.

Though the specification often refers to passing a Subject from one JVM to another, it is unlikely that an implementation would choose to literally pass a Subject. Rather, the Subject would generally be reconstructed in the target JVM based on security information supplied by the source JVM. In any case, the Subject is logically transferred, but the mechanism by which this transfer is achieved is implementation dependent.

35.1 Security Service

As previously described, the primary responsibility of the security service is to perform secure authentication. It does so by supplying a JAAS Login Module that communicates with the security service to perform authorization. The Login Module appears as a local Login Module, but the actual authorization is performed remotely in a trusted security service.

The JAAS, however, cannot directly use the Login Module as supplied by the security service because the JAAS expects to be able to *instantiate*, not retrieve a Login Module. There is also the issue of establishing trust in the Login Module and certificates supplied by the security service, which requires well known, but secure, access to the certificate of the security services. In addition, trust must be established in the proxy itself.

```
package javax.sxi.security;

import javax.security.auth.spi.LoginModule;
import java.rmi.RemoteException;
import java.security.cert.CertificateException;

/**Interface implemented by the security service proxy.
 */
public interface SecurityService extends TrustedProxy
{
    /**Factory method to get a new LoginModule for
     * authentication.
     * @param securityDomainName The name of the security
     *     domain against which the authentication should
     *     be performed.
     */
    LoginModule  newLoginModule( String securityDomainName )
       throws
          RemoteException,
          CertificateException;
}
```

The proxy must be trusted to verify its communications with the security service and for providing the certificate to the Login Modules returned by `newLoginModule()` so that the Login Modules can also validate their communications with the security service. In general, this requires that the proxy fetch the certificate of a particular security domain from the key store. The alias used to access the certificate as well as the search for the key store file are outlined in the previous section about certificates.

To establish trust in a proxy, one must establish trust in the proxy code, which may have been loaded from a remote source. This is a question of whether the client, not the server, trusts the proxy. Therefore, the validation depends on information available on the client side. A trusted proxy is a proxy to which the client has granted `javax.sxi.security.SecurityPermission` with a target of "`trustProxy`". This will require that the local (client side) policy file contain a grant entry granting this permission to the proxy supplied by the security service. Note that the particular class of this proxy will necessarily depend on the security service implementation. Changing the implementation of a security service will require changing client side policy files. The details of the process of trusting a proxy are general to all secure proxies and outlined in the following code segment.

```
// Check to see if a particular object is trusted as a
// secure proxy. Note that this check requires that the
// entity doing the checking have permission to access
// the protection domain of the class
// RuntimePermission("getProtectionDomain"). The
// proxy being checked must have the
// javax.sxi.security.SecurityPermission( "trustProxy" )
// permission to be trusted.
...
boolean trusted = false;
try
{
   trusted =
      aProxy.getClass().getProtectionDomain().implies(
         new javax.sxi.security.SecurityPermission(
               "trustProxy"
         )
   );
}
catch( NullPointerException e ) { }
...
```

To establish this trust, the proxy must have been signed, presumably by the vendor of the security service to which the proxy refers. Note that this trust only verifies that the *class* of the proxy is trusted. A proxy class worthy of such trust must ensure that it can trust the parties with which it communicates.

```
package javax.sxi.security;

/**Interface implemented by secure proxies.
 */
public interface SecureProxy { }
```

Proxies may implement the `javax.sxi.security.SecureProxy` tag interface. Implementations of stations may allow a mode in which proxies tagged as secure are only allowed to be loaded into the station if they are trusted according to the described trust testing algorithm. Thus, proxy clients can assume the proxy is trusted if it implements `javax.sxi.security.SecureProxy`.

```
// Grant statement for security services proxy signed by
// the vendor "wahoo". The "wahoo" certificate will need
// to be available from the key store.
grant SignedBy "wahoo" {
   permission javax.sxi.security.SecurityPermission
      "trustProxy";
};
```

The proper JAAS Login Module is a JAAS compliant login module that knows how to communicate with the security service. The implementation of the proper Login Module is independent of the security service implementation and may be provided by the

security service vendor or as part of a software development kit. The proper Login Module is a thin object that, when instantiated, locates a security service proxy, which shall be validated as trusted, to an appropriate remote security service. To request a Login Module from the security service, the Login Module must know the security domain name against which authorization shall be performed. This name is provided by the system property "`javax.sxi.security.domain`". This name is identical to the name of the management domain to which the security domain belongs. From then on, all method invocations are delegated to the Login Module provided by the security service.

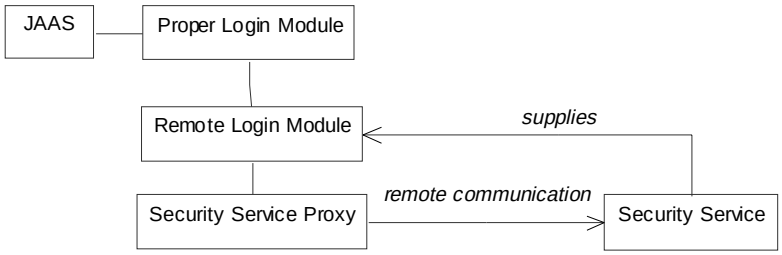

Figure 14. Remote Authorization Model.

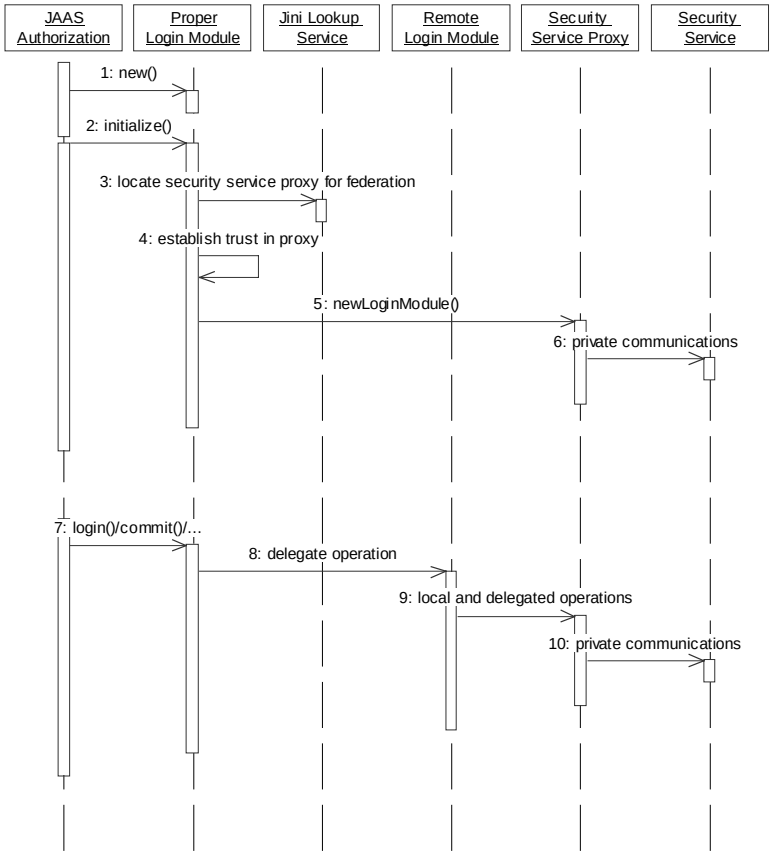

Figure 15. Remote Authorization Sequence.

1) The JAAS authorization module instantiates the proper Login Context. This Login Context is specified in the JAAS configuration.

2) The JAAS authorization module initializes the proper Login Context. Some of the information that must be available to the proper Login Context at the time of initialization is the public certificate for the security service and the security domain name.

3) Using the security domain name, the proper Login Context can locate an appropriate Jini lookup service using the group name "`<security domain name>`", such as "`us.co.boulder`".

4) The proper Login Context must establish trust in the proxy retrieved from the lookup service.

5) Once trust has been established, the proper Login Context requests a new Login Module from the security service for a particular security domain.

6) The security service proxy performs any needed private communications with the security service. Note that the proxy has access to the certificate of the specified security domain by way of the local key store.

7) The JAAS authorization module can perform any number of operations on the proper Login Context, such as `login()` and `commit()`.

8) The proper Login Context delegates these operations directly to the remote Login Module. Note that the remote Login Module is remote to the security service, but local to the proper Login Context.

9) The remote Login Module does whatever is necessary to perform the requested operation. In general, this will involve both local operations and remote communication with the security service.

10) The communication between the security service and its proxy is private. The proxy and security services shall appropriately secure all such communications. This may be done using the public/private key pair associated with the security services.

The security service implementation must satisfy a number of requirements:

1) The implementation cannot assume that the JVM hosting the Login Module can perform socket accepts. In general, this means that some sort of duplex communication or polling must be provided to simulate the Login Module callbacks.

2) All communications between the security service proxy and the security service shall be server authenticated, tamper resistant, and private. In particular, the proxy will be passing sensitive information, such as user names and passwords, to the security service. This information shall not be sent as clear text. Because the proxy cannot always distinguish between sensitive and public information, all communications shall be private. Because the security service has the private key of the security domain and the security service proxy has access to the public certificates of each security domain (from the local key store), providing secure communications should be straightforward.

3) The Login Module shall populate the Subject with a single Role and an associated private credential, represented as a `RoleKey`, containing the private key and public certificate of the authenticated Role. The `RoleKey` and Role are matched by Role.

4) If the authenticated Role was as a federation member, the Role shall be of type `FederationMember`.

35.2 Secure Subject

A secure Subject is an authenticated Subject containing one or more Role objects and matching `RoleKey` objects as private credentials. Referring back to Figure 13, each private key associated with a JVM (as opposed to the security services) is contained, along with its public certificate, in the RoleKey of a secure Subject as a result of authentication.

```
package javax.sxi.security;

import java.io.Serializable;
import java.security.PrivateKey;
import java.security.cert.Certificate;

/**Private credentials associated with a Role. The
 *  Role/RoleKey relationship is established by the
 *  Role property of the RoleKey. The RoleKey also
 *  contains the public key, as part of the Role's
 *  public certificate, and the private key for the
 * Role.
 */
public final class RoleKey implements Serializable
{
    /**Create a new RoleKey object.
     * @param key The private key for the associated Role.
     * @param certificate The public certificate for the
     *     associated Role.
     * @param role The associated Role.
     */
    public RoleKey(
              PrivateKey key,
              Certificate certificate,
              Role role );

    /**Get the private key for the associated Role. */
    public PrivateKey      getKey();

    /**Get the public certificate for the associated Role.
     */
    public Certificate  getCertificate();

    /**Get the associated Role. */
    public Role            getRole();
}
```

If the secure Subject was the result of the authentication of a federation member, then the Role associated with the Subject is a FederationMember object.

```
package javax.sxi.security;

/**Special Role issued to members of a federation.
 *  Note that FederationMember may be subclassed to
 *  establishRoles even within a Federation.
 */
public class FederationMember extends Role
{
    /**Construct a new FederationMember Role object.
     * @param roleName Name of the Role. Should not be
     *     null or contain unbalanced braces.
     * @param securityDomainName Name of the security
     *     domain issuing the Role. Should not be
     *     null or contain unbalanced braces.
     * @throws IllegalArgumentException if roleName or
     *     securityDomainName is null or contains un-
     *     balanced braces "{}".
```

```
     */
    public    FederationMember(
                  String roleName,
                  String securityDomainName
              );
}
```

The private credentials of a secure Subject shall never be exposed directly on the network, in persistent storage, or other unsecured environment.

35.3 Well Known Subject

Each authenticated station has a well-known, authenticated Subject. Authentication or re-authentication can occur at any time simply by proceeding through the remote authorization procedure and making the authenticated Subject well known. Authorized objects have access to the well-known Subject and can, therefore, assume the identity of the well-known Subject. Only privileged objects, such as the communication infrastructure, shall be allowed access to the `RoleKey` of the well-known Subject. This requires the permissions as specified by the JAAS. Usually only the Login Module has permission to invoke `javax.sxi.security.WellKnownSubject.setSubject()` to change the well known authenticated Subject.

```
package javax.sxi.security;

import javax.security.auth.Subject;

/**Support for setting and getting a well known Subject.
 */
public final abstract class WellKnownSubject
{
   /**Set the well known Subject.
    * Requires javax.sxi.security.SecurityPermission with
    *    a target of "setSubject".
    * @param The well known Subject. May be null to
    *    clear.
    * @return The previous well known Subject.
    * @throws java.security.AccessControlException
    *    if permission to set the Subject is not
    *    granted to the caller.
    */
   public static Subject  setSubject( Subject subject )
      throws AccessControlException;

   /**Get the well known Subject.
    * Requires javax.sxi.security.SecurityPermission with
    * a target of "getSubject".
    * @return The well known Subject.
    * @throws java.security.AccessControlException
    *    if permission to retrieve Subject is not
    *    granted to the caller.
    */
   public static Subject  getSubject();
}
```

36 Authorization

Authorization is the verification that a particular thread of execution has permission to perform a particular task, such as accessing a secure resource. This verification may require inspecting both the classes involved in the call chain, standard Java security, and the Subject associated with the current thread, the JAAS extension to the standard model. JAAS authorization is supported by Java 2 version 1.3 and later. The specification provides an similar means of authorization for version 1.2 for migration purposes.

36.1 JAAS Overview

JAAS authorization is directly supported by the JDK 1.3 AccessController, thus authorization does not require the use of any JAAS class and is retroactively applicable to classes written prior to JAAS. In general, authorization is done by performing a check permission call on the `AccessController` class.

```
FilePermission perm =
   new FilePermission( "/temp/testFile", "read" );
AccessController.checkPermission( perm );
```

This permission check verifies whether the current thread is allowed read access to the specified file. The decision is based on the grant entries in the policy file associated with the JVM, the class of objects in the call chain, and the Subject (specifically the Principals of the Subject) associated with the current thread. The Java and the JAAS security documentation fully detail how the permission checks are calculated.

36.2 Modifications

The security model specified is fully compatible with, and does not require modifications of, the JAAS authorization mechanism. However, to allow early use of authorization (before wide availability of JDK 1.3), this specification provides the following convenience class for migration of JAAS authorization.

```
package javax.sxi.security;

import java.security.AccessControlException;
import java.security.Permission;

/**AccessController provides a temporary entry point
 *  for java.security.AccessController.checkPermission()
 *  for Subject based authorization until it is available
 *  directly in the JDK.
 */
public final class AccessController
{
   /**Temporary implementation of
    *  java.security.AccessController.checkPermission()
    *  for Principal (Subject) based authorization without
    *  JDK 1.3.
    *  @see java.security.AccessController#checkPermission
    */
   public static void checkPermission( Permission perm )
      throws AccessControlException
}
```

The intent is to easily replace the implementation of this class when the JAAS becomes widely available. With the arrival of JDK 1.3, one should be able to change package names without perturbing source code in any other fashion.

36.3 Station Authorization

Stations may perform authorization when a remote operation is requested of the station. The permission required to perform the remote operation is of class `javax.sxi.security.AccessPermission`.

```
package javax.sxi.security;

import java.io.Serializable;
import java.security.Permission;

/**This class represents remote access to a station
 *  and its contents. In particular, operations invoked
 *  on the Station interface or the Acceptor interface
 *  may require specific AccessPermission.
 *  <P>
 *  An AccessPermission object consists of a target name
 *  and an action.
 *  <P>The target name is the name of the referent class
 *  or the class of the referent object, if applicable to
 *  the operation. A target name of the form
 *  "<package_name>.*" indicates all the class of the
 *  package. A target name of the form "<package_name>.-"
 *  indicates all the class of the package and
 *  subpackages. A target name of "*" indicates all
 *  classes.
 *  <P>
 *  Actions to be granted are as follows:
 *  <DL>
 *  <DT><code>"<signature>"</code><DD>object or
 *     class method invocation. <signature> may be "*"
 *     for all methods, a simple name for all methods
 *     of a given name (arguments not included), or a
 *     full Java method signature. Constructors are
 *     considered class methods with the name "<ctor>".
 *     The target name is application to this action.
 *  <DT><code>"deploy"</code><DD>Deploy or recall
 *     deployment groups. Includes inventory and code
 *     base queries. Note that the target name is not
 *     applicable to this action.
 *  </DL>
 */
public final class AccessPermission
   extends Permission implements Serializable
{
}
```

If a permission check fails during a remote operation request, the station shall throw a `javax.sxi.security.StationSecurityException`.

```
package javax.sxi.security;

/**StationSecurityException is thrown by a station when
 *  a front door security fails during a remote operation
 *  request.
 */
public class StationSecurityException
   extends SecurityException
{
}
```

Security

37 Client to Proxy

Up to this point, this chapter has been mainly concerned with the architecture of the management security model and the interactions with the JAAS. In addition, there are two other vendor boundary interfaces to address. The first is client to proxy. The entire interaction that a client has with the proxy, with respect to security, is through the Subject, or security, context. The client is responsible for:

1) providing Subject authenticated against the security domain and

2) associating the Subject with the current thread of execution before invoking methods on the Proxy.

This association is done using the `Subject.doAs()` methods.

38 Referent to Station

The actual security mechanisms (encryption, auditing, etc.) invoked when communicating with a particular referent are a result of security policy, supplied by the administrator, applied to information supplied by the developer. The former is not in the scope of this specification. The latter is part of the referent to station contract. The information supplied by the developer is classified as intrinsic, implicit, and explicit.

38.1 Intrinsic

Intrinsic information includes class names, interface names, method signatures and any other information intrinsically available from any object. The developer makes no special effort to provide this information; however, security decisions may be based on this information, if supported by a station implementation. For example, a station could allow call auditing to be specified by the administrator on a class-by-class basis. As another example, auditing of all remote operations on a particular class of objects.

38.2 Implicit

Implicit information supplied by the referent to the station includes semantics associated with certain method patterns by virtue of the JavaBeans component model. Methods can be categorized as accessors, mutators, and others. Security decisions can be based on this classification. For example, one possible security policy would allow unauthenticated access to accessors but require authorized access to mutators and other methods.

38.3 Explicit

Explicit information is supplied by the referent in the form of modifiers. With respect to security, the modifiers are grouped into two sets: sensitivity and delegation. The administrator is not expected to know the details of particular classes or objects. Thus, the developer specifies the sensitivity of particular operations. The sensitivity is specified as public, sensitive, or private. Note that the developer does not specify what mechanisms should be used with each of the levels of sensitivity. The mechanisms are specified by the administrator based on, possibly, a combination of intrinsic, implicit, and explicit

information, as supported by the administration capabilities of a particular station implementation.

The delegation modifier is the means by which a referent informs the station that it may need delegation to perform a particular operation. In others, to perform the operation, the referent will need to authenticate itself as the client. Delegation is described further in the following section.

39 Delegation

Delegation is the process by which a client (an entity requesting a remote operation) grants the server (an entity receiving the remote operation request - referent) permission to use the client's Subject. As this requires passing the private credentials of the client Subject to the server, the client must have a high level of trust in the server. If a particular method has specified that it needs delegation, the client Subject will have sufficient credentials such that any thread that has access to the Subject may authenticate itself as the client. In general, delegation happens automatically. Consider the case where a client A invokes a method on a remote object B, which in turn invokes a method on another object C. If B requires delegation and C requires authentication, then (for security purposes) the call to C appears to have come from A. If C does not require authentication, then B does not necessarily need to require delegation because the private credentials of the Subject will not be inspected.

40 Views

Different roles see and need to understand different aspects of security.

40.1 Client Developer

The client developer must understand the JAAS authentication framework as well as the extensions of this chapter, principally the Login Module.

40.2 Service Developer

The service developer must tag classes and objects with private, confidential, public, and delegation modifiers as well as following JavaBeans coding conventions for properties. Generally, this is done with tool assistance, but it is possible to perform the task manually by creating the modifier tables by hand. In advanced cases, developers may wish to perform explicit security checks (`javax.sxi.security.AccessController`) or assume the identity of the station (`javax.sxi.security.WellKnownSubject`).

40.3 System Administrator

The system administrator is responsible for providing security policies, both in the sense of Java security policy files and in the sense of specifying mechanisms. The former is covered by the security information associated with the JDK, the JAAS, and the specific permissions required to perform certain tasks. The latter is implementation specific but generally consists of some means by which the administrator can control encryption,

auditing, and the like based on various attributes, including the security modifiers assigned by the developer to particular methods.

Most of the complexity of security falls on the system administrator. The security model allows for a wide range of granularities. The following is an example of how an installation could be configured with a very coarse level of granularity to achieve a level of simplicity.

1) Grant all permissions to the role of administrator.
2) Grant permission for the roles of user to access get methods labeled as public.
3) Grant no permission to other roles.
4) Grant all permissions to classes signed by Sun, IBM, and the lead administrator.
5) Grant no permissions to other classes.
6) Encrypt communications with all methods tagged as private or sensitive.
7) Plain text communications with all method tagged as public.
8) Audit communications with all methods tagged as private.
9) Perform client authentication with all methods tagged as private or sensitive.

Aspects

Referents have three aspects: transaction, logical thread, and controller. Aspects are handled by the infrastructure on behalf of the referent according to aspect policy specified by the referent. These aspect policies are specified by the aspect modifiers (`SYNCHRONIZED_TRANSACTION`, `SYNCHRONIZED_LOGICAL_THREAD`, `SYNCHRONIZED_TRANSACTION_CONTROLLER`) applied to methods and constructors.

Each aspect serializes object access based on a particular concept, much in the same way as Java thread synchronization serializes object access based on language thread. For example, `SYNCHRONIZED_TRANSACTION`, serializes object access based on transactions such that the object may only be involved in a single transaction at a time. With all aspects, an exclusive lock on the object is acquired when a 'synchronized' method is invoked. (Each aspect has an independent lock.) The aspects differ however as to when the lock is released. Logical thread based locks are released when the invoked method returns. Transaction based locks are released when the transaction is committed or aborted. The basis for lock release is known as the *relevancy* of the aspect. Aspects also differ in their response to a failure to acquire an unavailable lock, as described in the following sections.

Aspects are only applied to an object when the object is accessed using its Proxy. Therefore, it is unsafe and forbidden to invoke any method with aspect modifiers except through the object's Proxy.

Much as adjunct modifiers are thought of as extensions to the Java language modifiers, aspects may be thought of as extensions to the Java thread synchronization primitive.

Transaction Aspect

41 Synchronized/Transactions

The developer indicates that a method is synchronized with respect to transactions by tagging the method with the `javax.sxi.server.Modifiers.SYNCHRONIZED_TRANSACTION` modifier. The set of such methods form an exclusion group such that, with respect to these methods, the object may only be involved with one transaction at a time. The semantics of the synchronization obey the following state diagram.

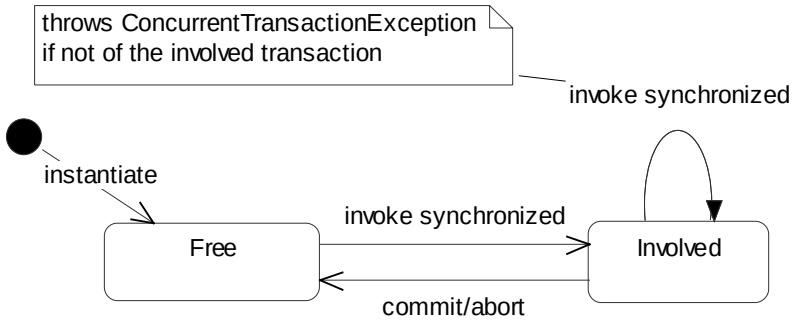

Figure 16. State Diagram of Object Methods Synchronized with Respect to Transactions.

42 Transactions Created on Behalf of an Object

If a thread of execution does not have an associated transaction (see `javax.sxi.common.Context`), then the station must initiate a transaction before invoking a method synchronized with respect to transactions. The station is then responsible for committing or aborting the transaction when the method returns. The

transaction shall by aborted if a throwable is thrown and committed otherwise. Stations may perform the following optimization. If an object is involved with a transaction and a synchronized/transaction method is invoked without a transaction, there is no need to create and abort the transaction before throwing the `javax.sxi.common.ConcurrentTransactionException`.

43 Deadlock Prevention

If the transaction lock is not available, the object is already involved. Then an exception is thrown. Thus, deadlocks will be broken; however, thrashing may result during contention for transaction locks. Stations shall attempt to acquire the transaction lock for a period of time specified by the "`javax.sxi.transaction_tolerance`" system property *before* throwing a `javax.sxi.common.ConcurrentTransactionException`. The default value is 10,000 milliseconds and is specified in milliseconds. Note that `javax.sxi.common.ConcurrentTransactionException`, is an unchecked exception.

Logical Thread Aspect

44 Synchronized/Logical Thread

The developer indicates that a method is synchronized with respect to logical threads by tagging the method with the `javax.sxi.server.Modifiers.SYNCHRONIZED_LOGICAL_THREAD` modifier. The set of such methods form an exclusion group such that, with respect to these methods, the object may only be involved with one logical thread at a time. The semantics of the synchronization obey the following state diagram.

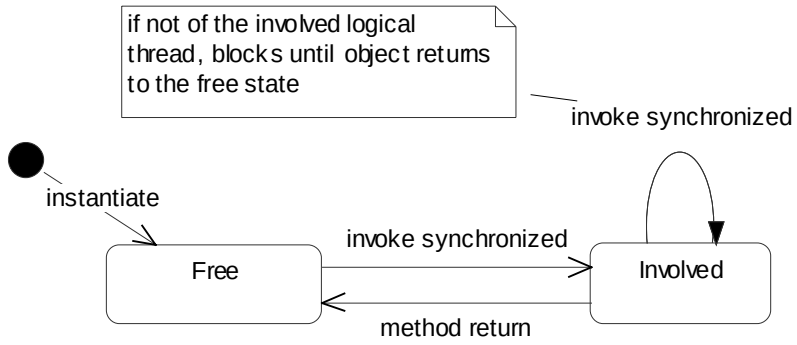

Figure 17. State Diagram of Object Methods Synchronized with Respect to Logical Threads.

45 Logical Threads Created on Behalf of an Object

If a thread of execution does not have an associated logical thread (see `javax.sxi.common.Context`), then the station must initiate a logical thread before

invoking a method synchronized with respect to logical threads. Stations may perform the following optimization. If an object is involved with a logical thread and a synchronized/logical thread method is invoked without a logical thread, there is no need to create the logical thread before throwing the `javax.sxi.common.CurrentThreadException`.

46 Distributed Deadlock

Distributed deadlock is a class of problems that are particularly difficult to diagnose and correct. Because objects in a distributed are more loosely coupled that in the local case, distributed deadlock is a more difficult situation than local deadlock. In general, the best cure is prevention by good coding practices. For example, one should avoid, as much as possible, holding locks when making *out* calls. Out calls are method invocations on objects, which are not encapsulated by the calling object. The calling object generally does not know what locks the target object will acquire. Therefore, it is not safe for the calling objects to hold any locks while invoking methods on the target object. A classic example of an out call is when a subject invokes a callback on an observer (Subject Observer pattern).

Clever use of synchronization and local variables can help release locks when it may initially appear impossible. However, sometimes it is simply not possible to release all locks before making an out call. If the out call is known not to cross a partition (involve a remote operation), one should judiciously use Java thread synchronization. Only as a last resort, when the out call involves a remote operation, should locking be performed based on logical thread. Methods synchronized with respect to logical threads should be rare.

To help avoid distributed deadlock even when logical threads are used judiciously, stations can be directed to give up waiting for a lock to become available after a given amount of time. This deadlock tolerance is controlled by the system property "`javax.sxi.thread_deadlock_tolerance`". The default value is infinite: a thread will block forever waiting to lock an object.

Controller Aspect

47 Controllers

Controllers allow various resources to be locked with respect to a controller for a long period of time: possibly the life of the controller, which may be persistent. This primitive forms the basis for control arbitration of managed resources. A controller must register itself with the controller service, a base service, and maintain the associated lease. The controller may then explicitly or implicitly reserve managed resources for its exclusive use. When the lease has been cancelled or expired, the reserved resources are released for use by other controllers. A controller's locks may also be released without releasing the controller itself. Unlike transaction and logical thread locks, controller locks are long lived.

48 Controller Architecture

While controllers are a common concept for both clients and stations, they are treated somewhat differently do to scalability requirements. Clients are expected to contain a few controllers, with a single controller being the most common case. Stations may contain thousands of controllers in large configurations. Thus, the interface used by clients and stations are slightly different.

48.1 Controllers

A Controller object represents a single controller, a single point in a chain of control. Generally controllers are dynamics services that determine policy or are management clients. Controllers are issued by the controller service for the management domain and may have over any number of generations. Clients contact the controller service directly to get a controller. Controller objects running in a station will have a controller allocated exclusively to the object by the containing station.

A given controller moves to its next generation when its owner, client or controller object, requests that the locks owned by the controller be released, or more precisely,

allowed to expire. The change in generation requires communication with the controller service.

48.2 Locks

Locks are issued by a controller against a specific generation of the controller. Given a lock, one can query the lock to see if it has been released. This query requires communication with the controller service. A lock is released under two conditions.

1) The lease maintaining the controller expires or is cancelled. In the case of controllers issued to clients, the lease if for a specific controller. In the case of controllers issued to controller objects in stations, the lease is for all controllers in the station.
2) If the controller issuing the lock is still valid, but is no longer of the generation that issued the lock. This allows a controller to effectively release all of its locks by changing generations.

48.3 State Distribution Between Stations and the Controller Service

Stations maintain a list of controllers that have been acquired on behalf of controller objects hosted by the station. The subset of this list pertaining to persistent controller objects must also be persistent. The controller service maintains a copy of this list so that other parties may query the validity of a lock without having to contact the station containing the controller, which may not be available. The station controller list is the master and the controller service list the slave.

The controller list in the controller service is leased by the station. Failure to renew this lease indicates that the station and controller service may be out of synch. The controller service provides a synchronization method to resynchronize with a station; however, loss of synchronization may imply the loss of controllers locks. The controller list state can change only as follows:

1) A controller is added.
2) A controller is removed.
3) A controller changes generation.

All of these changes require communication with the controller service in order to maintain synchronization.

48.4 Station Responsibilities

48.4.1 Remote Instantiation

When remotely instantiating a controller object, stations must contact the controller service serving the management domain to which the station belongs and request a new controller for the controller object. This controller is passed in context to the controller object whenever a method synchronized with respect to controllers is invoked remotely.

48.4.2 Controller Object Lifetime

When the controller object is garbage collected (transient) or removed (persistent), the station must also delete the associated controller. If it is unable to do so, state synchronization has been lost and the station should resynchronize with the controller service when the service again becomes reachable.

48.4.3 Remote Method Invocation

When invoking a method synchronized with respect to controllers, the station may need to verify the relevancy of a lock. When a lock is acquired on behalf of a controller, the station will need to request and retain a lock object from the controller.

48.4.4 Failed Lease Renewal

When the station fails to renew its lease with the controller service, it must start a prolonged attempt to resynchronize with the controller service. The retry interval shall be between 10 seconds and 5 minutes. If the station is not able to contact. The station shall assume that locks have been lost and notify controller objects as described in 48.4.6.

48.4.5 Station Restart

Station must persist their lease with the controller service and resume its maintenance when the station restarts by immediately attempting a lease renewal. Regardless of whether the renewal succeeds or fails, the station should begin state synchronization with the controller service because any transient controllers would have been lost in the station but still present in the controller service. If the renewal fails, the station must assume that locks have been lost and notify controller objects as described in 48.4.6.

48.4.6 Notify Controller Objects of Possible Lock Loss

If the station suspects the possible loss of controller locks, it must notify all controller objects to given them the opportunity to reestablish any locks that they may hold. All controller objects that implement the following method shall be notified as soon as practicable after the station detects the possible loss of lock integrity. Note that in many cases, the controller service will be unreachable it this time.

```
private void onControllerFailure()
```

After station the station has reestablished state synchronization with the controller service, it must inform the controller objects of the recovery. Controller objects that implement the following method signature shall be notified. In addition, the station should ensure that the controller of the controller object is established in context before invoking the method.

```
private void onControllerRecovery()
```

During the period between suspecting loss of lock integrity (lease renewal failure or remote communication failure with the controller service) and resynchronization with the controller service, the station should refuse all remote operations by throwing a `javax.sxi.services.ServiceFinder.ServiceNotFoundException`.

48.4.7 Persistent Objects

The list of controllers that must be synchronized with the controller service includes those of persistent controller object which are not activated. For example, on restart, the station must build a synchronization list of controllers associated with all persistent controller objects, none of which are activate at startup.

48.5 Client Responsibilities

Clients must directly contact the controller service to acquire a controller and are then responsible for maintaining the associated lease. Clients and there controllers are considered short lived and transient without any mechanism for reregistering a client controller.

49 Synchronized/Controller

The developer indicates that a method is synchronized with respect to controllers by tagging the method with the `javax.sxi.server.Modifiers.SYNCHRONIZED_CONTROLLER` modifier. The set of such methods form an exclusion group such that, with respect to these methods, the object may only be owned by only one controller at a time. The semantics of the synchronization obey the following state diagram.

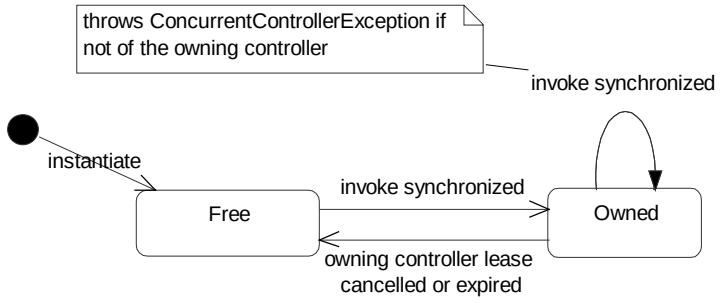

Figure 18. State Diagram of Object Methods Synchronized with Respect to Controllers.

50 Controllers Created on Behalf of an Thread

If a thread of execution does not have an associated controller (see `javax.sxi.common.Context`), then the station must create a controller before invoking a method synchronized with respect to controllers. The station is then responsible for canceling the controller lease when the method returns, regardless of whether or not the method threw a throwable. As with transactions, this is an expensive operation and clients should create a context (transaction and controller) to be used across many remote operations. Stations may perform the following optimization. If an object is owned by a controller and a synchronized/controller method is invoked without a controller, there is no need to create a new controller before throwing the `javax.sxi.common.ConcurrentControllerException`.

51 Deadlock Prevention

If the controller lock is not available, the object is already owned by another controller, then a `javax.sxi.common.ConcurrentControllerException` is thrown. Thus, deadlock is not a problem. Unlike transactions, however, controller locks are considered relatively static, thus contention thrashing is not likely. For this reason, stations are not required to attempt a controller lock over a period of time.

52 Clients as Controllers

Clients are always considered to be controllers. Clients must contact the controller service and request a controller to identify the client as a controller. It is possible that some forms of clients may be partitioned into more than one controller. A client controller must be associated with the current thread before invoking remote operations using Proxies.

53 Referent Objects as Controllers

Some referent objects should be controllers, known as controller objects. Consider the management control path from the source of activity (client, event service, or scheduling service) through the implementation object of one or more dynamic services to managed resources. Many of the intermediate object are points of control; they affect some sort of policy on the control path and are considered to be controller objects. Other objects simply route the management path or provide access to information: these are not controllers. Controllers include objects in services such as groupers, which manage a group of resources as a single consistent unit, and reactors that respond to an event by manipulating managed resources.

A class indicates that objects of that class are controller objects by using the `javax.sxi.common.Modifier.IS_CONTROLLER` modifier on the class. The station ensures that each controller object has an assigned controller and is responsible for

maintaining the associated lease. (The associated lease is actually the global lease between the station and the controller.)

53.1 Immutable Relationship Between Controller and Object

Each controller object has a single exclusive controller. The controller can only be set once and is an immutable relationship with the controller object. During remote instantiation of a controller object, the station must contact the controller service and request a controller on behalf of the object. This controller must be set in context before invoking the constructor of the controller object.

While the relationship between a controller and a controller object is immutable, the controller itself is mutable. Releasing controller locks held by the controller result in mutation of the controller: a change in generation. Thus, the issue of reference sharing and interning of controllers becomes important. The implementation must ensure the following:

1) Within the station containing the controller object, public copies (copies not under the exclusive control of the implementation) of the associated controller, as returned by the controller service, are not permitted.
2) A controller may by passed to other stations, such as during a remote operation; however, the resulting copy must be made immutable. The `releaseLocks()`operatios must be disabled in such cases, throwing a `java.lang.UnsupportedOperationException`.

53.2 Controller In Context

Though controller aspect locking is always based on the incoming controller, the controller within a controller object method synchronized with respect to controllers is always the controller of the controller object, not the controller of the calling thread.

53.3 Releasing Locks Held by a Controller

A controller object may need to cancel resource reservations, such as when the set of resources needed to perform a certain operation changes. In such cases, the controller object releases all locks associated with its controller and reestablishes new locks. To support this kind of change, controller objects will need to remember reservations that it has granted so that it may reaffirm the reservations, which will have been lost when the locks were released. In general, controller objects will also need to maintain this information to support recovery from suspected loss of lock integrity.

Locks are released in a lazy fashion. After the controller `releaseLocks()` method is invoked, the `isRelevent()` method of previously issued locks must return false. The controller aspect locking must proceed in a manner equivalent to the following:

1) The station, as previously described, ensures that the current thread has an associated controller.
2) If the referent object is currently in an owned state, as evidenced by the existence of a lock, the station invokes the `isOwner()` method on the lock,

passing the incoming controller, to determine if the controller owns the current lock.

a. If `isOwner()` returns false, the station must contact the controller service and determine if the old lock is still relevant. To do so, the station invokes the `isRelevent()` method on the lock object. If so, an `ConcurrentControllerException` is thrown, as previously described. If not, the associated lock is replaced by a new lock issued by the incoming controller.

b. If `isOwner()` returns true, the incoming thread is allowed to access the referent object.

3) If the reference is currently in a free state, the station places the object in an owned state and associates a new lock object, provided by the incoming controller, with the object. The incoming thread is then allowed to access the referent object.

In this scenario, remote communications with the controller service happens only when attempting to establish a new lock: a change in controller ownership of the object. This communication may fail if the controller service is unreachable. In such cases, the lock should assume to still be relevant.

54 Control Reservations

Controller locks may be acquired just in time, in the normal course of performing an operation involving synchronized/controller methods, or they may be reserved. Reservations are made by invoking ***reservation operations*** on an object with the reserving controller in context. Reservation operations are of the following form.

```
public void reserve<operation name>( <args> )
    throws javax.sxi.common.ConcurrentControllerException
```

The arguments and operation name are optional. If no operation name is provided, the reservation is assumed to be made for all operations supported by the object. Otherwise, the reservation is assumed to be made for a specific operation or set of operations. The implementations of the reserve operations (***reservation methods***) are generally nested calls to the reserve operations of other object. Note that reservation methods *must* be synchronized with respect to controllers.

Controller objects make a best attempt at reserving resources. Network failures, controller service failures, and other failure scenarios can result in the loss of reservations. Clients and services should be designed in such a way as to tolerate or recover from such reservation failures.

Persistent Objects

55 Specifying Persistent Objects

An object is persistent if it contains the public constant field `ReferentType` of type `String` and value `ReferentType.`PERSISTENT or `ReferentType.`PERSISTENT_LOGIC.

```
public class MyPer implements ReferentType
{
    public static final String ReferentType = PERSISTENT;
}
```

Stations that support persistent objects will provide specialized acceptors, when the referent is persistent, that implement the `PersistentAcceptor` interface. Proxies to persistent referent objects have a `remove( boolean force )` method that invokes the remove method of the `PersistentAcceptor`.

```
package javax.sxi.common;

import java.rmi.NoSuchObjectException;

public interface PersistentAcceptor extends Station
{
```

```
    /**Remove a persistent object from durable storage.
     * This method will fail if any operations are in
     * progress on the referent object, unless force is
     * true. Subsequent operation attempts result in an
     * NoSuchObjectException.
     * @force if true, the persistent referent object
     *     is removed even if there are operations in
     *     progress.
     * @throws NoSuchObjectException If the referent object has
     *     already been removed.
     */
    boolean       remove(boolean force )
       throws NoSuchObjectException;
}
```

If the removal succeeds and the persistent object implements a remove method, then that method is invoked to allow the object to clean up any persistence for which it is responsible. A failure of the remove method will not cause the remove operation to fail. However, stations should log the exception to indicate that some resources may not have been released. Note that unlike `finalize()`, `remove()` is guaranteed to be called when the persistent object is removed. The remove method signature is as follows:

```
    private void remove();
```

56 Kinds of Persistent State

56.1 Existence

The existence state of a persistent object is a durable record that the object element. This state is handled by the infrastructure and is invisible to the persistent object. When a persistent object is remotely instantiated (the only permissible means of instantiation for persistent objects) a record of its existence is noted. A persistent element exists until its persistent image is removed and is not subject to distributed garbage collection.

56.2 Implicit

The implicit state is that portion of a persistent object's state that is handled by the station on behalf of the object. The implicit state of a persistent object is captured by serializing the object directly. Thus, the transitive closure of the non-transient fields of a persistent object comprises its implicit state. The object output stream used for serialization shall be tagged with the `javax.sxi.server.PersistenceStream` interface to provide a means by which an object can discern if it is being serialized for the purpose of persistence.

56.3 Explicit

Persistent objects can also explicitly control state internal or external to the object. To do so, the object must implement the

`net.jini.core.transaction.server.TransactionParticipant` interface. The persistent object will write explicit state during the prepare or commit operations.

57 Reading State

The state of a persistent object is read when the object is activated and when a transaction in which the object is involved is aborted.

57.1 Activation

A persistent object is activated as a side effect of reference faulting while attempting a remote operation on the object as a referent. The object is stored as a serialized byte array and unserialized when activated. If the persistent object needs to acquire other state when activated, it must provide a `readObject()` or `readExternal()` method as provided for in the Java serialization specification.

When persistent objects are activated, they assume the latest deployed versions of the needed classes. This was detailed previously in Class Loading During Activation, page 45.

57.2 Transaction Abort

If a persistent object is involved with a transaction and the transaction is aborted, the object is immediately deactivated. Operations in progress during the abort will continue with the now stale object. Note that methods synchronized with respect to transactions will be blocked during the abort operation. If these operations have not timed out by the time the object is rolled back, they will continue by activating the object with the last committed state. Thus, the transaction abort reduces to a throw away followed by activation with the last committed state.

58 Writing State

The state of a persistent object is written when the object is remotely (using a Proxy) instantiated and when a transaction in which the object is involved is committed.

58.1 Instantiation

The state of the persistent object, when instantiated, is supplied as constructor arguments. Additional state may be acquired by the object in its constructor. The state, including the existence of the object, is not considered durable until the transaction under which the object was instantiated is committed. Constructors of persistent objects are *always* treated as synchronized with respect to transactions. Thus, if the client does not provide a transaction, the station will create and commit a transaction bracketing the constructor invocation. If the constructor throws an exception, the transaction is aborted.

In response to the constructor transaction preparing, the station serializes the persistent object and prepares to store the resulting byte stream. Any errors in serialization cause

the transaction to abort. On commit, the station is responsible for making the serialized state of the object durable.

If the persistent object implements `TransactionParticipant`, then the station will invoke the `prepare()`, `commit()`, and `abort()` operations *after* the station has performed its transaction duties, as previously outlined.

58.2 Transaction Commits

When a transaction is committed in which the persistent object is involved, the object may be serialized and persisted. The pattern follows that of instantiation: serialization during prepare, object transaction operations invoked after station transaction duties, etc.

If the transaction aborts, the station immediately deactivates the object. Subsequent communications with the object as a referent (the only valid way of communicating with the object) results in activation of the previously committed version.

58.3 Dirty Optimization

Stations are only required to persists dirty persistent objects as an optimization that avoids persisting objects that have not been mutated. The station assumes that mutator methods modify state and all other methods do not. To be considered a mutator, the method must follow the JavaBeans setter pattern. If other methods modify state such that the persistent object needs to be persisted, they must call the `javax.sxi.common.PersistentContext.setDirty()` method. The `PersistentContext` is available as the context for all persistent objects.

```
package javax.sxi.common;

/**Specialized Contextual for persistent object.
 */
public class PersistentContext extends Context
{
    /**Informs the station that this object has been
     * mutated.
     * @return boolean true if the object was already
     *  dirty.
     */
    public boolean setDirty();
}
```

58.4 Optimization for Logic Objects

Logic objects are objects that are not directly responsible for any state. The only interesting state, in such cases, is the state of existence. Persistent logic objects can indicate themselves as such as follows.

```
public class MyClass implements ReferentType
{
    public static final String ReferentType =
                                          PERSISTENT_ LOGIC;
}
```

Logic objects will not have their state persisted by the station nor does it make sense for them to implement the `TransactionParticipant` interface, as it will never be called by the station. This is strictly an allowable optimization and not required of either station implementations or logic objects.

59 Access of Persistent Objects Using Proxies

As with all referent objects, persistent object must only be instantiated and accessed through an appropriate Proxy. Only when the object is accessed by proxy is the station able to apply the semantics associated with the aspects and persistence.

60 Concurrent Operations

There may be operations in progress while a transaction is being prepared, committed, or aborted. From the time the prepare operation begins until the commit or abort ends, the station will react as follows to remote operation attempts.

60.1 Operation in Progress on Methods Not Synchronized/Transaction

These methods are assumed not to be involved with state (logic methods) and are allowed to continue uninterrupted.

60.2 Operation in Progress on Methods Synchronized/Transaction

These operations are allowed to complete before the prepare operation is handled by the station.

60.3 Operation Initiated on Methods Not Synchronized/Transaction

These methods are assumed not to be involved with state (logic methods) and are allowed to initiate without interruption.

60.4 Operation Initiated with New Transaction on Methods Synchronized/Transaction

Station throws a `javax.sxi.common.ConcurrentTransactionException` as the persistent object is still considered involved with the transaction in progress.

60.5 Operation Initiated with Old Transaction on Methods Synchronized/Transaction

Station throws an `net.jini.core.transaction.UnknownTransactionException` as the transaction in progress is considered closed to participants after the prepare operation has been initiated.

Registered Dynamic Services

Dynamic services must be registered with the lookup service for the management domain to which they belong. The hosting station handles the registration of the service with the lookup service as well as maintenance (lease renewal and removal) of the lookup service entry on behalf of the service.

60.6 Specifying the Service Entry

Each dynamic service has one single proxy registered with the lookup services serving the management domain to which the service belongs. Unlike static services, the proxy of a dynamic service is required to be a Proxy in the sense of the dynamic services model. The referent of this Proxy is the primordial point object of the service; thus, the primordial point object must be proxied and the Proxy available to the station in the same package as the point object. Each dynamic service must have a single primordial point object that implements the following method.

```
private net.jini.core.entry.Entry[] getLookupEntries();
```

This method provides a list of entries (possibly empty or null) under which the elements service be registered. The set of registration entries is considered immutable for the life of the service. When the primordial point object is remotely instantiated, the only allowable method of instantiation, the station will invoke `getLookupEntries()` after the constructor has completed. This is coincident with the time that the station begins to register the point object, and, therefore, the service itself, with the appropriate lookup services. As lookup services are dynamic in existence and the registration process asynchronous, there is no guarantee as to when the service will be successfully registered with any particular lookup service.

In addition to the entries provide by the service's primordial point object, the station will add an additional `ServiceInfo` entry, if a `ServiceInfo` entry is not already present in the entry list. The `ServiceInfo` entry must be populated using the package information associated with the primordial point object according to the following table.

ServiceInfo Property	**java.lang.Package Property**

manufacturer	ImplementationVendor
model	full class name of primordial point object
name	toString() of primordial point object
serialNumber	0
vendor	ImplementationVendor
version	ImplementationVersion

If the primordial point object provides the `ServiceInfo` itself, it may provide specialized values for model, name, serialNumber and manufacturer. Other fields should be taken from the package level information. Note that this information is specified in the manifest of the JAR file in which the service classes are resources are deployed.

The dynamic service will be registered with all lookups services for a particular management domain. The management domain is determined by the management domain to which the station hosting the service belongs. Thus, all services in a given station belong to the same management domain as the station itself. The station also oversees the maintenance of registrations and re-registrations in the case of lookup service or station restarts such that dynamic services are considered 'good' Jini technology citizens as outlined in the Jini technology specification.

60.7 Leases

Stations are responsible for maintaining the registration leases of all hosted dynamic services, including those that are persistent but not currently active.

60.8 Response to Lease Renewal Failure

Upon failure to renew a registration lease, the station will periodically attempt to reregister the associated service point object with any available lookup services. The reregistration strategy is implementation specific.

60.9 Service IDs

In the case of persistent service (services with a persistent primordial point object), stations shall persist the service ID, issued when the service was first registered with a lookup service, such that the service will always be registered under the same service ID across restarts of both the station and the lookup services.

Internationalization and Localization

Internationalization is the steps taken to make a program easier to localize. Localization is the process of having a program work in terms of the conventions appropriate to a particular locale. One of these conventions is the language appropriate for the locale. In fact, a localized program needs to have changes other than just language: often there must be changes in the recognition of time zone, the formatting of dates, currency, and other similar translations.

Internationalization must be done in a consistent manner throughout a system. To encourage the use of a single standard of internationalization, a method for internationalization and a class for localization are included as part of this specification. The defined method is an extension to the internationalization support provided by the JDK and uses the `java.util.Locale`, `java.util.Properties`, `java.util.ResourceBundle`, and `java.text.MessageFormat` JDK classes, with which the reader should be familiar.

61 Overview

Internationalization is performed by always referring to user viewable messages indirectly through resource bundles. Each class can have one or more associated resource bundles containing lists of key-message pairs, both of which are strings. The key is a simple string and the message is a string suitable for constructing a `java.text.MessageFormat` object. Thus, the message string can contain substitution placeholders. A particular message is specified by providing:

1) a context class (to be used to locate the resource bundle),

2) a message key (to identify a single message within the resource bundle), and

3) a possibly empty list of objects for substitution into the message.

In a distributed environment, the context class and the classes of parameter objects must be internally represented as a class name and code base pair, as is done with `java.rmi.MarshalledObject`, so that localization resources can be network loaded according to RMI network class loading semantics.

The substitution objects can be strings or more complex objects, such as a `java.util.Date` object. In the latter case, the substitution operation, which is done

during localization, performs format conversion as defined by `java.text.MessageFormat`.

The localization process uses the context class, combined with a specified locale, to locate the appropriate resource bundle. Once the bundle is loaded, localization can select the correct message using the message key. This message is converted into a `java.text.MessageFormat` object that can provide the fully localized message given the list of substitution objects.

62 Internationalization

62.1 LocalizableMessage

LocalizableMessage encapsulates the concept of an internationalized message that can be localized.

```
package javax.sxi.util;

import java.io.Serializable;
import java.util.Locale;

/**Encapsulation of a localizable message. Localizable
 * messages should be treated as immutable. To this
 * end, the constructor clones the substitution object
 * array. Callers should ensure that the individual
 * objects of the array are themselves immutable. In
 * addition, it is recommended that these objects be
 * of JDK classes, such as Number and String, which are
 * immutable. An exception is the use of java.util.Date,
 * which is mutable. Such objects should be cloned with
 * the array containing the only reference to the clone.
 */
public final class LocalizableMessage
   implements Serializable, Cloneable
{
```

```
/**Create a localizable message object.
 * @param context The class used as a root in order
 *     to load localization resources. If null, an
 *     IllegalArgumentException will be thrown.
 * @param key The message key to locate an
 *     individual message in a properties file. If
 *     null, an IllegalArgumentException will be
 *     thrown.
 * @param params Parameter (substitution) objects.
 *     may be null. It is recommended that only
 *     java.* class objects be used to avoid the
 *     need to network load other classes in support
 *     of the localization process.
 * @param locale The locale to be considered as the
 *     originating locale. If null, the default
 *     locale will be used. This locale is used to
 *     create the fall back text for this message.
 * @return The newly created message object. The
 *     fall back message will have already been
 *     created.
 */
public    LocalizableMessage(
             Class context,
             String key,
             Serializable[] params,
             Locale locale
          );

/**Get the localized text for this message.
 * If the localization fails (for example if the
 * resources needed to perform localization are
 * currently not available on the network) and
 * useFallback is set to true, then
 * the fall back text is returned. The fall back
 * text was formed when the message was created
 * using the locale provided to the factory
 * create method. If localization fails and
 * useFallback is set to false, an
 * LocalizationError is thrown.
 * @param locale Locale to be used for localization.
 * @param useFallback if true, on a localization
 *     failure, use the fall back text. if false, on
 *     a localization failure throw an
 *     LocalizationException.
 * @throws IllegalArgumentException if locale is
 *     null.
 * @ throwable LocalizationError if localization
 *     fails and useFallback is false.
 * @return Localized text of the message.
 */
public String getLocalizedText(
                    Locale locale,
                    boolean useFallback
                );
```

```
    /**Get the localized text for this message using the
     * default locale as returned by Locale.getDafult().
     * If the localization fails (for example if the
     * resources needed to perform localization are
     * currently not available on the network) and
     * useFallback is set to true, then
     * the fall back text is returned. The fall back
     * text was formed when the message was created
     * using the locale provided to the factory
     * create method. If localization fails and
     * useFallback is set to false, an
     * IllegalArgumentException is thrown.
     * @param locale Locale to be used for localization.
     * @param useFallback if true, on a localization
     *     failure, use the fall back text. if false, on
     *     a localization failure throw an
     *     LocalizationError.
     * @throws IllegalArgumentException if locale is
     *     null.
     * @throws LocalizationError if localization
     *     fails and useFallback is false.
     * @return Localized text of the message.
     */
    public String getLocalizedText( boolean useFallback );
```

```
    /**Get the locale used to create this message. This
     * will also be locale that was used to generate the
     * fall back text.
     * @return The locale used to create this message.
     */
    public Locale getFallbackLocale();

    /** Private localization method. This method must only
     *  be called by getLocalizedText(). This as an
     *  implementation delegation method. If the
     *  localization fails, a LocalizationError is returned.
     *  @param context The class used as a root in order
     *      to load localization resources.  Guarunteed
     *      to not be null.
     *  @param key The message key to locate an
     *      individual message in a properties file.
     *      Guarunteed to not be null.
     *  @param params Parameter (substitution) objects.
     *      May be null, but guarunteed to not contain null
     *      array entries.
     *  @param locale Locale used to perform localization.
     *      Guarunteed to not be null.
     *  @return Returns localized text.
     *  @throws Throws LocalizationError if localization
     *      fails.
     */
    private static String      localize(
                                  Class context,
                                  String key,
                                  Serializable[] params,
                                  Locale locale
                               );

    public static final class LocalizationError
       extends CompositeError
    {
    }
}
```

A `LocalizableMessage` encapsulates a context class, message key, substitution objects (possibly none), fall back locale, and a fall back text. During the localization process, described fully in section 63, a properties files containing the texts for a given locale must be loaded. Resource loading is always relative to a given class: in this case the context class. Thus, the context, locale, and key are used to load and select a single text for the message. Then the parameter objects, if any, are substituted into the text to arrive at a localized text for the message. Note that localization may involve the network loading of property files and classes if they are not available locally.

62.2 Providing Resource Files

The resources for a class of package `a.b.c` are located in the package `a.b.c.resources`. As described by `java.util.ResourceBundle`, the default resource file will have a base name identical to the unqualified class name. Resource files containing messages particular to a locale are named as specified by `java.util.ResourceBundle`. For example, the French resource file for the class `A`

would be `A_fr.properties`, if it is a properties file, and stored in the resource directory below `A.class`. The JDK allows resources to be class files or property files. In either case, the result is a key-value pair in which, for the purposes of localization, both the key (message key) and value (text) must be strings. For simplicity, the examples use property files with the understanding that equivalent behavior can be had with class files.

63 Localization

Localization of a given message happens first when the message is created, to create the fall back text, and subsequently whenever the `getLocalizedText()` is invoked.

63.1 Finding Text

Given a `Locale` and class `A`, a resource bundle is located using the `java.util.ResourceBundle` class. The search for a particular property or class file defining the resource bundle for a given locale is described by the `java.util.ResourceBundle` documentation. If the resource bundle is found and contains the desired message key, the resulting text is used for localization. If not, the search continues up the inheritance tree using a breadth first search with preference given to classes over interfaces at the same depth. No ordering is specified with respect to interfaces at the same depth. The search will not include classes that are rooted at java packages. If no message is found using this search, the fall back text will be used.

63.2 Localization Implementation

Localizable messages must use the localization facilities of the local station as provided by the `javax.sxi.util.LocalizableMessage.localize()` method. This method delegates to an implementation as described in section 14.

64 Serialization of Messages

Messages must be serializable for the purposes of marshaling during remote operations and for persistence of messages. The serialization shall follow RMI marshaling semantics: classes shall be annotated with their code bases. An implementation of `LocalizableMessage` might, for example, encapsulate the context and substitution parameters in a `java.rmi.MarshalledObject` object. Deserialization of the context class and the substitution objects could involve network class loading of the annotated classes: a high risk activity. As the localizable message is intended to be a highly reliable class, it must obey the following rules with respect to serialization failures.

64.1 Failure to Serialize

If any portion of a `LocalizableMessage,` except the fall back text, fails to serialize then the `LocalizableMessage` must recover and still serialize at least the fall back text. Localization attempts on the resulting deserialized message shall return the fall back text.

64.2 Failure to Serialize

If any portion of a `LocalizableMessage,` except the fall back text, fails to serialize then the `LocalizableMessage` must recover and still serialize at least the fall back text. Localization attempts on the resulting deserialized message shall return the fall back text.

64.3 Low Risk Substitution Objects

To reduce network resource loading and, therefore, increase the reliability of localizing messages, it is strongly encourage to use only substitution objects of classes in the java. packages, such as `java.lang.String` and `java.util.Date`.

64.4 Messages as Public Interfaces

Messages issued by a service are part of the public interface of that service. As such, all of the localization resources needed to localize the messages must be include in the dynamic ("-dl") JAR of the deployment group for the service.

Composite Exceptions and Errors

Much like internationalization, exception and error handling benefits from standardization and so are included in this specification as strong recommendations for dynamic service developers. In Java, error conditions are uniformly indicated by throwing throwables, which includes exceptions and errors of various sorts. Checked throwables are those that must be declared. Unchecked throwables are those that need not be declared.

Two distinct problems are being addressed by the proposed throwable extensions: nested throwables and internationalized throwables. At points of abstraction in an object oriented design, often indicated by interfaces, one wishes to decouple the implementation from the abstraction. Abstract throwables, particularly exceptions, must be defined in addition to the interface in order to achieve sufficient decoupling. Indeed the JDK has several examples of this pattern, including `java.lang.reflection.InvocationTargetException` and `java.rmi.RemoteException`. These abstract exceptions each have encapsulated target exceptions. In this specification, the mechanism is unified by providing a standard method of nesting one or more throwables within another throwable.

The messages of the JDK throwables are not internationalized and, therefore, not suitable for user viewing. It is essential that sophisticated users are able to view throwable messages to diagnose the cause of the failure.

The base classes defined to handle these problems are `javax.sxi.util.CompositeException` and `javax.sxi.util.CompositeError`. All throwables that could possibly be viewed by the user or considered abstract, in the sense that they can be thrown in response to another exception, should specialize either `javax.sxi.util.CompositeException` or `javax.sxi.util.CompositeError`.

65 Nested Throwables

An abstract throwable is one that is thrown in response to another thrown throwable. For example, consider a virtual volume component with a method `sizeVolume( long size )` to change the size of a virtual volume. The operation could fail for any number of reasons and many of those reasons would be specific to a particular implementation of the component. Therefore, it would be appropriate to have the method throw an abstract `ResizeFailedException` in response to an error condition during the operation

attempt. If the implementation caught an `IOException`, for example, during the execution of the `sizeVolume()` method, it should create a `ResizeFailedException` with the `IOException` as a nested child exception. The method can throw the `ResizeFailedException` without losing the information contained in the `IOException`. This is the basic nesting pattern.

The nesting of throwables is not necessarily linear. Particularly when alternate strategies and retries are involved in attempting to complete an operation, there can be more than one nested child throwable. Thus, a `javax.sxi.util.CompositeException` or `javax.sxi.util.CompositeError` can actually represent a tree of throwables containing information pertinent to the failure of the attempted operation.

`javax.sxi.util.CompositeException` and `javax.sxi.util.CompositeError` support multiple nested child throwables. One can navigate from the parent throwable to child throwables, but not from child to parent. All throwables are considered immutable objects; therefore, the list of child throwables is established during the instantiation of the parent and cannot be changed.

66 Internationalization and Localization of Throwables

Many exceptions are ultimately destined for informing the user, even if simply because the application has no other idea what to do with them. Exceptions usually carry message information for user viewing, whether in a graphical alert box or on the command line.

`javax.sxi.util.CompositeException` and `javax.sxi.util.CompositeError` require a `javax.sxi.util.LocalizableMessage` object, or the arguments needed to construct a `LocalizableMessage` object, as arguments to all constructors. The localizable message may also be retrieved using the `getMessage()` method.

67 Stack Traces and Throwable Serialization

When a throwable is serialized, such as when thrown during a remote operation, the stack trace is lost, as stack information is considered transient by the JDK. This behavior results in the loss of valuable diagnostic information. To compensate for this shortcoming, `javax.sxi.util.CompositeException` and `javax.sxi.util.CompositeError,` when first serialized, build text versions of the stack traces associated with each nested throwable. Stack trace information is maintained and can be retrieved. Because not all throwables subclass `javax.sxi.util.CompositeException` or `javax.sxi.util.CompositeError`, the root throwable must be responsible for the stack traces of all its descendents, not just for its immediate children.

For remote method calls, the logical stack trace for an exception spans JVMs. To assist in exception diagnosis, Proxy implementations shall append stack trace information for the local JVM when an exception thrown by a remote method is being rethrown in the local JVM.

68 Rules for Handling Throwables

1) Never discard one throwable and throw another throwable. The original throwable can contain valuable information needed to diagnose the problem.
2) Never concatenate messages as a way of nesting throwables. This primitive nesting is not consistent, cannot be reliably traversed, and cannot be localized.
3) Provide as much context information, in the form of localizable messages, as reasonable. Throwing a file permission exception without including the file name, for example, does little to help the user diagnose the problem.

69 Composite Throwable Interface

CompositeException and CompositeError both implement the CompositeThrowable interface. This interface provides operations for getting messages and nested exceptions.

```
package javax.sxi.util;

/**Common abstraction for CompositeException and
 *  CompositeError.
 */
public interface CompositeThrowable
{
    /**Returns a localized description of this
     *  CompositeThrowable using the default locale.
     *  @return Returns the localized message.
     */
    String       getLocalizedMessage();

    /**Returns a localized description of this
     *  CompositeThrowable using the given locale.
     *  @param locale in which to perform the localization.
     *     An IllegalArgumentException is thrown if locale
     *     is null.
     *  @return Returns the localized message.
     *  @throws IllegalArgumentException if locale is null.
     */
    String       getLocalizedMessage( Locale locale );

    /**Returns the array of (causal) nested exceptions
     *  included in
     *  the CompositeThrowable.
     *  @return the array containing the causal
     *     nested exceptions.
     */
    Throwable[]  getNestedExceptions();
}
```

70 Composite Exception Class

```
package javax.sxi.util;

/**Class for composite exceptions.
 *  This class exists to create a uniform method for
 *  handling of exceptions that are due to (multiple)
 *  causes, and to allow for the uniform localization of
 *  the messages associated with those exceptions.
 *  <P>
 * Each composite exception contains a LocalizableMessage
 *  object which encapsulates the localizable exception
 *  message.
 *  <P>
 * Additionally, each composite exception may contain
 *  references to a number of "nested" Throwable's, which
 *  are treated as being the cause of this exception.  To
 *  correctly use this, the causing Throwable's should be
 *  caught (based on tries and retries) and accumulated,
 *  and then included as nested exceptions in a new
 *  exception extended CompositeException.
 */
public class CompositeException extends
   Exception implements CompositeThrowable
{
   /**Construct CompositeException with provided message
    * and nested exception.
    * @param message Informative failure message.
    * @param nestedExceptions Throwables which are a
    *     cause of this exception. May be null. Null
    *     entries in the array are ignored.
    */
    public   CompositeException(
                  LocalizableMessage message,
                  Throwable[] nestedExceptions
             );

   /**Construct CompositeException using using its own
    *  class for the LocalizableMessage context.
    * @param key Identifies message within the resource
    *     bundle.
    * @param params Localization substitution parameters.
    * @param nestedExceptions Throwables which are a
    *     cause of this exception. May be null.  Null
    *     entries in the array are ignored.
    * @throws IllegalArgumentException if messageKey is
    *     null or if messageParams array contains nulls.
    */
    public   CompositeException(
                  String messagekey,
                  Serializable[] messageParams,
                  Throwable[] nestedExceptions
             );

   /**Returns a localized description of this
    *  CompositeException using the default locale.
    * @return Returns the localized message.
    */
    public String      getLocalizedMessage();
```

```
    /**Returns a localized description of this
     *  CompositeException, using the given locale.
     *  @param locale in which to perform the localization.
     *     An IllegalArgumentException is thrown if locale
     *     is null.
     *  @return Returns the localized message.
     *  @throws IllegalArgumentException if locale is null.
     */
     public String      getLocalizedMessage(
                              Locale locale
                         );

    /**Returns the array of (causal) nested exceptions
     *  included in the CompositeException.
     *  @return the array containing the causal
     *     nested exceptions.
     */
     public Throwable[] getNestedExceptions();
}
```

71 Composite Error Class

```
package javax.sxi.util;

/**Class for composite errors.
 *  This class exists to create a uniform method for
 *  handling of exceptions that are due to (multiple)
 *  causes, and to allow for the uniform localization of
 *  the messages associated with those exceptions.
 *  <P>
 * Each composite error contains a LocalizableMessage
 *  object which encapsulates the localizable exception
 *  message.
 *  <P>
 * Additionally, each composite error may contain
 *  references to a number of "nested" Throwable's, which
 *  are treated as being the cause of this error.  To
 *  correctly use this, the causing Throwable's should be
 *  caught (based on tries and retries) and accumulated,
 *  and then included as nested exceptions in a new
 *  exception extended CompositeError.
 */
public class CompositeError extends Error
    implements CompositeThrowable
{
    /**Construct CompositeError with provided message
     * and nested exception.
     * @param message Informative failure message.
     * @param nestedExceptions Throwables which are a
     *     cause of this exception. May be null. Null
     *     entries in the array are ignored.
     */
     public   CompositeError(
                    LocalizableMessage message,
                    Throwable[] nestedExceptions
              );
```

```
    /**Construct CompositeError using using its own class
     * for the LocalizableMessage context.
     * @param key Identifies message within the resource
     *     bundle.
     * @param params Localization substitution parameters.
     * @param nestedExceptions Throwables which are a
     *     cause of this exception. May be null.  Null
     *     entries in the array are ignored.
     * @throws IllegalArgumentException if messageKey is
     *     null or if  messageParams array contains nulls.
     */
    public   CompositeError(
                    String   messagekey,
                    Serializable[] messageParams,
                    Throwable[] nestedExceptions
             );

    /**Returns a localized description of this
     *  CompositeError using the default locale.
     * @return Returns the localized message.
     */
    public String      getLocalizedMessage();

    /**Returns a localized description of this
     *  CompositeError using the given locale.
     * @param locale in which to perform the localization.
     *     An IllegalArgumentException is thrown if locale
     *     is null.
     * @return Returns the localized message.
     * @throws IllegalArgumentException if locale is null.
     */
    public String      getLocalizedMessage( Locale locale );

    /**Returns the array of (causal) nested exceptions
     *  included in the CompositeError.
     * @return the array containing the causal
     *     nested exceptions.
     */
    public Throwable[] getNestedExceptions();
}
```

72 Exception Debugging

To facilitate the debugging of exceptions, this specification defines an abstract static method, `javax.sxi.util.Debug.debugException()`, which can be called when an exception is caught. The specific behavior of this method is left up to the implementation provider, but the method should usually store the exception somewhere external to the JVM such that it can be retrieved and analyzed by a developer at a later occasion. `Debug.debugException()` must never throw a throwable under any condition, return reasonably quickly, and not in any way impair the further functioning of a station.

The information passed to `debugException()` is intended for debugging use only.

For example, an implementation of `Debug.debugException()` might serialize exceptions and stack trace information into a file for later retrieval and viewing by a developer.

```
package javax.sxi.util;

public final class Debug
{
   /**Does something to facilitate debugging of an
    *  exception.
    * @param clue String giving a clue as to what
    *     happened.
    * @param exception Exception that happened.
    */
    public static void     debugException(
                               String clue,
                                   Throwable exception
                               );
}
```

Section 3: Static (Base) Services

Base services are a guaranteed part of the environment in a management domain. The base services include transaction, controller, logging, events, and scheduling. They are available for use by the clients and services belonging to a management domain and do not depend on the dynamic services model. In other words, the services are standalone and good Jini technology citizens in their own right. There must only be one of each type of service available in each management domain. If a given service features replication for the purposes of high availability, the replication is not visible to the service client and the service appears logically as a single service. In particular, the service registers a single service proxy in the lookup services for the domain.

Services must be registered with a populated `ServiceInfo` entry. Services must register their proxies with all lookup services that belong to the "`<management domain name>`" group. To do so, services must continually listen for the arrival of lookup services belonging to the management domain. Within a lookup service, the individual service types are distinguished by interface.

Static Services Model

While it is permissible to directly contact a lookup service and retrieve a proxy for a particular management service, stations are required to provide local convenience access to the base services using the abstract class `javax.sxi.services.ServiceFinder`, as follows.

```
package javax.sxi.server;

import javax.sxi.util.CompositeException;
import javax.sxi.services.controller.ControllerService;
import javax.sxi.services.log.LogService;
import javax.sxi.services.event.EventService;
import javax.sxi.services.scheduling.SchedulingService;
import net.jini.core.transaction.server.*;

/**Convenience access to static (base) services.
 *  Implementations may cache service proxies that
 *  have been retrieved. Implementations may also
 *  place limits on how long they will wait for
 *  a lookup service to respond before failing.
 *  Before providing a service proxy, the implementation
 *  must verify that the service is reachable using
 *  the proxy. If not and the proxy was from a cache,
 *  the cache must be invalidated and the service
 *  proxy refetched from a lookup service. If not and
 *  the service was not cached, the method must throw
 *  a ServiceNotFoundException.
 */
public final abstract class ServiceFinder
{
   public static TransactionManager
      getTransactionService()
         throws ServiceNotFoundException;

   public static ControllerService
      getControllerService()
         throws ServiceNotFoundException;
```

```
    public static LogService
       getLogService()
          throws ServiceNotFoundException;

    public static EventService
       getEventService()
          throws ServiceNotFoundException;

    public static SchedulingService
       getSchedulingService()
          throws ServiceNotFoundException;

    public static TransactionManager
       getTransactionService( String domain )
          throws ServiceNotFoundException;

    public static ControllerService
       getControllerService( String domain )
          throws ServiceNotFoundException;

    public static LogService
       getLogService( String domain )
          throws ServiceNotFoundException;

    public static EventService
       getEventService( String domain )
          throws ServiceNotFoundException;

    public static SchedulingService
       getSchedulingService( String domain )
          throws ServiceNotFoundException;

    public static final class ServiceNotFoundException
       extends CompositeException
    {
    }
}
```

`ServiceFinder` is an interface class that uses implementation delegation; however, only the methods taking a management domain name are delegated directly to the implementation. The methods that do not take a management domain name are delegated to the previous methods while using the "`javax.sxi.domain`" system property to supply the management domain domain. This property is dynamic and must be refetched each time a `get<name>Service()` method is called.

Transaction Service

The well-known transaction service is a Jini technology transaction manager serving a particular management domain.

73 No Transaction Service

If no transaction service is present, transaction activity cannot be initiated, but previously completed transactions are not affected. Sources of activity, principally clients, the event service, and the scheduler service, will need to wait until a transaction service is available before initiating activity. Failure to do so results in thrown exceptions when an attempt is made to create a new transaction, directly or indirectly.

74 Failed Transaction Service

Transactions are not considered long-lived and will be lost if the transaction service fails while a transaction is in progress. A transaction in progress when the transaction manager fails will generally fail when the transaction initiator aborts or commits the transaction. As the transaction initiator does not have knowledge of all the transaction participants, participants should consider verifying that the transaction in which they are participating is still valid if a reasonable length of time, such as five minutes, has passed without a commit or abort. If the transaction is no longer valid, participants should behave as if the transaction had been aborted. As any exceptions encountered during the abort will not be thrown to the transaction initiator, as would normally be the case, the exceptions should usually be logged and possibly result in a notification event.

75 Recovered Transaction Service

A transaction manager is not required to recover any state, other than its service ID, when restarted after failure as all transactions in progress are assumed to have been lost.

Controller Service

The controller service is responsible for issuing controllers to both clients and stations acting on behalf of controller objects. It maintains a centralized view of all the controllers in the system; however, this view is considered slave, not master, state. The master state is maintained internally by the clients and stations. Leases are in place such that when is lease fails to renew, it is an indication that state synchronization may have been lost and the state of the controller service should be rebuilt. The state rebuilding is done by stations informing the controller service about the controllers for which the station is responsible. Client controllers may be lost when the controller service fails or becomes unreachable. Thus, clients may have to restarted if the controller service fails.

Objects never need to contact the controller service or invoke methods on a controller or lock object directly. These duties are handle by the station on behalf of controller objects. Controller objects may cancel locks held by a controller by calling `javax.sxi.common.Context.releaseLocks()`.

76 Controller and Controller Generations

In the course of remotely invoking methods synchronized with respect to controllers, object level locks may be acquire and assigned to the calling controller based on the semantics of the controller aspect. These locks belong forever to a specific controller and generation. A single controller can undergo a change in generation after which it is the same controller, but of a different generation. To effectively release locks held by a controller (`Controller.releaseLocks()`), the generation is changed, a matter of internal bookkeeping. As the previous generation of controller no longer exists, previously issued lock become irrelevant, effectively releasing them to be acquired by another controller.

77 Controller Service Interface

```
package javax.sxi.services.controller;

import java.io.Serializable;
import java.rmi.MarshalledObject;
import java.rmi.RemoteException;
import javax.sxi.util.CompositeException;
import net.jini.core.Lease;
import net.jini.core.lookup.ServiceID;

/**Interface to the controller service. Only station
 *  implementations and clients should contact the
 *  controller service directly. Even then, clients
 *  should only invoke the newClientController()
 *  operation.
 */
public interface ControllerService
{
    //
    // Operations for clients. These are the only
    // operations that may be invoked by a client.
    //

    /**Create a new controller that will live in a
     *  client. The duration of the returned Lease,
     *  embedded in the returned ClientController, shall
     *  be between 1 minute and 5 minutes. Lease
     *  termination will release all locks belonging to
     *  the client controller.
     *  @return ClientController containingg a controller
     *     and a lease of 1 to 5 minute duration to be
     *     be maintained by the station. Cancellation or
     *     expiration of the lease may result in the
     *     controller service releasing the resources,
     *     including controller locks, assigned to this
     *     client
     *  @throws RemoteException Error communicating with
     *     the controller service.
     */
    ClientController newClientController(
                            long leaseDuration
                          )
       throws    RemoteException;

    //
    // Operations for stations. These are the only
    // operations that may be invoked by a station.
    //
```

```
/**Synchronize a station's state (list of controllers)
 * with the controller service. The station state is
 * considered the master and overrides any state the
 * service has for that particular station. Stations
 * are uniquely identified by their service IDs,
 * which are issued when the station registers with a
 * lookup service. Stations MUST call this method
 * before any controllers are created in the station
 * even if the station does not currently have any
 * controllers. Regardless of the requested lease
 * duration, the returned lease shall have a duration
 * between 5 minutes and 30 minutes. Shorter lease
 * durations mean locks are released sooner when the
 * controller holding the locks becomes unreachable.
 * The controller service will need to block certain
 * operations while synchronizing to ensure proper
 * state mirroring.
 * @param controllers list of controllers issued to
 *     the station. Must not be null or contain null
 *     entries.
 * @param stationID Station identifier. Must not
 *     be null.
 * @param leaseDuration A suggested lease duration.
 * @return A lease of 5 to 30 minute duration to be
 *     be maintained by the station. Cancellation or
 *     expiration of the lease may result in the
 *     controller service the releasing resources,
 *     including controller locks, assigned to the
 *     controllers of this station.
 * @throws IllegalArgumentException If controllers
 *     is null or contains a null element, or if
 *     if stationID is null.
 * @throws RemoteException Unable to communicate with
 *     the controller service.
 */
Lease               synchronizeWithStation(
                       Controller[] controllers,
                       ServiceID stationID,
                       long leaseDuration
                    )
   throws    RemoteException;

/**Create a new controller that will live in the
 * station identified by the provided service ID.
 * @param serviceID Station identifier of the station
 *     requesting a new controller. Must not be null.
 * @throws UnknownStationException The service ID
 *     is not known by the service.
 * @throws IllegalArgmentException stationID was null.
 * @throws RemoteException Unable to communicate with
 *     the controller service.
 */
Controller       newController( ServiceID stationID )
   throws    RemoteException,
             UnknownStationException;
```

```
    //
    // Callback operations for controllers and
    // locks. Only controllers and locks are allowed
    // to invoke these callback methods and then only on
    // the controller service that issued the controller
    // or lock.
    //

    /**Delete an existing controller.
     * @param handBack Handback embedded in a controller
     *     issued by this service.
     * @throws UnknownControllerException the hand back
     *     does not correspond to controller known by this
     *     service.
     * @throws RemoteException Unable to communicate with
     *     the controller service.
     */
    void                deleteController(
                                MarshalledObject handBack
                        )
       throws    RemoteException,
                 UnknownControllerException;

    /**Release the locks held by this controller. This
     *  method changes the generation of this controller.
     *  Locks held by the previous generation are no
     *  longer valid. This method returns a new hand back
     *  representing the new generation of the controller.
     *  @param handBack Handback embedded in a controller
     *     issued by this service.
     *  @return New handBack object for the controller.
     *  @throws UnknownControllerException the hand back
     *     does not correspond to controller known by this
     *     service.
     *  @throws RemoteException Unable to communicate with
     *     the controller service.
     */
    MarshalledObject releaseLocks(
                                MarshalledObject handBack
                        )
       throws    RemoteException,
                 UnknownControllerException;

    /**Returns true if the lock ID is still relevant,
     *  false otherwise. A lock ID becomes irrelevant
     *  if the issuing controller was cancelled
     *  or if the issuing controller released
     *  its locks.
     *  @param handBack The owner field of the lock
     *     being verified.
     *  @throws RemoteException Unable to communicate with
     *     the controller service.
     */
    boolean             isRelevant(
                           MarshalledObject handBack
                        )
       throws RemoteException,
                 UnknownControllerException;
```

```
    /**Interface representing objects returned from a
     * controller registration.
     */
    public final static class ClientController
    {
        /**Should only be called by the controller
         * service.
         * @throws IllegalArgumentException If either
         *     argument is null.
         */
        public    ClientController(
                      Controller controller,
                      Lease lease
                  );

        /**Return the lease that a client must
         * maintain to sustain the controller locks
         * held by the client.
         */
        Lease                  getLease();

        /**Return the controller itself.
         */
        Controller          getController();
    };

    public final static class UnknownControllerException
        extends CompositeException
    {
    }

    public final static class UnknownStationException
        extends CompositeException
    {
    }
}
```

78 Controller Interface

```
package javax.sxi.services.controller;

import java.io.Serializable;
import java.rmi.MarshalledObject;
import java.rmi.RemoteException;

/**Interface representing objects returned from a
 * controller registration.
 */
public final static class Controller
{
   /**Opaque closure object that uniquely
    * identifies the controller/generation
    * pair to the controller service.
    */
   private MarshalledObject   handBack;

   /**Remote reference back to the controller service.
    */
   final ControllerService service;

   /**Called only by a controller service.
    * @param service A remote reference (RMI Stub,
    *     proxy, ...) back to the controller service
    *     issuing this controller. service must be
    *     useable across restarts of the controller
    *     service and movement of the service from one
    *     host to another.
    * @param handBack Closure object that uniquely
    *     identifies this controller in its first
    *     generation.
    */
   public    Controller(
                ControllerService service,
                MarshalledObject handBack
             );

   /**Return a proxy to the controller service that
    * owns this controller.
    */
   public ControllerService      getControllerService();

   /**Cancel a controller as irrelevant. The
    * controller will no longer issue locks
    * and all locks issued by the controller
    * become released.
    */
   public void                        delete()
      throws RemoteException;

   /**Invalidate all previously issued locks.
    * This effectively releases locks held by this
    * controller by incrementing the generation of
    * the controller.
    */
   public void                        releaseLocks()
      throws RemoteException;
```

```
    /**Issue a new lock.
     */
    public Lock                       newLock();

    /**Abstract type representing a lock. Locks
     *  may be compared for equality or used as keys
     *  in hash tables and the like.
     */
    public final static class Lock extends Serializable
    {
        final MarshalledObject owner;
        final ControllerService service;

        /**Only called by Controller.
         */
        Lock(
            MarshalledObject owner,
            ControllerService service
        );

        /**Return true if this lock was issued by the
         *  given controller/generation. This is true
         *  iff the owner field of this lock is equal
         *  to the handBack field of the controller.
         */
        public boolean            isOwner(
                                     Controller controller
                                  );

        /**Returns true if the lock is still valid. If
         *  a remote exception is thrown, it is unknown
         *  whether the lock is valid or not. On a remote
         *  exception, the lock should usually be considered
         *  relevant if it was ever known to have been
         *  relevant.
         *  @throws RemoteException Communication error
         *      with the controller service.
         */
        public boolean         isRelevant()
            throws RemoteException;
    }
}
```

79 No Controller Service

If no controller service is present, any attempt to initiate an operation on a component method synchronized with respect to the controller aspect will fail with an exception. Attempting to start, or restart, a station may fail or block depending on the implementation of the station, until the controller service for the management domain is again operational.

80 Failed Controller Service

Unlike transactions, controllers are considered long-lived and are bound to the service with which they are registered. The failure of a particular controller service affects those

components that are locked, for the purposes of controller concurrency control, by a controller registered with the failed service. Operations on these components will not be able to proceed until the failed controller service has recovered.

81 Controller Service Recovery

A controller need only persist its serviceID. Additional persistence capabilities are considered optimizations to reduce the network flooding while stations resynchronize with a restarted controller service. In order to station sufficient time to resynchronize, a restarted controller service should not respond to any requests other than resynchronization for a period of time greater than the longest issued lease duration.

82 Breaking Controller Service Locks

There may be conditions under which it becomes necessary to break controller service locks and controller service implementations may provide administrative interfaces to do so; however, this specification does not standardize administrative interfaces of any kind.

Whether or not an object is acting autonomously (on its own accord or thread), it may wish to log certain decisions that have been made, operations that have been requested, or any other information deemed interesting by the object. Log messages can be very important for auditing, and certain guarantees must be given that a log message is posted and will not be lost. It is also important that the information contained in a log message is internationalized so that the message can be viewed by any particular locale.

It is important to note that the log service proxy performs some important client side processing of arguments. Specifically, the log service proxy decorates log messages with a times stamp, and other information, before passing the information to the remote log service. Communication between the log service proxy and the log service is private to the implementation.

83 Log Service Interfaces

83.1 Log Messages

Log messages contain a localizable message, a category, and possibly a throwable, if the log message is in response to an error condition manifest as a throwable. The category is a dot (".") delimited string that must begin with one of the major categories enumerated in the LogMessage class.

```
package javax.sxi.services.log;

import javax.sxi.util.LocalizableMessage;
import java.io.Serializable;

/**A log message. Log messages are immutable. Thus,
 * mutable objects, such as Date objects, are cloned
 * at the LogMessage interface to preserve immutability
 * of the LogMessage. Throwable are not cloned under
 * the assumption that all throwables are immutable.
 */
public final class LogMessage implements Serializable
{
   //constants for major categories
   static public final String AUDIT     = "audit";
   static public final String DEBUG     = "debug";
   static public final String WARNING   = "warning";
   static public final String INFO      = "info";
   static public final String ERROR     = "error";
   static public final String TRACE     = "trace";

   /**Construct a log message object. The constructor
    * adds the time stamp.
    * @throws IllegalArgumentException If message or
    *     category is null.
    */
   public LogMessage(
                LocalizableMessage message,
                String category,
                Throwable exceptionObject
             );

   /**Returns the localizable message for this
    * log message.
    */
   LocalizableMessage  getMessage();

   /**Returns category of log message, a dot delimited
    * string beginning with one of the major
    * categories.
    */
   String                  getCategory();

   /**Return the throwable object, if one exists
     */
   Throwable           getThrowable();
```

```
    /**Return posting date and time in UTC.
     */
    Date                    getTimeStamp();

    /**Special exception indicating that the throwable
     *  failed to serialize when this log message was
     *  posted.
     */
    public static final class
       SerializationFailureException extends Exception
    {
    }
}
```

83.2 The Log Service Interface

Log service implementations must implement the `javax.sxi.services.log.LogService` interface. The interface includes one method for posting log messages and another for retrieving log messages based on search criteria.

```
package javax.sxi.services.log;

import java.rmi.RemoteException;

public interface LogService
{
    /**Log a message. This method shall not, under
     *  any condition, throw an throwable. The log
     *  service proxy is responsible for dealing with
     *  all error conditions.
     */
    void      log( LogMessage message  );

    /**Perform an synchronous search for log records
     *  matching the provided criteria, which must not be
     *  null. The search can be cancelled by canceling or
     *  not maintaining the Search lease.
     *  @param predicate Predicate to determine interesting
     *  log messages to be matched.
     *  @param batchSize target size of a batch of
     *      delivered
     *  log messages to the iterator.
     *  @param leaseValue leas duration in milliseconds.
     *  @return Search used to enumerate the result set.
     */
    Search    search(
                 Predicate predicate,
                 int batchSize,
                 long leaseValue
              )
       throws RemoteException;
}
```

83.3 Retrieving Log Messages

83.3.1 Predicates

Log messages are logically retrieved, for enumeration or removal, by invoking the `LogService.search()` method. The most significant argument is the predicate object. The predicate, which is passed to the log service by value, selects which log messages are returned as part of the search.

```
package javax.sxi.services.log;

import java.io.Serializable;

/**Unary predicate used to select log messages during
 *  a query operation.
 */
public interface Predicate extends Serializable
{
    /**Execute the predicate. Iff true, the log message
     *  is selected for the search.
     *  @param message Log message to evaluate. May not
     *      be null.
     */
    boolean   execute( LogMessage message );
}
```

Since the predicate object is passed by value, the log service will need to network load the class of the predicate object according to RMI semantics. Some clients may not be able to provide a predicate class through a class server to support such an operation. These clients, and others, can use the well known `javax.sxi.util.LogSearchCriteria` class to create predicate objects that support a fixed selection criteria. Because the `LogSearchCriteria` is supplied as part of the infrastructure, it does not need to be loaded over the network.

```
package javax.sxi.services.log;

import javax.sxi.services.log.LogService;

/**Convenience log searching predicate that searches
 * based on posting date, category, and message.
 */
public final class LogSearchCriteria
   implements Predicate
{
   /**Construct a search criteria object.
    * Dates are compared directly with the posting Date
    * of the log messages without localizing.
    * @param beginDate Beginning date, inclusive, or no
    * beginning date, if null.
    * @param endDate Ending date, inclusive, or no ending
    * date, if null.
    * @param category Dot-delimited category (i.e.,
    * "error.severe.disk_failure"). Most significant
    * word must be one of predefined constants
    * in LogMessage.
    * @param searchLocale Locale in which message are
    * localized before comparison to messagePattern.
    * A null value indicates comparison between the
    * messagePattern and each log message's
    * fall backMessage.
    * This approach is faster, but the messagePattern
    *    must be supplied in the posting locale of the
    *    messages or it must be in a locale independent
    *    form.
    * @param messagePattern Localized pattern to search
    *    for. For example, "disk" would match any log
    *    message whose localized message contains "disk".
    */
   public    LogSearchCriteria(
                Date beginDate,
                Date endDate,
                String category,
                Locale searchLocale,
                String messagePattern
             );
}
```

83.3.2 Searches

A search operation on a log service return a `javax.sxi.services.log.Search` object, which is a kind of iterator. The log service must maintain the results of a particular search, which consumes significant resources. These resources are reserved by the search using the lease returned as part of the search result. If this lease is cancelled or expires, the log service may discard resources associated with search and any further attempts to access the `Search` object may throw a throwable.

```
package javax.sxi.services.log;

import java.rmi.RemoteException;
import java.util.Iterator;
import net.jini.core.lease.Lease;

/**Specialized leased that also supports polling to
 * retrieve log messages and remove them. Search does
 * not support the Iterator.remove() operation.
 */
public interface Search extends Iterator
{
    /**Return the lease used to maintain the
     * resources associated with this search.
     */
    Lease           getLease();

    /**Returns an array (batch) of messages. The target
     * size of the batch was specified when initiated
     * the search.
     * @return Array of log messages (LogMessage[])
     * @throws NoSuchElementException no more
     *     messages available that match the search
     *     predicate or a remote exception during
     *     communication with the log service.
     */
    Object          next();

    /**Strongly typed version of next().
     */
    LogMessage[]    +nextMessageBatch()
       throws RemoteException;

    /**Remove all messages matching this search. If the
     * search has been enumerated, fully or partially,
     * it is guaranteed that only the messages that
     * were enumerated will be removed.
     */
    void            removeAll()
       throws RemoteException;
}
```

83.4 Removing Log Messages

Log messages can be removed by invoking the `removeAll()` operation on a valid `Search` object. If `next()` or `nextMessageBatch()` has never been called on the `Search` object, all log messages matching the search criteria, at some point in time after the search was initiated, shall be removed. Otherwise, only the specific messages which have been enumerated shall be removed.

84 Posting Failure Scenarios

84.1 Posting Reliability

The `LogService.post()` method must not, under any circumstances, throw an exception to the posting client. The log service proxy must handle any failure conditions to the best of its ability. In some failure scenarios, this may imply that log messages are not posted.

84.2 Log Service Unavailable

If the log service is unavailable at the time of posting, the log service proxy may drop the log message. More capable log services may provide proxies that queue postings until such time as the log service again becomes reachable; however, this is not required. Log service unavailability means that the proxy was unable to post the log message to the log service for a reason other than a marshaling failure.

84.3 Marshaling Failure

A log message consists of a localizable message, optional throwable, category (String), and time stamp (Date). The localizable message, category, and time stamp are guaranteed to always serialize. Thus, if one can guarantee that the throwable, if present, will serialize, one can guarantee that the log message as a whole will serialize, avoiding marshaling errors when posting. To this end, the `LogMessage` serialization method must recover from a serialization error of the throwable object by replacing it with a `LogMessage.SerializationFailureException`.

84.4 Log Service Failure While Writing

If the log service terminates while in the process of writing a log message to its persistent store (file, data base, ...), it shall not corrupt any log messages already written nor the durable log as a whole. Only the log message being posted at the time of the termination is allowed to be lost.

Event Service

An event service is a collection of topics to which event sources may post events and from which event subscribers may receive events. Each topic accepts events from event sources and forward them to event subscribers that have indicated an interest in the topic by subscribing to the topic. Each management domain has a single (possibly replicated) centralized event service for the domain. This well-known event service is registered with the lookup services for a particular management domain and implements the `javax.sxi.services.event.EventService` interface.

The topics of the event service are organized into a hierarchy such that each topic has a single parent topic and all topics ultimately descend from the root topic of the service. Each topic has an associated unordered list of observing listeners that have subscribed to the topic.

Each topic, in addition to its unordered event subscribers, may have an optional chain of responsibility. The chain of responsibility (Chain of Responsibility pattern) supports an ordered list of subscribers to support cases in which at most one subscriber should respond to a particular event.

85 Use of the Jini Technology Event Mechanism

Events are based on the Jini event specification, which provides the basic mechanisms for distributed event systems of many types. This specification specializes Jini events for the specific purpose of supporting a transient publish/subscribe event service. By adhering to the Jini specification, general purpose adapters, such as mailboxes and store/forward delegates, that are developed for Jini technology can be used with the event service specified herein.

Events all have a event ID of `javax.sxi.services.event.Event.ID.` The events are further discriminated by the topic ('.' delimited strings), available as a topic property, to which the event was posted.

Event services must provide the minimal Jini specification guarantees with respect to event sequence numbers. Details are available in the Jini event specification. In summary, each event posted to the event service must be assigned a unique and increasing sequence number. The conditions under which this guarantee holds, such as a minimum reboot time, are implementation dependent.

Event Service

86 The Event Object

86.1 Inherited Event Properties

The event object inherits the following event object properties from `net.jini.core.event.RemoteEvent`.

86.1.1 Event ID

The event ID is always set to `javax.sxi.services.event.Event.ID`.

86.1.2 Handback

The handback is a closure object that is provided by a listener and passed back to the listener as part of the event object delivered to that particular listener.

86.1.3 Sequence Number

A number such that each posted event is assigned a unique number that increases monotonically in the order that events are posted to the event service in accordance with the Jini event specification. The sequence number are only guaranteed to be increasing, not necessarily increasing by increments of one.

86.1.4 Source

The event source is of type `java.lang.Object`. Topics to which an event is posted may further constrain the type of the source property as part of the contract between event sources and listeners coupled through the topic. For example, the topic x.y.z may imply that the source is of type Proxy. Verification that the event source is of an acceptable type is not performed by the event service. In the presence of poorly behaved event sources, listeners may receive events with invalid event sources. Note also that if a topic x.y.z specifies a source type of T, then all specialized topics of x.y.z (such as x.y.z.1) must specify a source Ts such that Ts specializes (implements or extends, directly or indirectly) T.

86.2 Declared Event Properties

In addition to the inherited properties, events add the following declared properties.

86.2.1 Topic

The topic property is a '.' delimited `String` specifying the topic to which the event was posted. Note that this is not necessarily the topic from which the event was delivered to a given listener. Thus, a listener registered for topic x may receive events with topics such as x.y, x.y.z, and the like.

86.2.2 Base Event Object

An event object class is any class that, directly or indirectly, extends `javax.sxi.events.Event`. Event classes must be immutable, safely serializable, and conform to JavaBeans coding conventions in terms of exposing properties as standard getter methods.

```
package javax.sxi.services.event;

import net.jini.core.event.RemoteEvent
import java.rmi.MarshalledObject;

/** Abstract event class. Subclasses must override
 *  to add a type safe constructor for the events source.
 *  Each event class implies a type for the source:
 *  Proxy, String, URL, ... All subclass must also ensure that
 *  clone(...) operates correctly.
 */
public abstract class Event extends RemoteEvent
{
   public static final long
      ID = -2479143000061671589L;

   /**Topic to which this event was posted.
    * This may not be the topic from which the event
    * is delivered.
    */
   private final String   topic;

   /**Create an event object with populated source
    * and topic fields. The sequence number is
    * undefined and the event ID will be set to
    * Event.ID.
    */
   protected Event( Object source, String topic );

   /**Topic to which this event was posted.
    * This may not be the topic from which the event
    * is delivered.
    */
   public String    getTopic();

   /**Clone this event and add a handback field. Used
    * only by the event service to create an event for
    * each subscription during delivery.
    */
   public Event     clone( MarshalledObject handback );
}
```

86.3 Root Event Object

```
package javax.sxi.services.event;

/**The root event is the type of event issued by
 * by the root topic. Generally, root events are passed
 * from a source event service to another listener
 * event service, of different management domains.
 * In addition to the contained event, the root event
 * carries a flag indicating whether the event was
 * already handled by a responsible listener. In such
 * cases, the event must not be passed to additional
 * responsible listeners.
 */
public final class RootEvent implements Event
{
   /**True if the contained event has been handled by a
    * responsible listener.
    */
   private final boolean  handled;

   /** Event wrapped by this root event.
    */
   private final Event    containedEvent;

   /**Create a root event object containing a
    * contained event, which may or may not have been
    * handled. The source is set to the name of the
    * management domain containing the event service
    * providing the event. The topic is the root topic
    * (empty String) EventService.ROOT_TOPIC.
    */
   protected RootEvent( Event event, boolean handled );

   /**Get the contained event.
    */
   Event               getContainedEvent();

   /**Return if this event has been handled by a
    * responsible listener.
    */
   public boolean      isHandled();

   /**Clone this event and add a handback field. Used
    * only by the event service to create an event for
    * each subscription during delivery.
    */
   public Event    clone( MarshalledObject handback );
}
```

87 EventService Interface

```
package javax.sxi.services.event;

import net.jini.core.event.RemoteEventListener;
import java.rmi.MarshalledObject;
import java.rmi.RemoteException;

public interface EventService
   implements RemoteEventListener
{
   /** Topic path for root topics
    */
   String ROOT_TOPIC = ("");

   /** Post an event to a topic.
    */
   void      post( Event event )
      throws RemoteException;

   /** Register as a subscriber of this event service.
    *   All events posted to the service will be sent to
    *   the subscriber, regardless of the posting topic,
    *   after they have been fully processed by the local
    *   event service. In particular, the service must
    *   determine whether the event will be handled
    *   locally by a responsible listener. Only a single
    *   listening event service may subscribe to a source
    *   event service. The event sent to the listening
    *   service is of type RootEvent.
    */
   Lease     subscribeToEventService(
                EventService subscriber,
                long leaseLength
             )
      throws RemoteException, TooManyListenersException;

   /** Subscribe as a observing listener to a topic.
    *   All events posted to the topic, or a subtopic,
    *   will be sent to the subscriber.
    */
   Lease     subscribeObserver(
                String topic,
                RemoteEventListener subscriber,
                MarshalledObject handback,
                long leaseLength
             )
      throws RemoteException;
```

```
    /** Register as a responsible listener.
     *  @param index Listener in front of which the new
     *     listener must be inserted. If null, the new
     *     listener is added as the first in the list for
     *     the given topic.
     */
    Lease     subscribeResponsibleBefore(
                 ResponsibleListenerInfo index,
                 RemoteEventListener subscriber,
                 String description,
                 MarshalledObject handback,
                 long leaseLength
              )
       throws RemoteException, UnknownListenerException;

    /** Register as a responsible listener.
     *  @param index Listener after which the new
     *     listener must be inserted. If null, the new
     *     listener is added as the last in the list for
     *     the given topic.
     */
    Lease     subscribeResponsibleAfter(
                 ResponsibleListenerInfo index,
                 RemoteEventListener subscriber,
                 String description,
                 MarshalledObject handback,
                 long leaseLength
              )
       throws RemoteException, UnknownListenerException;

    /** Return a list of responsible listeners for a
     *  given topic.
     */
    ResponsibleListenerInfo[]  listResponsibleListeners(
                                     String topic
                                  )
       throws RemoteException;

    public static final class ResponsibleListenerInfo
    {
       public final String           description;
       public final MarshalledObject cookie;
    }
}
```

88 Topics

The topic space is a tree of individual topics. Each topic is uniquely identified by appending the name of the topic to the name of its parent topic, using a '.' (period) as a delimiter, to a form a topic path. The root topic is special. It is denoted by the empty string and the topic path for a child of the root topic is simple the topic name without prepending a '.'.

Each topic implies a specific class of event that it will accept and deliver. There is no runtime maintenance or checking of this mapping, but is rather part of the contract

between event sources and listeners. The topic operates as an ignorant decoupling between the source and listener without enforcing any aspects of such a contract. Thus, it is possible that listeners receive an event object of an unexpected class. Listeners should be written defensively to ignore such occurrences.

89 Chain of Responsibility

The event service features an implementation of the Chain of Responsibility pattern to support events that warrant at most one response, such as a corrective action, to the event. In addition to the unordered list of listeners, ***observing listeners***, associated with a topic, each topic also has an ordered list of ***responsible listeners***. The topic will deliver a given event to responsible subscribers synchronously in their order of registration. Delivery to responsible subscribers may be done before, after, or concurrently with delivery to observer subscribers. Events are first delivered to the most specialized chain of responsibility and then, if not consumed by a responsible listener, to the chain of responsibility of the next more general topic.

A responsible subscriber is considered to have handled an event if it returns from its post operation without throwing an exception, checked or unchecked. If the exception is of class `EventNotHandledException`, an unchecked exception, then the event service must simply continue with the remaining responsible subscribers. Otherwise, the event service may choose to log the exception as an indication of a faulty listener.

90 Subscribing

Listeners subscribe as observing or responsible listeners to a particular topic. In either case, a `Lease` is returned that must be maintained.

90.1 Observing Listeners

Observing listeners are an unordered set of listeners for a particular topic. Listeners for a topic will also receive events from all descendent topics. There is no order of delivery implied with respect to listeners of a topic versus listeners of a subtopic.

90.2 Responsible Listeners

Responsible listeners are ordered both within a topic and between child and parent topics. Given an event posting to a particular topic T, the ordered set of responsible listeners is formed by taking the responsible listeners of T and appending the responsible listeners of the parent of T and so on recursively. Thus, events are delivered first to the responsible listeners of the most specific topic.

Because responsible listeners are ordered, the subscription methods include variants to control where a responsible listener is placed within the list of responsible listeners.

90.3 Event Service as Listeners

When one event service subscribes as a listener to another event service, special listener semantics apply in order to ensure that only one responsible listener handles a particular event, even across event services. The first of these is that an event service supports only a single listener to the service itself. Secondly, the event must by fully processed in the source event service, with respect to responsible listeners, before passing it to the listening event service. This allows the source event service to inform the listening event service as to whether the event must be propagated to additional responsible listeners.

The event object passed from a source event service to a listening event service is of type `RootEvent`. These objects contain an the posted event as well as a handled flag. An event service receiving a handled event must not pass it on to any responsible listeners under its control. In addition, it must make sure the handled flag is set if it passes the event to another listening event service.

90.4 Listeners as Good Citizens

Events delivered to listeners are done so in threads granted to the listener by the event service. Listeners must only perform simple, low risk operations in the event delivery thread and decouple more complex tasks to a thread owned by the subscriber. The event service can detect a hung or unresponsive listener and cease to deliver events to the listener in order to conserve resources within the event service. A subscription may be cancelled autonomously by the event service only in response to the following conditions.

1) The listener fails to return the event delivery thread within a reasonable time limit, not shorter than 15 seconds, as perceived by the event service.
2) The `Lease` associated with the subscription is not renewed.

The events service must cancel a subscription in response to the following conditions.

1) Event delivery results in a `java.rmi.RemoteException` that is not a `java.rmi.UnexpcetedException`. This would indicate a communication error or that the listener has otherwise become unreachable.
2) Event delivery throws a `java.sxi.services.event.RemoveListenerException.`
3) The `Lease` associated with the subscription is cancelled.

Listeners should only throw `java.sxi.services.event.EventNotHandledException`, `java.sxi.services.event.RemoveListenerException`, and `net.jini.core.event.UnknownEventException`.

A listener can cancel its own subscription explicitly by canceling the associated `Lease`. Regardless of how the subscription is terminated, the event service must ensure that any attempt to renew or otherwise access the associated `Lease` associated with a terminated subscription results in an `UnknownLeaseException`.

90.5 Leases

Leases are used to reserve the resources associated with a subscription. Either the listener (or another party, which has access to the `Lease`) or the event service, may nullify the `Lease`. Leases will also be lost if an event service crashes and is restarted. The inability to maintain a `Lease`, indicated by an `net.jini.core.lease.UnknownLeaseException` thrown during renewal, indicates that the subscription is no longer intact, for whatever reason. In response, the listener can choose to heal the situation by subscribing again and do any work that may be required given the expected loss of events.

91 Event Ordering

For the purposes of event ordering, one can consider observing listeners and responsible listeners independently as there is no ordering specified between the two groups.

Ordering is specified with respect to event postings and subscriptions. An event posting E_2 is said to be after an event posting E_1 if and only if the posting of E_1 returns before the posting of E_2 is initiated.

Each subscription to a topic results in a single, unique subscription. If a listener subscribes to N topics (or one topic N times), the result is N subscriptions. Ordering is specified with respect to a subscription, not a listener. Subscriptions may be unordered (observing listeners) or ordered (responsible listeners). Subscriptions also have relationships based on the hierarchical relationships of their associated topics. A subscription S_1 is superior to a subscription S_2 if and only if the topic associated with S_1 is the parent of the topic associated with S_2.

One says that an event E_1 is delivered before E_2 to a subscription if and only if the delivery method of the associated listener returns from delivering E_1 before the method is invoked to deliver E_2: non-overlapping deliveries.

One says that an event is delivered to a subscription S_1 before a subscription S_2 if and only if the delivery associated with S_1 returns before the delivery associated with S_2 is invoked: non-overlapping deliveries.

91.1 Observing Listeners

The following subscriptions are with respect to observing listeners.

1) For a given event posting, there is no specified order in which the event is delivered to observing listener subscriptions.
2) If E_2 is posted after E_1, then E_2 must be delivered after E_1, with respect to any one observing listener subscription. The effect is as if each subscription had an associated queue to which events were posted synchronously and delivered asynchronously.

91.2 Responsible Listeners

The following subscriptions are with respect to responsible listeners.

1) For a given event posting, if a subscription S_1 is before S_2, both of the same topic, then the event must be delivered to the listener of S_1 before being delivered to the listener of S_2.
2) For a given event posting, if a subscription S_2 is superior to a subscription S_1, then the event must be delivered to the listener of S_1 before being delivered to the listener of S_2.
3) If E_2 is posted after E_1, then E_2 must be delivered after E_1, with respect to any one responsible listener subscription with the exception that this ordering is no longer pertinent if E_1 will never be delivered to the responsible listener (presumably because is was handled).

91.3 Event Service Listeners

1) An event service may have at most one event service listener.
2) For a given event, the event must have been delivered to all of the pertinent responsible listeners or have been handled before delivering the event to the event service listener.
3) If E_2 is posted after E_1, then E_2 must be delivered to the event service listener after E_1.

91.4 Sequence Numbers

1) If E_2 is posted after E_1, then E_2 must be assigned a sequence number that is greater than the sequence number assigned to E_1.

92 Transactions

Listener subscription and event dispatching ignore any transaction context that may exist. If a listener registers interest in a topic within the context of a transaction and the transaction aborts, the listener will not be removed. If an even source delivers an event to a topic, which forwards the events asynchronously to interested listeners, the listeners will not receive the event in the transaction context in which it was sent.

93 Event Service Persistence

An event service persists only its service ID. The event service is otherwise stateless and must not persist subscriptions and associated information. When an event service is shutdown and restarted, all subscription information is lost. The subscriptions are rebuilt over time as listeners respond to failed Leases by re-subscribing and performing any actions associated with the possible loss of events while the event service was inoperative.

94 Management Facades

A management façade, a group of objects acting on behalf of a managed resource, directly handles events on behalf of its resource. This includes subscribing, generating events, translating, filtering, correlating, and posting events from its resource.

94.1 Event Listening

The management façade subscribes for events that are relevant to its resource, and using an appropriate message and its associated protocol, notifies the resource of the event.

94.2 Event Generation

The management façade should generate events and post them to the event service on behalf of its resource when the relevant conditions occur, but for which the resource itself does not generate notifications. For example, the management façade can generate events to indicate:

1) Loss of contact, or restored contact with the resource.
2) Unexpected or incorrect behavior by the resource.
3) Incorrect behavior by other objects interacting with the management façade.
4) Inconsistent internal states inside the management façade itself.
5) Important incidents or changes of state in the managed resource that the management façade detects by polling, rather than by notifications from the resource.

94.3 Event Translation and Posting

The management façade is responsible for receiving notifications from its resource, translating them into management events and posting the event to the appropriate event service.

94.4 Event Filtering

The management façade can choose not to post a notification from its resource to the event service, if the notification is not relevant to other objects in the system. For example, if the notification indicates a condition that can be handled completely by the management façade itself, without intervention by the network operator or by other components, the management façade may choose not to post that notification to the event service.

94.5 Event Correlation

The management façade should correlate event notifications from its resource whenever possible and consolidate multiple related events into a single post to the event service.

Scheduling Service

The scheduling service allows autonomous tasks to be scheduled for performance at some future time or times. Scheduled tasks are persistent and do not have to be rescheduled if the scheduler server terminates and is restarted. The scheduling service is used to schedule large-scale activities, not for small-scale activities or transient tasks, as the scheduling and notification overhead (remote communication, security, transactions, leases, etc.) is substantial. For scheduling small, rapid, or transient tasks, a local facility is recommended.

95 SchedulingService Interface

Scheduling services proxies must implement the `javax.sxi.services.scheduling.SchedulingService` interface. Implementations must persist scheduled tasks to allow continued performance of tasks in case of scheduling service failure and recovery. Scheduled tasks should be performed as close as reasonable to the scheduled time, but timeliness guarantees are not required.

Tasks are scheduled using the `scheduleTask()` method, which returns a `Ticket` object representing the scheduling. If a single task is scheduled multiple times with the scheduling service (i.e., the `scheduleTask()` method is called more than once with the same task), a different `Ticket` is returned for each schedule. A scheduling service does not attempt to detect or disallow the duplicate scheduling of a task.

Scheduled tasks are cancelled with the `cancel()` method of the `Ticket` object. Alternately, a task may throw a net.jini.core.event.`UnknownEventException` exception from within its `notify()` method. When all scheduled performances of a task are completed, the task is automatically cancelled.

```
package javax.sxi.services.scheduling;

import java.io.Serializable;
import java.rmi.MarshalledObject;
import java.rmi.RemoteException;
import java.util.Date;

import javax.sxi.util.LocalizableMessage;

import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;

public interface SchedulingService
{
    // Constants for latePerformancesAllowed parameter

    /**No late performances are allowed.
     * Late performances will be skipped.
     */
    public static final int NONE = 0;

    /**Only one late performance is allowed.
     * Duplicate late performances will be skipped.
     */
    public static final int ONE      = 1;

    /**All late performances are allowed.
     * All scheduled performances will occur.
     */
    public static final int ALL      = Integer.MAX_VALUE;

   /**Schedule task to be performed according to
    *  schedule.
    * @param task Task to be performed.
    * @param description Description describing this
    *     task. May be used in administrative interfaces
    *     to the scheduling service.
    * @param schedule Schedule of task performances.
    * @param latePerformancesPolicy Number of
    *     performances that should be initiated when
    *     performance times have been missed.  NONE, ONE,
    *     or ALL are the only allowable values.
    * @param handback Closure handback object to be
    *     passed back to the Task when performed. May be
    *     null.
    * @return Returns Ticket object for canceling the
    *     task.
    * @throws IllegalArgumentException If an argument
    *     except handback is null or if
    *     latePerformancesPolicy is not one of NONE, ONE,
    *     or ALL.
    */
    Ticket   scheduleTask(
                RemoteEventListener task,
                LocalizableMessage description,
                Schedule schedule,
                int latePerformancesPolicy,
                MarshalledObject handback
             )
       throws RemoteException;
```

```
    /**Cancel a scheduled task. Only called by
     *  the Ticket.cancel() method.
     * @param cookie The cookie of a Ticket issued by this
     *     scheduling service.
     * @throws IllegallArgumentException The cookie is
     *     null or was not issued by this service, the task
     *     was already cancelled, or the cookie is
     *     otherwise invalid.
     */
    void      cancel( MarshalledObject cookie )
       throws RemoteException;

    /**Return from a scheduling operation so that a
     *  scheduled task may be cancelled.
     */
    public static final class Ticket
       implements Serializable
    {
       private final MarshalledObject cookie;

       /**Construct a Ticket. May only be called by a
        *  scheduling service.
        */
       public    Ticket(
                    SchedulingService service,
                    MarshalledObject cookie
                 );

       /**Cancel a previously scheduled task. Invokes
        *  the cancel method of the service with the
        *  provided cookie.
        */
       public void  cancel()
          throws RemoteException;
    }

    /**The schedule interface. Encapsulates a
     *  schedule.
     */
    public interface Schedule extends Serializable
    {
       /**Return the next scheduled performance after the
        * specified time.
        * @param date Date beyond which to search for
        *     the next performance.
        * @return Return Date at which the next
        *     performance is scheduled to occur or null
        *     if no more performances are scheduled.
        * @throws IllegalArgumentException If date is
        *     null.
        */
       Date getNextPerformance( Date date );
    }
}
```

Scheduling Service

96 Ticket

A `Ticket` is evidence of the scheduling of a task. `Ticket` objects are created exclusively by the `SchedulingService.scheduleTask()` method. Future scheduled performances of a task may be cancelled using the `cancel()` operation on `Ticket`. Note that the cancel operation prevents further execution of the task under a particular schedule. It does not abort a currently executing task. In general, it is considered unsafe to interrupt executing tasks and no means are provided for doing so.

97 Tasks

Tasks must implement the `net.jini.core.event.RemoteEventListener` interface and be serializable. The `RemoteEventListener.notify()` method provides the point of initiation for a task. The event object shall be of type`net.jini.core.event.RemoteEvent` and contain the hand back object, if any, provide when the task was scheduled. The event source shall be a String of the form: "Management Scheduling Service". The event sequence number shall increase with each task initiated by the service, but is not required to increment by one. If a task is reinitiated, because of failure of the task or the scheduling service, the event shall retain its original number. The event ID is not used and shall be set to 0L.

Tasks must be serializable to allow the scheduling service to persist the scheduled task. Implementations of the scheduling service must hold a hard reference to the task to ensure that it is not garbage collected before all scheduled performances are completed.

98 Schedules

The `Schedule` interface has three standard implementations: `DateSchedule`, `RepeatedDateSchedule`, and `DurationSchedule`. Other custom implementations are allowable. Implementations must be safely serializable.

```
package javax.sxi.services.schedule;

import java.util.Date;

public final class DateSchedule implements Schedule
{
    /**Construct a Schedule which will perform a task at
     *  the  specified date.
     * @param performanceDates Dates on which the task
     *     should be performed.
     * @throws IllegalArgumentException If array is null
     *     or empty.
     */
    public Schedule( Date[] performanceDates );
}
```

```
package javax.sxi.services.schedule;

import java.util.Date;

public final class DurationSchedule implements Schedule
{
   /**Construct a Schedule that will perform a task on
    * even intervals between a start and end date.
    * @param startDate Date first performance is
    *     scheduled.
    * @param endDate Date after which no performances are
    *  to be scheduled.
    * @param intervalMillis Interval (in milliseconds)
    *     between performances of the task.
    * @throws IllegalArgumentException If dates are null
    *     or if  intervalMillis is negative.
    */
    public   DurationSchedule (
                Date startDate,
                Date endDate,
                long intervalMillis
             );
}
```

```
package javax.sxi.services.schedule;

import java.util.Date;
import java.util.Calendar;

public final class RepeatedDateSchedule
   implements Schedule
{
   /**Convenience constant to indicate every month,
    * day, or hour as an argument to the constructor.
    */
   public final static int[] EVERY = { 0 };
```

```
    /**Constructs a Schedule which will repeatedly perform
     *  a task according to a calendar. Parameters are
     *  similar to UNIX crontab scheduling. Day and month
     *  constants are found in the Calendar class. EVERY
     *  may be used to indicate every month, day, hour,
     *  etc.
     * @param startDate Date before which no performances
     *     are to be scheduled.
     * @param endDate Date after which no performances are
     *     to be scheduled.
     * @param months Months during which task should be
     *      run (i.e., Calendar.OCTOBER).
     * @param daysOfMonth Days of the month performances
     *     are scheduled. May be null or empty if
     *     daysOfWeek is specified.
     * @param daysOfWeek Days of the week performances are
     *  scheduled (i.e., Calendar.FRIDAY).  May be
     *     null or empty if daysOfMonth are specified.
     * @param hours Hours (0-23) at which performances are
     *  scheduled.
     * @param minutes Minutes (0-59) at which performances
     *     are scheduled.
     * @param timeZone The time zone in which performances
     *     are being scheduled.
     * @throws IllegalArgumentException If daysOfMonth
     *     and daysOfWeek are both null or empty arrays, or
     *     if months, hours, or minutes are null or empty
     *     arrays, or if any array values are out of
     *     bounds.
     */
    public   RepeatedDateSchedule (
                Date startTime,
                Date endTime,
                int[] months,
                int[] daysOfMonth,
                int[] daysOfWeek,
                int[] hours,
                int[] minutes,
                TimeZone timeZone
             );
}
```

99 Task Performance

When the scheduled time for a task arrives, the task is performed by calling the `RemoteEventListener.notify()` method. A task may indicate that no future scheduled performances shall be performed by throwing a `net.jini.core.event.UnknownEventException`.

99.1 Thread

Threads are granted from the scheduling service to the task. Implementations of the scheduling service may limit the number of threads available or otherwise limit the resource consumption of the scheduling service.

100 Scheduling Conflicts

The `SchedulingService` will not initiate concurrent performances of a scheduled task. Thus, it is possible that one or more scheduled performance times may pass while the service is waiting for the current performance to complete. When a task completes, if scheduled performances have been missed, the scheduling service determines how many (if any) of the missed performances will be performed. This is determined by the value of the `latePerformancesPolicy` parameter of the `scheduleTask()` method. Acceptable values are `SchedulingService.NONE`, `SchedulingService.ONE`, and `SchedulingService.ALL`.

Concurrent performances of a single task can be achieved by scheduling a task with multiple schedules (i.e., calling `scheduleTask()` multiple times).

101 Protection from Task Exceptions

The scheduling service must protect itself from throwables thrown from task execution. These exceptions shall be caught and logged to the log service, but not propagated further.

102 Scheduling Service Failure

If a scheduling service fails and is recovered, it must not lose scheduled tasks. Thus, `SchedulingService` implementations are responsible for persisting scheduled tasks. Remote tasks in progress when the service fails will continue to run except that they will be unable to renew their leases. If detected, the tasks should consider this an indication to abort. When the scheduling service recovers, the running tasks will be considered incomplete and will be restarted.

It is possible, depending on the timing of a task and the failure of the service, that a task is executed more than once.

It is also possible that copies of a task can be executed concurrently. For example, if a task is running when the service fails, but the task doesn't notice (failure to renew the lease) the failure until some time after the service has recovered. The service, on recovery, will attempt to restart the task, which could result in two copies of the task running concurrently. Tasks, which are not idempotent, may protect themselves against multiple execution runs using means specific to the task. None-the-less, the scheduling service should minimize the possibility by, on recovery, waiting T seconds, where T is twice the longest outstanding task lease duration, before restarting tasks. This delay allows orphan tasks a reasonable chance to detect the failure of the service and react accordingly. To be reasonable, the lease duration should be at least 60 seconds and as high as 300 seconds.

Glossary

A

access control — The security control of a particular thread of execution to a protected resource.

acceptor stubs — Remote references to corresponding acceptors. The acceptor stub/acceptor pair forms the RMI based outer interfaces to a referent object.

appliance — A managed resource with an embedded station capable of hosting dynamic services.

auditing — The durable recording of the performance of certain operations, such as authentication success or failure, for the purposes of respective analysis, often in response to a suspected security breach.

authentication — A classification of security constraint that verifies that the operation is executed on behalf of a certain Principal.

C

client — A client is an external source of activity: external in the sense of being outside of the specified system whether it be a federation of station or a middle tier. The term "client" is sensitive the context in which it being used. For example, consider two-object communicating peer-to-peer. In the context of a single communication, the object initiating the communication (source of activity) is the client while the other is the server. In the context of another communication, the roles may be reversed. Thus, one cannot label a particular entity as a client (or server) with specifying the context in which the labeling applies, such as a particular communication.

Common Information Model (CIM) — A specification that is a description of an object model and of a language in which to describe the classes and the instances of objects of that model.

Common Information Model Schema — A set of standardized default objects and associations for representing computing systems.

confidentiality — The protection, or desire for protection, of information as to be incomprehensible by unauthorized parties.

Glossary

F

Federation of Stations — The set of authenticated JVMs, usually stations that are considered completely secure.

I

implementation delegation — A technique used to abstract static methods when separating implementation from specification. Object methods may be abstracted for this purpose using interfaces. However, constructors and class methods require some form of implementation delegation.

Interdomain federation — A union of shared and private management servers within a single domain.

integrity — The protection, or desire for protection, of information as to be unalterable, without detection, by unauthorized parties.

Intrinsic class servers — Small HTTP class servers, which are embedded in clients or stations, for the purpose of support RMI network class loading.

J

JAAS — Java Authentication and Authorization Service

L

logic method — A method that is not directly responsible for state - stateless.

M

management façade (MF) — A dynamic service that provides access to a managed resource.

management server — A station capable of supporting dynamic services.

management server federation — The union of shared and private management servers with a single management domain. The domain maps to the Jini technology group with which the servers are registered. From a practical standpoint, a federation must generally be contained within a Local Area Network (LAN). Thus, members of the federation are not expected to be separated by unreliable networks or such constructs as firewalls.

O

observer subscribers — Topic subscribers that are not "responsible subscribers". Observer subscribers may receive event notification but cannot consume an event.

P

Principal — A JAAS and Java concept that can be thought of as one possible name for a subject.

private management servers — A class of management server that is embedded in storage appliances.

proxy — A remote reference to a referent. The referent can be an object or class.

Proxy binding — The process of associating a Proxy with its acceptor, whether initially or for refreshing.

R

referents — The object or class to which a particular Proxy refers.

replication group — A group of stations that are considered to be a single logical entity but replicated for the purposes of redundancy. The members of a replication group, therefore, must be interchangeable.

responsible subscriber — A subscriber that can consume (handle) and event and prevent propagation of the event to other responsible subscribers.

Roles — A standard class of Principal in the security model.

S

security domain — A realm of trust against which Subjects are authorized and Roles defined.

Security service — A server that performs secure authentication and user/role management.

shared management server — The server (or replicated set of servers) that belongs to and represents an entire management domain.

station — A JVM enabled to support dynamic services. Stations are themselves Jini technology services.

station proxy — A proxy, in the Jini technology sense, that refers to a station.

subject — A JAAS concept that represents the source of an operation request, such that a person or service.

sub topics — Topics that specialize, either directly or indirectly, a base topic.

Index

D

E

F

H

I

J

L

M

O

P

R

S

T

V