SA

Jini™ Technology Surrogate Architecture Specification

SA.1 Introduction

In order for a hardware or software component to join in a network of Jini™ technology-enabled services (*Jini network*), it must satisfy several critical requirements: it must be able to participate in the Jini discovery and join protocols, and it must be able to download and execute classes written in the Java™ programming language. In addition, it may need the ability to export classes written in the Java programming language so that they are available for downloading to a remote entity. For many hardware or software components, these requirements are not difficult to meet; however, there is a category of components that, for one reason or another, cannot satisfy one or more of the requirements and therefore cannot participate directly in a Jini network. The *Jini™ Technology Surrogate Architecture Specification* addresses this problem by defining a means by which these components, with the aid of a third party, can participate in a Jini network while still maintaining the plug-and-work model of Jini technology.

The common attribute of the hardware or software components targeted by the surrogate architecture is the inability to download code, because of either computational resource or network connectivity limitations.

SA.1.1 Requirements

The following requirements for the surrogate architecture can be stated:

- *Device Type Independence* - The surrogate architecture must be able to support a wide range of hardware and software components with different capabilities.
- *Network Type Independence* - The surrogate architecture must be able to accommodate a diverse range of connectivity technologies. Network type independence includes support for different protocols simultaneously on the same physical transport.
- *Preserve Plug-and-Work* - The surrogate architecture must preserve the plug-and-work model of Jini technology. The Jini architecture includes the concepts of discovery, code downloading, and leasing of distributed resources.

SA.1.2 Definitions

In this specification, the following terms are defined to mean:

- The term *device* refers to a *hardware* or *software* component that is not capable of directly participating in a Jini network.
- A *surrogate* is an object that represents a device. The implementation of the object may be a collection of Java software components and other resources.
- A *host-capable machine* is a system that allows the downloading of code, can run a surrogate, is part of a Jini network, and is accessible to the entity offering the surrogate.
- The *surrogate host* is a framework that resides on the host-capable machine and provides a Java application environment for executing the components of the surrogate architecture. In addition to providing computational resources, an execution environment, and life-cycle management, the surrogate host may also provide other host resources to assist the components in the architecture.
- An *export server* is a component or set of components, that works with, or is part of, the surrogate host that provides the means by which surrogate resources are exported so that they are available for downloading to remote entities. The export of resources, particularly class files, are necessary for many Jini technology operations.
- An *interconnect* is the logical and physical connection between the surrogate host and a device. There may be more than one interconnect defined for a physical connection. It is also possible for the interconnect to be on the same physical connection that forms the Jini network.

- A *interconnect protocol* includes the interconnect-specific mechanisms for discovery, retrieval of the surrogate, and liveness.

SA.1.3 Overview

The first assumption of the surrogate architecture is that there exists a host-capable machine that is connected to both the interconnect and the Jini network. By definition, the host-capable machine has the computational resources to execute code written in the Java programming language on behalf of the device, and can provide the necessary resources that such code might need.

The following text describes, in general terms, the components of the surrogate architecture, the relationships between those components, and the mechanism for running a surrogate. In this description, the interconnect and Jini network are described as two separate paths to the host-capable machine. They could, however, be the same path.

The initial state illustrated in figure 1.1, shows a surrogate host on a host-capable machine. The surrogate host monitors the interconnect. It is possible that a single surrogate host may monitor more than one interconnect.

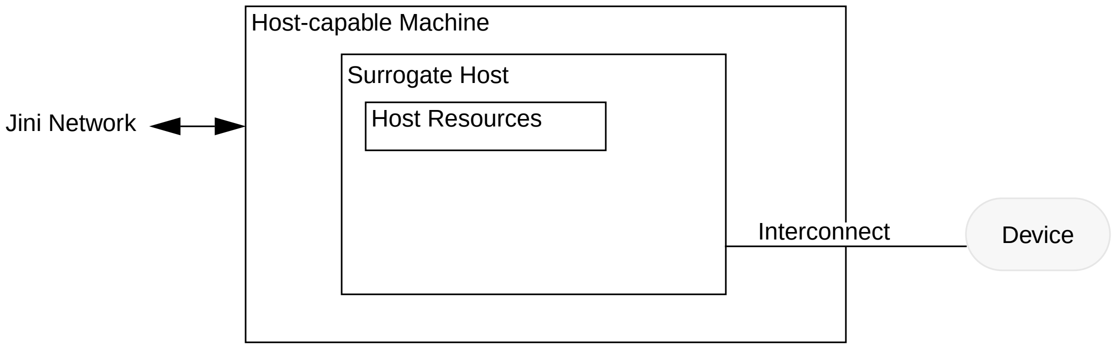

FIGURE 1.1: ***Surrogate Architecture Components***

SA.1.3.1 Discovery and Surrogate Loading

The surrogate host is responsible for implementing the interconnect protocol for a specific interconnect. The first part of the interconnect protocol that comes into play is *discovery*[1]. Discovery, in this context, is the protocol that is used by the device and surrogate host to find each other. A particular discovery protocol is interconnect specific and may be very different from any other interconnect's discovery protocol, depending on the capabilities of the interconnect and the device. A likely scenario would be for the device to broadcast a request for a surrogate host over an interconnect. The surrogate host would respond letting the device know that there is a surrogate host available. If the device or interconnect does not support device-initiated discovery, it might be necessary for the surrogate host to detect the arrival of the device on the interconnect.

Once discovery has been performed, the device's surrogate must be *retrieved*. Again, depending on the interconnect and device's capabilities this operation may be a *push* (the device uploads the surrogate to the surrogate host) or *pull* (the surrogate host must extract, or download, the surrogate from the device). It is also possible for the device to specify that the surrogate be retrieved from a location other than the device.

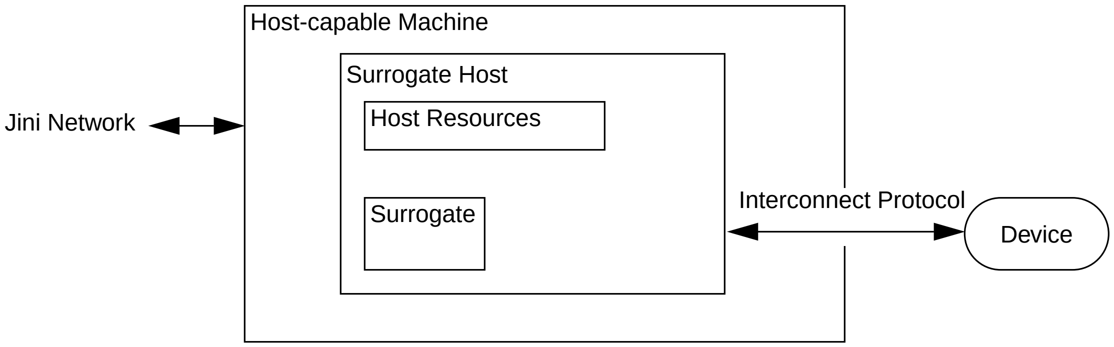

FIGURE 1.2: ***Discovery and Surrogate Download***

SA.1.3.2 Surrogate Execution

The surrogate is *activated* by the surrogate host and may use the resources provided by the surrogate host. Examples of these resources might be an HTTP server or the Jini technology classes.

Once the surrogate has been activated, it may perform any task necessary on behalf of the device, including accessing the Jini network. The surrogate may also

[1] Note that the term *discovery*, in this context, does not refer to the Jini lookup discovery protocol. When this document must refer to the Jini lookup discovery protocol it always uses the terms Jini discovery or Jini discovery protocol.

communicate back to the device, using any (possibly private) protocol that is appropriate.

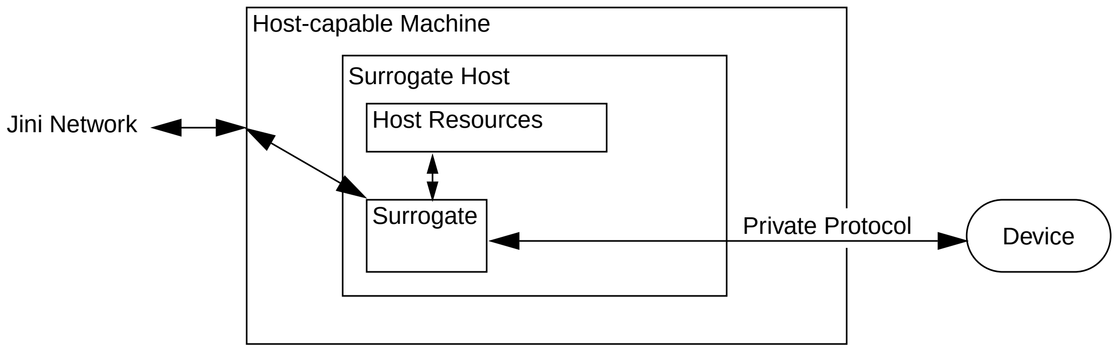

FIGURE 1.3: ***Surrogate Execution***

SA.1.3.3 Liveness

Once a surrogate is loaded and activated, either the surrogate or the surrogate host monitors the device for *liveness*. Liveness means that a usable communication path exists between the surrogate and the device that it represents and that both entities are active. Loss of the communication path could be the result of either normal or error conditions in the surrogate host, the surrogate, the interconnect, or the device.

If the device is no longer reachable, because of interconnect or device failure, or because the device has been disconnected or shut down, the surrogate that it represents must be *deactivated* so that the resources allocated to it by the surrogate host can be reclaimed. Also, the surrogate should release any remote resources that it holds, such as lookup service registrations.

The device must be able to determine that it is no longer in contact with the surrogate, possibly because of an interconnect failure, a fault in the surrogate, or a surrogate host failure or shut down. In any one of these events, the device should resume discovery (if it supports device-initiated discovery) or otherwise prepare to upload a new surrogate.

SA.1.4 Specifications

The complete specification of a surrogate architecture for a particular interconnect requires the following two documents:

- the *Jini™ Technology Surrogate Architecture Specification* (this document) and,
- an interconnect specification for the target interconnect.

The *interconnect specification* describes the interconnect protocol between the device and the surrogate host as well as interconnect-specific additions to the surrogate programming model. The requirements of an interconnect specification are enumerated in Section SA.4, “Interconnect Specification”.

All compliant implementations of the surrogate architecture must satisfy the requirements described in this specification as well as one or more interconnect specifications.

SA.2Jini Technology-Enabled Surrogate

A surrogate is a Java language object that represents a device and performs actions on behalf of the device. A surrogate may be retrieved from a device in an interconnect-specific manner or from some other appropriate source specified by the device, and loaded into the surrogate host for execution. This section describes the surrogate host execution environment and, therefore, the surrogate programming model.

The *surrogate programming model* refers to the programming model available to surrogate developers. This programming model is defined by:

- The Java 2 platform runtime environment
- The APIs as specified by the Jini specifications (*Jini APIs*)
- The export server
- The surrogate APIs
- The packaging of the surrogate
- Security

The surrogate programming model may be extended by an interconnect specification, as described in Section SA.4.2, “Interconnect-Specific Programming Model”.

SA.2.1 Java 2 Platform Runtime Environment

The surrogate host must provide the surrogate with a Java 2 platform that can support the Jini APIs defined in the next section. The minimal Java 2 platform is the Java 2 Standard Edition, v1.2.2 or greater.

Each surrogate is given its own thread that may be in its own thread group. Any method call made to the surrogate must be in its own thread. The runtime environment can be either a separate thread within the surrogate host’s Java virtual machine (JVM) or a thread in a separate JVM. This choice is dependent on the implementation of the surrogate host. Therefore, the surrogate developer should not assume or depend upon a particular implementation.

Each surrogate is instantiated in its own class loader. The implementation of the Jini APIs, the surrogate APIs (described in the following sections), and any interconnect specific APIs (see Section SA.4.2, “Interconnect-Specific Programming Model”) must be available through this class loader.

In order to maintain RMI call semantics it may be necessary for the surrogate host to set the context class loader to be the surrogate’s class loader. The context class loader is a thread local variable, therefore all threads created for making method calls to the surrogate must have the context class loader set.

SA.2.2 Jini APIs

The surrogate host must provide implementations of the Jini APIs as part of the surrogate execution environment to decrease the storage burden on the device and to optimize the downloading of the surrogate. This requirement also enables the same surrogate to run on different platforms, each with a different implementation of Jini technology. The set of Jini API implementations the surrogate host must provide are those defined by version 1.1 or greater of the following specifications:

- *Jini^{TM} Technology Core Platform Specification*
- *Jini^{TM} Discovery Utilities Specification*
- *Jini^{TM} Entry Utilities Specification*
- *Jini^{TM} Lease Utilities Specification*
- *Jini^{TM} Join Utilities Specification*
- *Jini^{TM} Service Discovery Utilities Specification*
- *Jini^{TM} Lookup Discovery Service*
- *Jini^{TM} Lookup Attribute Schema Specification*
- *Jini^{TM} Lease Renewal Service Specification*
- *Jini^{TM} Event Mailbox Service Specification*
- *JavaSpaces^{TM} Service Specification*

Implementations of the Jini APIs must not allow a surrogate to affect the state seen by other surrogates.

Before a surrogate is loaded, the Jini API classes must be available through the surrogate’s class loader. The Jini API classes must be annotated with the same codebase as the surrogate.

SA.2.3 Export Server

The surrogate host's export server is the means by which surrogate resources are exported so that they are available for downloading to remote entities (through the surrogate's codebase annotation). The export server provides two functions: it exports resources and it provides Universal Resource Locators (URLs) for each exported resource.

The surrogate specifies which resources are to be exported through the manifest header described in Section SA.2.5.2, "`Surrogate-Codebase` Header". If the `Surrogate-Codebase` header is present, the URLs provided for the specified resources are used to set the surrogate's codebase annotation. Because of default security behavior of RMI-downloaded code, the codebase annotation provided by the export server (the URLs) must reference the same host machine that the surrogate, is running on. The URLs must also be usable in a distributed system.

Each unique resource that is exported must be given a unique URL. For a given implementation of a surrogate host this URL must be unique across surrogates and unique across time. This is because a remote client can hold a reference to a URL long after a surrogate host has been shut down and restarted.

The exact form of the generated URL is not mandated in this specification and is dependent on the implementation and target deployment of the export server. The surrogate developer must not assume or depend on any particular format of the generated URL.

If a surrogate is deactivated, any resources being exported on its behalf must no longer be downloadable.

If there is a fault in the export server that renders it incapable of fulfilling export requests, the host must deactivate all surrogates that have a dependency on the export server.

SA.2.4 Surrogate APIs

The activating and deactivating of the surrogate by the host is controlled by a set of APIs called the *surrogate APIs*. These APIs also provide a surrogate access to its environment. The next sections describe those interfaces and their behaviors.

The `Surrogate` interface and the `GetCodebase` interface are to be implemented by the surrogate object. Both of these interfaces define methods that the surrogate host calls to control the surrogate and setup its environment. The `HostContext` interface is implemented by an object provided by the surrogate host and passed to the surrogate. This object provides the surrogate access to the environment in which the surrogate executes. The use of the `Surrogate`, `GetCodebase`, and `HostContext` interfaces are interconnect independent.

The KeepAliveManagement and KeepAliveHandler interfaces provide facilities to help a surrogate manage its liveness obligations. The KeepAliveManagement interface is implemented by an object provided by the surrogate host and passed to the surrogate. The KeepAliveHandler is implemented by an object provided by the surrogate and passed to the surrogate host. These two interfaces are interconnect specific in that their use depends on the interconnect that the surrogate is being written for.

SA.2.4.1 The Surrogate Interface

The net.jini.surrogate.Surrogate interface defines the activate and deactivate methods, used by the surrogate host to control the execution of the surrogate. This interface also provides the surrogate access to its execution environment. The surrogate class must implement the Surrogate interface.

The surrogate host may create an instance of the surrogate as needed. For any instance of the surrogate, the activate method is only called once. If the activate method executes successfully, the surrogate host makes its best effort to invoke that same instance's deactivate method when the surrogate is to be deactivated. The deactivate method is only called once for any instance of a surrogate.

The class implementing the Surrogate interface must be a public class and must have a public constructor that takes no parameters, so that a surrogate object can be created by Class.newInstance().

```
package net.jini.surrogate;

public interface Surrogate {
    void activate(HostContext hostContext,
                  Object context)
        throws Exception;

    void deactivate();
}
```

The Semantics

- The activate method is called by the surrogate host to activate the surrogate. Surrogate developers should use this method to allocate resources and start any threads that the surrogate needs.

  The activate method is called once by the surrogate host to activate surrogate. This call must return in a timely manner. If the activate method throws an exception, the surrogate host calls `deactivate` and discards the surrogate.

The `hostContext` parameter provides access to the execution context of the surrogate being activated. This parameter implements the `HostContext` interface (see Section SA.2.4.3, “The `HostContext` Interface”).

The surrogate host must provide a non-`null` `hostContext` parameter. If `null` is passed in, the surrogate should throw an appropriate run time exception.

The `context` parameter provides access to surrogate-specific context of the surrogate being activated. The type of this object is dependent on the type of interconnect utilized by the associated device. The type for a given interconnect would be defined by the specification for that interconnect (see Section SA.4.2, “Interconnect-Specific Programming Model”).

Depending on the interconnect, the `context` object might also implement the `KeepAliveManagement` interface (see Section SA.2.4.7, “The `KeepAliveManagement` Interface”).

- The `deactivate` method is called by the surrogate host to deactivate the surrogate. In general, this method should undo the work done by the `activate` method. When the `deactivate` method returns, the surrogate must have no active thread. This method must complete and return to its caller in a timely manner.

  The `deactivate` method is called once by the surrogate host to deactivate the surrogate.

  After the `deactivate` method returns, the states of the `hostContext` and `context` objects, originally passed to the `activate` method, are undefined, meaning that future calls to methods on those objects will produce undefined results or exceptions. In addition, the surrogate host is free to destroy the surrogate execution environment, including the destruction of any surrogate threads and the JVM that the surrogate is running in.

SA.2.4.2 The `GetCodebase` Interface

Each surrogate is assigned a codebase annotation. The `net.jini.surrogate.GetCodebase` interface provides a method that allows the surrogate to set its codebase.

The `GetCodebase` interface is optionally implemented by the surrogate class (the same class implementing the `Surrogate` interface). Implementing this interface indicates that the surrogate is specifying its codebase. If the surrogate implements this interface, the surrogate host invokes the `getCodebase` method and uses the returned URLs to set the surrogate's codebase before the surrogate's `activate` method is invoked.

If a `Surrogate-Codebase` header is present in the manifest of the surrogate JAR file (see Section SA.2.5.2, "`Surrogate-Codebase` Header") and the surrogate implements the `GetCodebase` interface, the surrogate host uses the return value of the `getCodebase` method to set the surrogate's codebase annotation.

```
package net.jini.surrogate;

public interface GetCodebase {

    java.net.URL[] getCodebase(HostContext hostContext,
                               Object context)

        throws Exception;
}
```

The Semantics

- The `getCodebase` method returns an array of `java.net.URL`s that are used as the codebase of the surrogate.

  The `getCodebase` method is called once per surrogate instance by the surrogate host and must return in a timely manner. This method is invoked before the `activate` method is called.

  If the `getCodebase` method throws an exception, the surrogate host assumes that the surrogate cannot be activated. In this event, the `activate` method is not invoked and the surrogate is discarded.

  The `hostContext` parameter provides access to the execution context of the surrogate. This parameter is provided by the surrogate host and implements the `HostContext` interface (see Section SA.2.4.3, "The `HostContext` Interface").

  The surrogate host must provide a non-`null` `hostContext` parameter. If `null` is passed in, the surrogate should throw an appropriate run time exception.

  The `context` parameter provides access to surrogate-specific context of the surrogate. The type of this object is dependent on the type of interconnect utilized by the associated device. The type for a given interconnect would be defined by the specification for that interconnect (see Section SA.4.2, "Interconnect-Specific Programming Model").

SA.2.4.3 The HostContext Interface

The net.jini.surrogate.HostContext interface provides methods to access the execution environment provided by the surrogate host. An object that implements HostContext is passed as a parameter to the surrogate's activate method and to the getCodebase method of the GetCodebase interface.

```
package net.jini.surrogate;

import net.jini.discovery.DiscoveryManagement;

public interface HostContext {
    DiscoveryManagement getDiscoveryManager();

    void cancelActivation();

    SurrogateController newSurrogate(InputStream surrogate,
                                     Object context,
                                     DeactivationListener listener)
        throws SurrogateCreationException;
}
```

The Semantics

- The getDiscoveryManager method returns an object that implements the net.jini.discovery.DiscoveryManagement interface. This object defines the Jini discovery management policy for this surrogate (see the DiscoveryManagement interface defined in the *Jini^{TM} Service Discovery Utilities Specification*). Multiple calls to the getDiscoveryManager method return the same instance of the discovery management object.
- The cancelActivation method informs the surrogate host that the surrogate must be deactivated. An active surrogate calls this method at anytime to request that it be removed from the surrogate host. When the cancelActivation method is called, the surrogate host marks the surrogate for deactivation and returns. The surrogate host then asynchronously calls the surrogate's deactivate method.

  Once cancelActivation is called, any additional calls will have no effect.

  If the surrogate is to remove itself from the surrogate host while still in the getCodebase method of GetCodebase, it should throw an exception instead of calling cancelActivation. If cancelActivation is called before the `activate` is invoked, the surrogate is discarded without its `deactivate` method being called.

- The `newSurrogate` method requests that the surrogate host load and activate a new surrogate. This method returns after the new surrogate's `activate` method returns successfully. The new surrogate conforms to the contract defined by this specification, with the following two exceptions:

    - The new surrogate is guaranteed to be activated in the same JVM as parent surrogate. The *parent surrogate* is defined to be the caller of the `newSurrogate` method. The new *child* surrogate will have the same security permissions as the parent surrogate. Furthermore, the parent of the class loader created for the child surrogate is set by the host to be the class loader of the parent surrogate. This deviates from the execution environment described in Section SA.2.1, "Java 2 Platform Runtime Environment".
    - The type of the `context` parameter of the `activate` and `getCodebase` methods of the child surrogate is determined by the caller of the `newSurrogate` method. This deviates from `activate` and `getCodebase` method contracts in Section SA.2.4.1, "The `Surrogate` Interface" and Section SA.2.4.2, "The `GetCodebase` Interface".

  The `surrogate` parameter is an input stream that is interpreted to be a surrogate JAR file as described in section Section SA.2.5, "Surrogate Packaging".

  The `context` parameter is passed to the child surrogate as the `context` parameter in the `activate` and `getCodebase` methods (see Section SA.2.4.1, "The `Surrogate` Interface" and Section SA.2.4.2, "The `GetCodebase` Interface"). The context parameter may be `null`. The host does not access or otherwise interpret this parameter.

  The `listener` parameter is an object that is called when the newly created surrogate's `deactivate` method is called (see Section SA.2.4.4, "The `DeactivationListener` Interface"). The value of `null` is passed in if no listener is required.

  An object implementing the `SurrogateController` interface is returned if the child surrogate is successfully activated (see Section SA.2.4.5, "The `SurrogateController` Interface").

  A `SurrogateCreationException` is thrown if any error occurs during the loading or activation of the surrogate (see Section SA.2.4.6, "The `SurrogateCreationException` Class").

If the surrogate host invokes the parent surrogate's `deactivate` method, the surrogate host must first call the child surrogate's `deactivate` method.

Like the Jini classes provided by the host, any classes exported by the child surrogate that are present in the parent surrogate must have their codebase annotation set to the new surrogate's codebase annotation.

If the surrogate has been deactivated because the surrogate host invoked the surrogate's `deactivate` method, the state of the `hostContext` object is undefined.

SA.2.4.4 The `DeactivationListener` Interface

The `net.jini.surrogate.DeactivationListener` interface allows a parent surrogate to be notified when its child surrogate is deactivated.

```
package net.jini.surrogate

public interface DeactivationListener {

    public void deactivated();
}
```

The Semantics

- The `deactivated` method is called by the surrogate host after the child surrogate's `deactivate` method has returned. This call must return in a timely manner.

SA.2.4.5 The `SurrogateController` Interface

The `net.jini.surrogate.SurrogateController` interface allows a parent surrogate to deactivate a child surrogate. An object implementing the `SurrogateController` interface is returned from a successful call to the `newSurrogate` method of the `HostContext` interface.

```
package net.jini.surrogate

public interface SurrogateController {

    public void deactivate();
}
```

The Semantics

- The `deactivate` method requests that the surrogate host deactivate the child surrogate this controller represents. This method returns immediately upon making the request. If the child surrogate has not been deactivated, the host must deactivate the child surrogate by calling it's `deactivate` method.

  If the controller's `deactivate` method had previously been called or the child surrogate has already been deactivated, any calls to the `deactivate` method have no effect.

SA.2.4.6 The `SurrogateCreationException` Class

A `net.jini.surrogate.SurrogateCreationException` is thrown if there is any failure during the call to the `newSurrogate` method of the `HostContext` interface.

```
package net.jini.surrogate;

public class SurrogateCreationException
                                extends java.lang.Exception {

    public SurrogateCreationException(String message) {...}

    public SurrogateCreationException(String message,
                                Throwable nestedException) {...}

    public String getMessage();

    public Throwable getNestedException();
}
```

The Semantics

- The `getMessage` method returns the message associated with the exception, including the message from the nested exception if there is one.
- The `getNestedException` returns the nested exception, if there is one, otherwise `null` is returned.

SA.2.4.7 The `KeepAliveManagement` Interface

The use of the `KeepAliveManagement` (as well as the `KeepAliveHandler`, described in the next section) is interconnect-specific and is therefore specified by an interconnect specification (see Section SA.4.2, “Interconnect-Specific Programming Model”). The use of these interfaces is dependent on how liveness is defined for an interconnect. If the surrogate is involved in the liveness function, the keep-alive interfaces can aid the surrogate in fulfilling its obligations. These interfaces also allow the host to control the amount of liveness-related activity that goes on between the device and the surrogate.

The `net.jini.surrogate.KeepAliveManagement` interface provides a method to the surrogate to set a keep-alive handler. If an interconnect specification specifies that the `context` parameter implement the `KeepAliveManagement` interface, an object that implements `KeepAliveManagement` is passed as the `context` parameter to the surrogate’s `activate` method and to the `getCodebase` method of `GetCodebase`. A surrogate for that interconnect should use the `KeepAliveManagement` interface to set a surrogate-supplied keep-alive handler. Once a keep-alive handler is set, it is called periodically so that the surrogate can perform liveness-related work, including communicating with the device. After the first call to the keep-alive handler, the time between subsequent calls is called the *keep-alive period*.

```
package net.jini.surrogate;

public interface KeepAliveManagement {
    void setKeepAliveHandler(KeepAliveHandler handler);
}
```

The Semantics

- The `setKeepAliveHandler` method sets the keep-alive handler for this surrogate. The object passed in as the `handler` parameter must either implement the `KeepAliveHandler` interface or be `null` (see Section SA.2.4.8, “The `KeepAliveHandler` Interface”).

  To set a keep-alive handler the surrogate calls this method with a non-`null` handler. The `keepAlive` method of that handler is called periodically. If a keep-alive handler was set in a previous call to `setKeepAliveHandler`, that handler is no longer called, the keep-alive period is recalculated, and the next `keepAlive` call is made to the new handler.

  A `null` handler indicates that no keep-alive handler is to be set for this surrogate. If a keep-alive handler was set in a previous call to `setKeepAliveHandler` that handler is no longer called.

The keep-alive period is determined by the caller of the handler. The initial keep-alive period is provided to the surrogate in the first call to the handler's `keepAlive` method.

The handler's `keepAlive` method is called immediately after the `setKeepAliveHandler` method has returned in order to initialize the keep-alive period. If the surrogate's `activate` method has not been called, calling the handler must be deferred until after `activate` has executed successfully.

SA.2.4.8 The `KeepAliveHandler` Interface

The `net.jini.surrogate.KeepAliveHandler` interface provides a method to aid the surrogate in fulfilling its liveness obligations. The use of the keep-alive mechanism is specified by the interconnect specification (see Section SA.4.2, "Interconnect-Specific Programming Model").

The surrogate must provide an object that implements the `KeepAliveHandler` interface to the `setKeepAliveHandler` method of `KeepAliveManagement`. The `keepAlive` method is called periodically so that the surrogate may perform operations related to liveness. The exact nature of those operations is dependent on the requirements specified in the interconnect specification and the implementation of the surrogate and device.

If the surrogate finds that the device is no longer reachable, the surrogate must shut itself down by calling the `cancelActivation` method of the `hostContext` object that was passed into the surrogate's `activate` method.

```
package net.jini.surrogate;

public interface KeepAliveHandler {

    void keepAlive(long period) throws java.lang.Exception;
}
```

The Semantics

- The `keepAlive` method is called periodically. The surrogate should use this call to perform operations related to liveness.

  The `period` parameter is the maximum time, in milliseconds, that elapses before `keepAlive` is called again. The `period` value may change from call to call.

  The `period` value can be used to inform the device on how long to wait for the surrogate to check back again. If the device does not hear from the surrogate within the specified time the device can assume that it is no longer in contact with the surrogate.

If the `keepAlive` method is about to be invoked, and a previous call to a keep-alive handler's `keepAlive` method has not returned, the caller must not call `keepAlive` and must deactivate the surrogate.

If an exception is thrown by the `keepAlive` method, the caller must deactivate the surrogate.

SA.2.4.9 Serialized Forms

Class	`serialVersionUID`	Serialized Fields
`SurrogateCreationException`	`-1866927322476603821L`	`Throwable nestedException`

SA.2.5 Surrogate Packaging

A surrogate is packaged in a JAR file. This JAR file contains:

- A manifest file containing at least the following elements:
  - a single, mandatory `Surrogate-Class` header specifying the name of the surrogate class and
  - an optional `Surrogate-Codebase` header, specifying the resources of the surrogate's codebase.
- A class that implements the `Surrogate` interface, as well as any other resources that implement the surrogate. These resources may be classes written in the Java programming language, as well as any other data such as HTML files or icons.

The manifest of the surrogate JAR file must not contain a `Class-Path` header. If this header is present the surrogate JAR file is discarded.

How the surrogate JAR file is retrieved is defined by an interconnect specification.

SA.2.5.1 Surrogate-Class Header

The required Surrogate-Class header provides the fully qualified name of the class within the surrogate JAR file that implements the Surrogate interface. The surrogate host uses this information to locate the surrogate class so that it can be instantiated and activated. For example:

```
Surrogate-Class:    com.acme.surrogate.mySurrogate
```

The surrogate manifest file must contain only one Surrogate-Class header; therefore, there may be only one surrogate in a JAR file.

SA.2.5.2 Surrogate-Codebase Header

The optional Surrogate-Codebase header in the manifest of the surrogate JAR file specifies the resources that comprise the surrogate's codebase. The header has the following syntax:

```
Surrogate-Codebase: resource[ resource]*
```

resource is a name that represents a resource. The name must refer to an entry in the surrogate JAR file. The entry must be a JAR file. More than one resource name may specified and are separated by a space. If the resource names are not unique from each other, the results are undefined.

When the surrogate is loaded, the surrogate host checks for the existence of the Surrogate-Codebase header. If it is present, the specified resources are passed to the surrogate host's export server (see Section SA.2.3, "Export Server"). If a resource in an Surrogate-Codebase header is not found, the surrogate is discarded.

Before the activate method of the surrogate is called the surrogate host sets the codebase annotation for the surrogate. This annotation is expressed as a set of URLs and is used to annotate the surrogate's Java classes that are exported so they are available for downloading to remote entities. This set of URLs consists of the URLs that are generated by the export server for all of the resources specified in the Surrogate-Codebase header. These entries are assumed to represent JAR files containing the export Java classes of the surrogate. If there is no Surrogate-Codebase header in the manifest of the surrogate JAR file, the surrogate's codebase annotation is null.

The codebase annotation can also be set by the surrogate through the GetCodebase interface (see Section SA.2.4.2, "The GetCodebase Interface")

SA.2.6 Security

There are many aspects of security. First, for the surrogate host to be reliable and robust, it should protect itself from broken or malicious surrogates. Likewise, each surrogate should be isolated from other surrogates that the host is managing and from the host itself. Lastly, all parties should be protected from code that the surrogate may download. These security considerations impact the capabilities granted to the surrogate and are discussed next.

SA.2.6.1 Surrogate Isolation

The surrogate host must provide a Java 2 platform runtime environment that maintains adequate isolation of the surrogate. In this context, isolation means that the action, outside of defined interfaces, of one entity does not affect any other. The need for isolation is between the surrogate host and the surrogate and between each surrogate being managed by the host. The basic mechanism for isolation is described in Section SA.2.1, "Java 2 Platform Runtime Environment". Any further isolation is dependent on the surrogate host implementation and must not impact the surrogate programming model. The surrogate developer must not assume nor depend on a particular isolation mechanism.

SA.2.6.2 Permissions

In general, the surrogate has limited access to its runtime environment. This section describes the minimal set of capabilities and permissions provided to the surrogate by the surrogate host.

Threads

The surrogate must have permission to create new threads in the same thread group as the thread in which the surrogate's `activate` method is invoked. The surrogate must also have permission to interrupt any thread that it creates.

Persistent Storage

The surrogate developer must not assume that there is any type of persistent storage provided by the surrogate host or that the surrogate is granted any permissions to access persistent storage. If the surrogate needs to persist any information, such as its service ID, it should store that information on the device itself or use an outside persistent storage service.

Jini Discovery Permission

It is recommended that the surrogate use the `DiscoveryManagement` object returned by the `getDiscoveryManager` method of `HostContext`. This Jini discovery management object must be able to perform Jini lookup service group discovery using the groups configured for that surrogate host. The surrogate may choose to perform Jini discovery using its own selection of groups, if the surrogate is granted group Jini discovery permission. The following example permission allows Jini discovery of all groups:

```
net.jini.discovery.DiscoveryPermission “*”
```

If the surrogate does not use the Jini discovery management object, the surrogate must be prepared to handle a security exception because the host is not required to grant any Jini discovery permission to the surrogate.

Connection Permissions

The implementation of the `net.jini` classes and interfaces provided by the surrogate host might require that the surrogate have connection permission for the Jini discovery protocol. If they do the host must grant those permissions. The following is an example of connection permissions that might be required:

```
java.net.SocketPermission “224.0.1.84”, “connect,accept”
java.net.SocketPermission “224.0.1.85”, “connect,accept”
```

Connection permission to HTTP servers must be granted in order for code to be downloaded to the surrogate. The following example permissions allow connections to HTTP servers on commonly used ports:

```
java.net.SocketPermission “*.80”, “connect,accept”
java.net.SocketPermission “*:1024-”, “connect,accept”
```

Code downloaded into the surrogate, such as the Jini lookup service proxy, may need connection permission back to the machine that is providing the back-end of the proxy. Since the back-end locations are not known it is recommended that general connection permission be granted to the surrogate.

Interconnect-Specific Permissions

There might be additional permissions required by the surrogate due to the requirements of a particular interconnect. These permissions must be described in the interconnect specification for that interconnect.

SA.2.6.3 Security Manager

The implementation of the `net.jini` classes and interfaces that the surrogate host provides may require that a security manager be installed to manage the security considerations related to foreign code downloaded into the JVM in which each surrogate is executed. If a security manager is required, the surrogate host must properly set the system security manager on the JVM in which the surrogate is run.

SA.3Surrogate Programming Considerations

This section describes some of the behavior of the surrogate host and additional programming considerations for the surrogate developer.

SA.3.1 Surrogate Activation and Deactivation

Putting the previous sections together, it is possible to describe the overall process that takes place when a surrogate is activated and when it is deactivated.

SA.3.1.1 Activation

The following steps are taken by the surrogate host to activate a surrogate:

1. The manifest of the surrogate JAR file is searched for the mandatory `Surrogate-Class` header. If the `Surrogate-Class` header is not present, the surrogate JAR file is discarded.
2. The manifest of the surrogate JAR file is searched for the optional `Surrogate-Codebase` header. If the `Surrogate-Codebase` header is present in the specified resources are extracted from the JAR file and provided to the export server. The resulting URLs is used as the codebase annotation of the surrogate. If the host is unable to locate any of the specified resources, the surrogate is discarded. If the `Surrogate-Codebase` header is not present, the codebase annotation is `null`.
3. A suitable execution environment is created that contains a new thread and a class loader for the surrogate.
4. The surrogate object is instantiated. If the surrogate class is not found or some other error occurs the surrogate is discarded.
5. The surrogate object is checked to see if it implements the `GetCodebase` interface. If it does, the `getCodebase` method is invoked and the return

value is used to set the codebase annotation of the surrogate. If the `getCodebase` method throws an exception, the surrogate is discarded.

If the surrogate object does not implement the `GetCodebase` interface the result of the search for the `Surrogate-Codebase` header is used to set the surrogate's codebase annotation.

6. The surrogate's `activate` method is invoked. If the `activate` method throws an exception, the surrogate's `deactivate` method is called and the surrogate is discarded.

Because of the possibility that the surrogate may be discarded after it is instantiated, it is strongly recommended that the surrogate not start any threads in its constructor or in the `getCodebase` method. For the same reason, the surrogate should not inform the device that it has started until the `activate` method is called. If it is the responsibility of the surrogate host to inform the device that the surrogate has activated, it must do so after the `activate` method returns successfully.

SA.3.1.2 Deactivation

The following steps are taken by the surrogate host to deactivate a surrogate:

1. The surrogate's `deactivate` method is invoked. The surrogate preforms all necessary cleanup including stopping of threads, releasing of remote resources. Upon returning from `deactivate`, the states of the `hostContext` and the `context` objects for the deactivated surrogate are undefined.
2. Once `deactivate` returns, the surrogate is discarded and all resources associated with it are released, including the resources being exported by the export server. The host may also destroy the surrogate execution environment, including destroying any surrogate threads and the surrogate's JVM.

A surrogate host may initiated deactivation for a number of reasons. For example, the host may be in the process of shutting down, or in the case where the host is responsible for liveness, it may have determined that the device that the surrogate represents it no longer reachable. An active surrogate can also initiate deactivation by calling the `cancelActivation` method on the `hostContext` object.

Because of the possibility that the surrogate might stop execution abnormally, without its `deactivate` method being called, a device must not rely on being informed by the surrogate or the surrogate host that its surrogate has been deactivated.

SA.3.2 Non-responsiveness

When the surrogate host calls a method of the surrogate, the method may not return. Although these methods are required to return in a "timely manner", it is up to the specific surrogate host implementation to determine what is "timely" and reasonable for its environment. If a surrogate method does not return quickly enough, the host may decide that the surrogate is non-responsive and should take implementation-dependent steps to remedy the problem. For example, if the surrogate is run in its own JVM, the surrogate host could destroy the sub-process in which the JVM is running.

SA.3.3 Logging

There are no explicit logging facilities provided by the surrogate host. It is recommended that if the surrogate wishes to publish information about error conditions or debugging, that it either locate a logging service or use facilities in the Java 2 SDK, such as `System.out` and `System.err`.

SA.3.4 Service ID

If the device, through the surrogate, registers as a Jini technology-enabled service, the convention is that the service should use the same service ID each time it registers, if it is the same service. There are two things to consider: obtaining a service ID and persisting the ID for future registrations.

The device or surrogate may create its own service ID, possibly based on some unique information within the device, or it can obtain one that is generated when the service is registered for the first time. If the service ID is obtained from a Jini lookup service registration, it should be stored so that future registrations use the same ID. The surrogate host does not provide any persistence facility so the surrogate should either locate a persistence service or, if the device has some writable persistent store, pass the service ID back to the device.

SA.3.5 Downloaded Classes

By default, a class downloaded via RMI is only granted permission to communicate with the machine from where the class was downloaded. This restriction impacts the capabilities of any class (proxy) that a surrogate exports.

The host machine from where a surrogate's proxy is downloaded is specified in the surrogate's codebase annotation. The codebase annotation of the surrogate is set by the surrogate host. This annotation is either specified in the `Surrogate-Codebase` manifest header or retrieved through the `GetCodebase` interface (see Section SA.2.5.2, "`Surrogate-Codebase` Header" and Section SA.2.4.2, "The `GetCodebase` Interface").

If the `Surrogate-Codebase` header is present in the manifest of the surrogate JAR file, the surrogate host, will use the export server to provide the annotation for the resources described in the header. The provided annotation must reference the host machine on which the surrogate is running. So a proxy downloaded from this codebase may only have permission to communicate with the surrogate running on the host machine, and the proxy might not have permission to communicate directly with the device its surrogate represents.

If the surrogate specifies its own codebase annotation through the `GetCodebase` interface, a proxy exported by that surrogate may only have permission to communicate to the host machine described in that codebase annotation. If the specified host machine is different than the host machine on which the surrogate is running, the proxy might not have permission to communicate with its surrogate.

SA.4Interconnect Specification

As defined in Section SA.1.2, “Definitions”, the term interconnect protocol defines the mechanisms for discovery, surrogate retrieval, and liveness. These mechanisms are device and interconnect dependent and are, therefore, not defined in this specification. However, for each interconnect there must be an *interconnect specification* that fully describes discovery, surrogate retrieval, and liveness. In addition, an interconnect specification must also fully describe any interconnect-specific additions to the surrogate programming model.

A device or surrogate host that uses the surrogate architecture for a given interconnect must comply with both an interconnect specification for that interconnect and the *Jini™ Technology Surrogate Architecture Specification*.

SA.4.1 Interconnect Protocol Requirements

The following sections describe the required components of a interconnect protocol. In defining these components for a particular interconnect, the existing protocols of that interconnect may be used or new protocols may be developed. This choice depends on the ability of the existing protocols to meet the requirements of discovery, retrieval, and liveness.

SA.4.1.1 Discovery

The mechanism by which a device and a surrogate host locate one another must be specified. This mechanism is necessary and may consist of hardware or software protocols, or other means that are appropriate to the specific interconnect to maintain the plug-and-work model.

The discovery mechanism must define a way in which a device, when attached to an interconnect, can either: announce its presence such that a surrogate host can detect it, or detect a surrogate host on that interconnect. Likewise, the mechanism must define a way that a surrogate host attaching to an interconnect can announce its presence or, through some other means, find all of the devices on the interconnect.

The discovery mechanisms must operate without requiring administration of the surrogate host. This means that no external action should be required in order for the surrogate host to recognize a new device on the interconnect that it supports.

SA.4.1.2 Surrogate Retrieval

All interconnect specifications must define how surrogate components are retrieved. The surrogate retrieval portion of the interconnect specification must describe a mechanism that is device independent. Device independence in this context means that the surrogate host does not need prior knowledge of any specific device that implements the interconnect specification. This restriction is important in order to introduce a new device type for an existing interconnect, without requiring changes to the surrogate host.

SA.4.1.3 Liveness

All interconnect specifications must specify the mechanism for determining whether the device is still reachable on the interconnect. This requirement is necessary to maintain the surrogate's residency in the surrogate host and to allow for the deactivation of the surrogate if the device is no longer functioning, is no longer attached to the interconnect, or if the interconnect has failed. This mechanism is also necessary to allow the release of any leased resources held by the surrogate. Liveness also includes the ability of the device to determine if it is still in contact with its surrogate. It may be the responsibility of the surrogate host or the surrogate to determine liveness.

SA.4.2 Interconnect-Specific Programming Model

The interconnect-specific programming model refers to the programming model for a surrogate that is to be used with a specific interconnect. An interconnect specification may define an interconnect-specific programming model by extending the programing model defined in Section SA.2, "Jini Technology-Enabled Surrogate" by specifying additional required Java programming language APIs and surrogate APIs.

There may also be security considerations related to the interconnect which must be fully described.

SA.4.2.1 Java Programming Language APIs

It is permissible for the interconnect specification to extend the surrogate programming model by including the definition of additional APIs that are required to be made available to the surrogate. This set may be any required APIs that would not be included in the set of APIs already defined in this specification. For example, this set may include such things as utility or extension classes for communicating over the interconnect.

If additional APIs are specified, their implementation must be provided by all surrogate hosts conforming to the interconnect specification. The surrogate host must ensure that the necessary classes are accessible through the surrogate's class loader. If any classes are meant to be exported they must have their codebase annotation set to the same codebase annotation of the surrogate.

SA.4.2.2 Surrogate APIs

The interconnect specification may define one or more object or interface types for the `context` parameter of the surrogate's `activate` and `getCodebase` methods to provide the surrogate access to the interconnect-specific context.

The interconnect specification may also specify that the `context` object must implement the `KeepAliveManagement` interface. The use of this interface depends on how liveness is defined for the interconnect. If the surrogate is involved in liveness, the `KeepAliveManagement` and the `KeepAliveHandler` can aid the surrogate in performing this task.

The state of the `context` object is undefined after the surrogate's `deactivate` method has been called by the surrogate host or if the surrogate has otherwise been discarded.

SA.4.2.3 Security

The interconnect specification may contain interconnect-specific security components such as mechanisms for secure communication over the interconnect, authentication, and encryption. It may also include programming considerations for the surrogate such as additional permissions to be granted by the surrogate host.

License