lock, then serializability might be violated. When a subtransaction commits, its locks are inherited by the parent transaction.

In addition to locks, transactional operations can be defined in terms of object creation and deletion visibility. If an object is defined to be created under a transaction, then the existence of the object is visible only within that transaction and its inferiors, but will disappear if the transaction aborts. If an object is defined to be deleted under a transaction, then the object is not visible to any transaction (including the deleting transaction) but will reappear if the transaction aborts. When a nested transaction commits, visibility state is inherited by the parent transaction.

Once a transaction reaches the `VOTING` stage, if all execution under the transaction (and its subtransactions) has finished, then the only reasons the transaction can abort are:

- The manager crashes (or has crashed)
- One or more participants crash (or have crashed)
- There is an explicit abort

Transaction deadlocks are not guaranteed to be prevented or even detected, but managers and participants are permitted to break known deadlocks by aborting transactions.

An active transaction is an *orphan* if it or one of its ancestors is guaranteed to abort. This can occur because an ancestor has explicitly aborted or because some participant or manager of the transaction or an ancestor has crashed. Orphans are not guaranteed to be detected by the system, so programmers using transactions must be aware that orphans can see internally inconsistent state and take appropriate action.

Causal ordering information about transactions is not guaranteed to be propagated. First, given two sibling transactions (at any level), it is not possible to tell whether they were created concurrently or sequentially (or in what order). Second, if two transactions are causally ordered and the earlier transaction has completed, the outcome of the earlier transaction is not guaranteed to be known at every participant used by the later transaction, unless the client is successful in using the variant of `commit` or `abort` that takes a timeout parameter. Programmers using non-blocking forms of operations must take this into account.

As long as a transaction persists in attempting to acquire a lock that conflicts with another transaction, the participant will persist in attempting to resolve the outcome of the transaction that holds the conflicting lock. Attempts to acquire a lock include making a blocking call, continuing to make non-blocking calls, and registering for event notification under a transaction.

TX.3.6 Serialized Forms

Class	serialVersionUID	Serialized Fields
Transaction.Created	–5199291723008952986L	*all public fields*
NestableTransaction.Created	–2979247545926318953L	*all public fields*
TransactionManager.Created	–4233846033773471113L	*all public fields*
ServerTransaction	4552277137549765374L	*all public fields*
NestableServerTransaction	–3438419132543972925L	*all public fields*
TransactionException	–5009935764793203986L	*none*
CannotAbortException	3597101646737510009L	*none*
CannotCommitException	–4497341152359563957L	*none*
CannotJoinException	5568393043937204939L	*none*
CannotNestException	3409604500491735434L	*none*
TimeoutExpiredException	3918773760682958000L	*all public fields*
UnknownTransactionException	4437986299363270091	*none*
CrashCountException	42992261252450156711	*none*

LU

The Jini Lookup Service Specification

LU.1 Introduction

THE Jini Lookup service is a fundamental part of the federation infrastructure for a *djinn,* the group of devices, resources, and users that are joined by the Jini software infrastructure. The *lookup service* provides a central registry of services available within the djinn. This lookup service is a primary means for programs to find services within the djinn, and is the foundation for providing user interfaces through which users and administrators can discover and interact with services in the djinn.

Although the primary purpose of this specification is to define the interface to the djinn's central service registry, the interfaces defined here can readily be used in other service registries.

LU.1.1 The Lookup Service Model

The lookup service maintains a flat collection of *service items*. Each service item represents an instance of a service available within the djinn. The item contains the RMI stub (if the service is implemented as a remote object) or other object (if the service makes use of a local proxy) that programs use to access the service, and an extensible collection of attributes that describe the service or provide secondary interfaces to the service.

When a new service is created (for example, when a new device is added to the djinn), the service registers itself with the djinn's lookup service, providing an initial collection of attributes. For example, a printer might include attributes indi-

cating speed (in pages per minute), resolution (in dots per inch), and whether duplex printing is supported. Among the attributes might be an indicator that the service is new and needs to be configured.

An administrator uses the event mechanism of the lookup service to receive notifications as new services are registered. To configure the service, the administrator might look for an attribute that provides an applet for this purpose. The administrator might also use an applet to add new attributes, such as the physical location of the service and a common name for it; the service would receive these attribute change requests from the applet and respond by making the changes at the lookup service.

Programs (including other services) that need a particular type of service can use the lookup service to find an instance. A match can be made based on the specific data types for the Java programming language implemented by the service as well as the specific attributes attached to the service. For example, a program that needs to make use of transactions might look for a service that supports the type `net.jini.core.transaction.server.TransactionManager` and might further qualify the match by desired location.

Although the collection of service items is flat, a wide variety of hierarchical views can be imposed on the collection by aggregating items according to service types and attributes. The lookup service provides a set of methods to enable incremental exploration of the collection, and a variety of user interfaces can be built by using these methods, allowing users and administrators to browse. Once an appropriate service is found, the user might interact with the service by loading a user interface applet, attached as another attribute on the item.

If a service encounters some problem that needs administrative attention, such as a printer running out of toner, the service can add an attribute that indicates what the problem is. Administrators again use the event mechanism to receive notification of such problems.

LU.1.2 Attributes

The attributes of a service item are represented as a set of attribute sets. An individual *attribute set* is represented as an instance of some class for the Java platform, each attribute being a public field of that class. The class provides strong typing of both the set and the individual attributes. A service item can contain multiple instances of the same class with different attribute values, as well as multiple instances of different classes. For example, an item might have multiple instances of a `Name` class, each giving the common name of the service in a different language, plus an instance of a `Location` class, an `Owner` class, and various service-specific classes. The schema used for attributes is not constrained by this

specification, but a standard foundation schema for Jini systems is defined in the *Jini Lookup Attribute Schema Specification*.

Concretely, a set of attributes is implemented with a class that correctly implements the interface `net.jini.core.entry.Entry`, as described in the *Jini Entry Specification*. Operations on the lookup service are defined in terms of template matching, using the same semantics as in the *Jini Entry Specification*, but the definition is augmented to deal with sets of entries and sets of templates. A set of entries matches a set of templates if there is at least one matching entry for every template (with every entry usable as the match for more than one template).

LU.1.3 Dependencies

This specification relies on the following other specifications:

- *Java Remote Method Invocation Specification*
- *Java Object Serialization Specification*
- *Jini Entry Specification*
- *Jini Distributed Event Specification*
- *Jini Distributed Leasing Specification*
- *Jini Discovery and Join Specification*

LU.2 The ServiceRegistrar

THE types defined in this specification are in the net.jini.core.lookup package. The following types are imported from other packages and are referenced in unqualified form in the rest of this specification:

```
java.rmi.MarshalledObject
java.rmi.RemoteException
java.rmi.UnmarshalException
java.io.Serializable
java.io.DataInput
java.io.DataOutput
java.io.IOException
net.jini.core.discovery.LookupLocator
net.jini.core.entry.Entry
net.jini.core.lease.Lease
net.jini.core.event.RemoteEvent
net.jini.core.event.EventRegistration
net.jini.core.event.RemoteEventListener
```

LU.2.1 ServiceID

Every service is assigned a universally unique identifier (UUID), represented as an instance of the ServiceID class.

```
public final class ServiceID implements Serializable {
    public ServiceID(long mostSig, long leastSig) {…}
    public ServiceID(DataInput in) throws IOException {…}
    public void writeBytes(DataOutput out) throws IOException
        {…}
    public long getMostSignificantBits() {…}
    public long getLeastSignificantBits() {…}
}
```

A service ID is a 128-bit value. Service IDs are equal (using the `equals` method) if they represent the same 128-bit value. For simplicity and reliability, service IDs are intended to be generated only by lookup services, not by clients. As such, the `ServiceID` constructor merely takes 128 bits of data, to be computed in an implementation-dependent manner by the lookup service. The `writeBytes` method writes out 16 bytes in standard network byte order. The second constructor reads in 16 bytes in standard network byte order.

The most significant long can be decomposed into the following unsigned fields:

```
0xFFFFFFFF00000000        time_low
0x00000000FFFF0000        time_mid
0x000000000000F000        version
0x0000000000000FFF        time_hi
```

The least significant long can be decomposed into the following unsigned fields:

```
0xC000000000000000        variant
0x3FFF000000000000        clock_seq
0x0000FFFFFFFFFFFF        node
```

The `variant` field must be 0x2. The `version` field must be either 0x1 or 0x4. If the `version` field is 0x4, then the most significant bit of the `node` field must be set to 1, and the remaining fields are set to values produced by a cryptographically strong pseudo-random number generator. If the `version` field is 0x1, then the `node` field is set to an IEEE 802 address, the `clock_seq` field is set to a 14-bit random number, and the `time_low`, `time_mid`, and `time_hi` fields are set to the least, middle, and most significant bits (respectively) of a 60-bit timestamp measured in 100-nanosecond units since midnight, October 15, 1582 UTC.

The `toString` method returns a 36-character string of six fields separated by hyphens, each field represented in lowercase hexadecimal with the same number of digits as in the field. The order of fields is: `time_low`, `time_mid`, `version` and `time_hi` treated as a single field, `variant` and `clock_seq` treated as a single field, and `node`.

LU.2.2 ServiceItem

Items are stored in the lookup service using instances of the `ServiceItem` class.

```
public class ServiceItem implements Serializable {
    public ServiceItem(ServiceID serviceID,
                       Object service,
```

```
                        Entry[] attributeSets) {…}
    public ServiceID serviceID;
    public Object service;
    public Entry[] attributeSets;
}
```

The constructor simply assigns each parameter to the corresponding field.

Each `Entry` represents an attribute set. The class must have a public no-arg constructor, and all non-static, non-final, non-transient public fields must be declared with reference types, holding serializable objects. Each such field is serialized separately as a `MarshalledObject`, and field equality is defined by `MarshalledObject.equals`. The only relationship constraint on attribute sets within an item is that exact duplicates are eliminated; other than that, multiple attribute sets of the same type are permitted, multiple attribute set types can have a common superclass, and so on.

The `net.jini.core.entry.UnusableEntryException` is not used in the lookup service; alternate semantics for individual operations are defined later in this section.

LU.2.3 `ServiceTemplate` and Item Matching

Items in the lookup service are matched using instances of the `ServiceTemplate` class.

```
public class ServiceTemplate implements Serializable {
    public ServiceTemplate(ServiceID serviceID,
                           Class[] serviceTypes,
                           Entry[] attributeSetTemplates) {…}
    public ServiceID serviceID;
    public Class[] serviceTypes;
    public Entry[] attributeSetTemplates;
}
```

The constructor simply assigns each parameter to the corresponding field. A service item (`item`) matches a service template (`tmpl`) if:

- `item.serviceID` equals `tmpl.serviceID` (or if `tmpl.serviceID` is `null`), and
- `item.service` is an instance of every type in `tmpl.serviceTypes`, and
- `item.attributeSets` contains at least one matching entry for each entry template in `tmpl.attributeSetTemplates`.

An entry matches an entry template if the class of the template is the same as, or a superclass of, the class of the entry, and every non-`null` field in the template equals the corresponding field of the entry. Every entry can be used to match more than one template. For both service types and entry classes, type matching is based simply on fully qualified class names. Note that in a service template, for `serviceTypes` and `attributeSetTemplates`, a `null` field is equivalent to an empty array; both represent a wildcard.

LU.2.4 Other Supporting Types

The `ServiceMatches` class is used for the return value when looking up multiple items.

```
public class ServiceMatches implements Serializable {
    public ServiceMatches(ServiceItem[] items,
                          int totalMatches) {...}
    public ServiceItem[] items;
    public int totalMatches;
}
```

The constructor simply assigns each parameter to the corresponding field.

A `ServiceEvent` extends `RemoteEvent` with methods to obtain the service ID of the matched item, the transition that triggered the event, and the new state of the matched item.

```
public abstract class ServiceEvent extends RemoteEvent {
    public ServiceEvent(Object source,
                        long eventID,
                        long seqNum,
                        MarshalledObject handback,
                        ServiceID serviceID,
                        int transition) {...}
    public ServiceID getServiceID() {...}
    public int getTransition() {...}
    public abstract ServceItem getServiceItem() {...}
}
```

The `getServiceID` and `getTransition` methods return the value of the corresponding constructor parameter. The remaining constructor parameters are the same as in the `RemoteEvent` constructor.

The rest of the semantics of both these classes is explained in the next section.

LU.2.5 ServiceRegistrar

The ServiceRegistrar defines the interface to the lookup service. The interface is not a remote interface; each implementation of the lookup service exports proxy objects that implement the ServiceRegistrar interface local to the client, using an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal RMI remote interface semantics except where explicitly noted. Two proxy objects are equal (using the equals method) if they are proxies for the same lookup service.

Methods are provided to register service items, find items that match a template, receive event notifications when items are modified, and incrementally explore the collection of items along the three major axes: entry class, attribute value, and service type.


```
public interface ServiceRegistrar {
    ServiceRegistration register(ServiceItem item,
                                 long leaseDuration)
        throws RemoteException;

    Object lookup(ServiceTemplate tmpl)
        throws RemoteException;

    ServiceMatches
        lookup(ServiceTemplate tmpl, int maxMatches)
        throws RemoteException;

    int TRANSITION_MATCH_NOMATCH = 1 << 0;
    int TRANSITION_NOMATCH_MATCH = 1 << 1;
    int TRANSITION_MATCH_MATCH = 1 << 2;

    EventRegistration notify(ServiceTemplate tmpl,
                             int transitions,
                             RemoteEventListener listener,
                             MarshalledObject handback,
                             long leaseDuration)
        throws RemoteException;

    Class[] getEntryClasses(ServiceTemplate tmpl)
        throws RemoteException;

    Object[] getFieldValues(ServiceTemplate tmpl,
```

```
                                int setIndex,
                                String field)
        throws NoSuchFieldException, RemoteException;

    Class[] getServiceTypes(ServiceTemplate tmpl,
                                String prefix)
        throws RemoteException;

    ServiceID getServiceID();
    LookupLocator getLocator() throws RemoteException;

    String[] getGroups() throws RemoteException;
}
```

Every method invocation on `ServiceRegistrar` and `ServiceRegistration` is atomic with respect to other invocations.

The `register` method is used to register a new service and to re-register an existing service. The method is defined so that it can be used in an idempotent fashion. Specifically, if a call to `register` results in a `RemoteException` (in which case the item might or might not have been registered), the caller can simply repeat the call to `register` with the same parameters, until it succeeds.

To register a new service, `item.serviceID` should be `null`. In that case, if `item.service` does not equal (using `MarshalledObject.equals`) any existing item's service object, then a new service ID will be assigned and included in the returned `ServiceRegistration` (described in the next section). The service ID is unique over time and space with respect to all other service IDs generated by all lookup services. If `item.service` does equal an existing item's service object, the existing item is first deleted from the lookup service (even if it has different attributes) and its lease is cancelled, but that item's service ID is reused for the newly registered item.

To re-register an existing service, or to register the service in any other lookup service, `item.serviceID` should be set to the same service ID that was returned by the initial registration. If an item is already registered under the same service ID, the existing item is first deleted (even if it has different attributes or a different service instance) and its lease is cancelled by the lookup service. Note that service object equality is not checked in this case, to allow for reasonable evolution of the service (for example, the serialized form of the stub changes or the service implements a new interface).

Any duplicate attribute sets that are included in a service item are eliminated in the stored representation of the item. The lease duration request (specified in milliseconds) is not exact; the returned lease is allowed to have a shorter (but not

longer) duration than what was requested. The registration is persistent across restarts (crashes) of the lookup service until the lease expires or is cancelled.

The single-parameter form of `lookup` returns the service object (i.e., just `ServiceItem.service`) from an item matching the template or `null` if there is no match. If multiple items match the template, it is arbitrary as to which service object is returned by the invocation. If the returned object cannot be deserialized, an `UnmarshalException` is thrown with the standard RMI semantics.

The two-parameter form of `lookup` returns at most `maxMatches` items matching the template and the total number of items that match the template. The return value is never `null`, and the returned items array is `null` only if `maxMatches` is zero. For each returned item, if the service object cannot be deserialized, the `service` field of the item is set to `null` and no exception is thrown. Similarly, if an attribute set cannot be deserialized, that element of the `attributeSets` array is set to `null` and no exception is thrown.

The `notify` method is used to register for event notification. The registration is leased; the lease duration request (specified in milliseconds) is not exact. The registration is persistent across restarts (crashes) of the lookup service until the lease expires or is cancelled. The event ID in the returned `EventRegistration` is unique at least with respect to all other active event registrations at this lookup service with different service templates or transitions.

While the event registration is in effect, a `ServiceEvent` is sent to the specified listener whenever a `register`, lease cancellation or expiration, or attribute change operation results in an item changing state in a way that satisfies the template and transition combination. The `transitions` parameter is the bitwise OR of any non-empty set of transition values:

- `TRANSITION_MATCH_NOMATCH`: An event is sent when the changed item matches the template before the operation, but doesn't match the template after the operation (this includes deletion of the item).
- `TRANSITION_NOMATCH_MATCH`: An event is sent when the changed item doesn't match the template before the operation (this includes not existing), but does match the template after the operation.
- `TRANSITION_MATCH_MATCH`: An event is sent when the changed item matches the template both before and after the operation.

The `getTransition` method of `ServiceEvent` returns the singleton transition value that triggered the match.

The `getServiceItem` method of `ServiceEvent` returns the new state of the item (the state after the operation) or `null` if the item was deleted by the operation. Note that this method is declared `abstract`; a lookup service uses a subclass of `ServiceEvent` to transmit the new state of the item however it chooses.

Sequence numbers for a given event ID are strictly increasing. If there is no gap between two sequence numbers, no events have been missed; if there is a gap, events might (but might not) have been missed. For example, a gap might occur if the lookup service crashes, even if no events are lost due to the crash.

As mentioned earlier, users are allowed to explore a collection of items down each of the major axes: entry class, attribute value, and service type.

The `getEntryClasses` method looks at all service items that match the specified template, finds every entry (among those service items) that either doesn't match any entry templates or is a subclass of at least one matching entry template, and returns the set of the (most specific) classes of those entries. Duplicate classes are eliminated, and the order of classes within the returned array is arbitrary. A `null` reference (not an empty array) is returned if there are no such entries or no matching items. If a returned class cannot be deserialized, that element of the returned array is set to `null` and no exception is thrown.

The `getFieldValues` method looks at all service items that match the specified template, finds every entry (among those service items) that matches `tmpl.attributeSetTemplates[setIndex]`, and returns the set of values of the specified field of those entries. Duplicate values are eliminated, and the order of values within the returned array is arbitrary. a `null` reference (not an empty array) is returned if there are no matching items. If a returned value cannot be deserialized, that element of the returned array is set to `null` and no exception is thrown. `NoSuchFieldException` is thrown if `field` does not name a field of the entry template.

The `getServiceTypes` method looks at all service items that match the specified template and, for every service item, finds the most specific type (class or interface) or types the service item is an instance of that are neither equal to, nor a superclass of, any of the service types in the template and that have names that start with the specified prefix, and returns the set of all such types. Duplicate types are eliminated, and the order of types within the returned array is arbitrary. A `null` reference (not an empty array) is returned if there are no such types. If a returned type cannot be deserialized, that element of the returned array is set to `null` and no exception is thrown.

Every lookup service assigns itself a service ID when it is first created; this service ID is returned by the `getServiceID` method. (Note that this does not make a remote call.) A lookup service is always registered with itself under this service ID, and if a lookup service is configured to register itself with other lookup services, it will register with all of them using this same service ID.

The `getLocator` method returns a `LookupLocator` that can be used if necessary for unicast discovery of the lookup service. The definition of this class is given in the *Jini Technology Discovery and Join Specification*.

The `getGroups` method returns the set of groups that this lookup service is currently a member of. The semantics of these groups is defined in the *Jini Technology Discovery and Join Specification*.

LU.2.6 ServiceRegistration

A registered service item is manipulated using a `ServiceRegistration` instance.

```
public interface ServiceRegistration {
    ServiceID getServiceID();
    Lease getLease();
    void addAttributes(Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;
    void modifyAttributes(Entry[] attrSetTemplates,
                          Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;
    void setAttributes(Entry[] attrSets)
        throws UnknownLeaseException, RemoteException;
}
```

Like `ServiceRegistrar`, this is not a remote interface; each implementation of the lookup service exports proxy objects that implement this interface local to the client. The proxy methods obey normal RMI remote interface semantics.

The `getServiceID` method returns the service ID for this service. (Note that this does not make a remote call.)

The `getLease` method returns the lease that controls the service registration, allowing the lease to be renewed or cancelled. (Note that `getLease` does not make a remote call.)

The `addAttributes` method adds the specified attribute sets (those that aren't duplicates of existing attribute sets) to the registered service item. Note that this operation has no effect on existing attribute sets of the service item and can be repeated in an idempotent fashion. `UnknownLeaseException` is thrown if the registration lease has expired or been cancelled.

The `modifyAttributes` method is used to modify existing attribute sets. The lengths of the `attrSetTemplates` and `attrSets` arrays must be equal, or `IllegalArgumentException` is thrown. The service item's attribute sets are modified as follows. For each array index `i`: if `attrSets[i]` is `null`, then every entry that matches `attrSetTemplates[i]` is deleted; otherwise, for every non-`null` field in `attrSets[i]`, the value of that field is stored into the corresponding field of every entry that matches `attrSetTemplates[i]`. The class of `attrSets[i]` must be the same as, or a superclass of, the class of `attrSetTemplates[i]`, or

`IllegalArgumentException` is thrown. If the modifications result in duplicate entries within the service item, the duplicates are eliminated. An `UnknownLeaseException` is thrown if the registration lease has expired or been cancelled.

Note that it is possible to use `modifyAttributes` in ways that are not idempotent. The attribute schema should be designed in such a way that all intended uses of this method can be performed in an idempotent fashion. Also note that `modifyAttributes` does not provide a means for setting a field to `null`; it is assumed that the attribute schema is designed in such a way that this is not necessary.

The `setAttributes` method deletes all of the service item's existing attributes and replaces them with the specified attribute sets. Any duplicate attribute sets are eliminated in the stored representation of the item. `UnknownLeaseException` is thrown if the registration lease has expired or been cancelled.

LU.2.7 Serialized Forms

Class	serialVersionUID	Serialized Fields
`ServiceID`	–7803375959559762239L	`long mostSig` `long leastSig`
`ServiceItem`	717395451032330758L	*all public fields*
`ServiceTemplate`	7854483807886483216L	*all public fields*
`ServiceMatches`	–5518280843537399398L	*all public fields*
`ServiceEvent`	1304997274096842701L	`ServiceID serviceID` `int transition`

Lookup
(LU)

THE JINI LOOKUP ATTRIBUTE SCHEMA SPECIFICATION defines a set of attributes that a local administrator might choose to place on a service. These are "serving suggestions"—nobody is required to use these attribute definitions, but they give a starting point for people who need such attributes to either use directly or use for inspiration. This also describes the common style for entry design, including the canonical way to present your entry as a JavaBean object.

LS

The Jini Lookup Attribute Schema Specification

LS.1 Introduction

THE Jini Lookup service provides facilities for services to advertise their availability and for would-be clients to obtain references to those services based on the attributes they provide. The mechanism that it provides for registering and querying based on attributes is centered on the Java platform type system, and is based on the notion of an *entry*.

An entry is a class that contains a number of public fields of object type. Services provide concrete values for each of these fields; each value acts as an attribute. Entries thus provide aggregation of attributes into sets; a service may provide several entries when registering itself in the lookup service, which means that attributes on each service are provided in a set of sets.

The purpose of this document is to provide a framework in which services and their would-be clients can interoperate. This framework takes two parts:

- We describe a set of common predefined entries that span much of the basic functionality that is needed both by services registering themselves and by entities that are searching for services.
- Since we cannot anticipate all of the future needs of clients of the lookup service, we provide a set of guidelines and design patterns for extending, using, and imitating this set in ways that are consistent and predictable. We also construct some examples that illustrate the use of these patterns.

LS.1.1 Terminology

Throughout this document, we will use the following terms in consistent ways:

- *Service*—a service that has registered, or will register, itself with the lookup service
- *Client*—an entity that performs queries on the lookup service, in order to find particular services

LS.1.2 Design Issues

Several factors influence and constrain the design of the lookup service schema.

Matching Cannot Always Be Automated

No matter how much information it has at its disposal, a client of the lookup service will not always be able to find a single unique match without assistance when it performs a lookup. In many instances we expect that more than one service will match a particular query. Accordingly, both the lookup service and the attribute schema are geared toward reducing the number of matches that are returned on a given lookup to a minimum, and not necessarily to just one.

Attributes Are Mostly Static

We have designed the schema for the lookup service with the assumption that most attributes will not need to be changed frequently. For example, we do not expect attributes to change more often than once every minute or so. This decision is based on our expectation that clients that need to make a choice of service based on more frequently updated attributes will be able to talk to whatever small set of services the lookup service returns for a query, and on our belief that the benefit of updating attributes frequently at the lookup service is outweighed by the cost in network traffic and processing.

Humans Need to Understand Most Attributes

A corollary of the idea that matching cannot always be automated is that humans—whether they be users or administrators of services—must be able to understand and interpret attributes. This has several implications:

- We must provide a mechanism to deal with localization of attributes
- Multiple-valued attributes must provide a way for humans to see only one value (see Section LS.2)

We will cover human accessibility of attributes soon.

Attributes Can Be Changed by Services or Humans, But Not Both

For any given attribute class we expect that attributes within that class will all be set or modified either by the service, or via human intervention, but not both. What do we mean by this? A service is unlikely to be able to determine that it has been moved from one room to another, for example, so we would not expect the fields of a "location" attribute class to be changed by the service itself. Similarly, we do not expect that a human operator will need to change the name of the vendor of a particular service.

This idea has implications for our approach to ensuring that the values of attributes are valid.

Attributes Must Interoperate with JavaBeans Components

The JavaBeans specification provides a number of facilities relating to the localized display and modification of properties, and has been widely adopted. It is to our advantage to provide a familiar set of mechanisms for manipulating attributes in these ways.

LS.1.3 Dependencies

This document relies on the following other specifications:

- *Jini Entry Specification*
- *Jini Entry Utilities Specification*
- *JavaBeans Specification*

LS.2 Human Access to Attributes

LS.2.1 Providing a Single View of an Attribute's Value

CONSIDER the following entry class:

```
public class Foo implements net.jini.core.entry.Entry {
    public Bar baz;
}

public class Bar {
    int quux;
    boolean zot;
}
```

A visual search tool is going to have a difficult time rendering the value of an instance of class `Bar` in a manner that is comprehensible to humans. Accordingly, to avoid such situations, entry class implementors should use the following guidelines when designing a class that is to act as a value for an attribute:

- Provide a property editor class of the appropriate type, as described in Section 9.2 of the *JavaBeans Specification*.
- Extend the `java.awt.Component` class; this allows a value to be represented by a JavaBeans component or some other "active" object.
- Provide either a non-default implementation of the `Object.toString` method or inherit directly or indirectly from a class that does so (since the default implementation of `Object.toString` is not useful).

One of the above guidelines should be followed for all attribute value classes. Authors of entry classes should assume that any attribute value that does not satisfy one of these guidelines will be ignored by some or all user interfaces.

LS.3 JavaBeans Components and Design Patterns

LS.3.1 Allowing Display and Modification of Attributes

We use JavaBeans components to provide a layer of abstraction on top of the individual classes that implement the `net.jini.core.entry.Entry` interface. This provides us with several benefits:

- This approach uses an existing standard and thus reduces the amount of unfamiliar material for programmers.
- JavaBeans components provide mechanisms for localized display of attribute values and descriptions.
- Modification of attributes is also handled, via property editors.

LS.3.1.1 Using JavaBeans Components with Entry Classes

Many, if not most, entry classes should have a bean class associated with them. Our use of JavaBeans components provides a familiar mechanism for authors of browse/search tools to represent information about a service's attributes, such as its icons and appropriately localized descriptions of the meanings and values of its attributes. JavaBeans components also play a role in permitting administrators of a service to modify some of its attributes, as they can manipulate the values of its attributes using standard JavaBeans component mechanisms.

For example, obtaining a `java.beans.BeanDescriptor` for a JavaBeans component that is linked to a "location" entry object for a particular service allows a programmer to obtain an icon that gives a visual indication of what that entry class is for, along with a short textual description of the class and the values of the individual attributes in the location object. It also permits an administrative tool to view and change certain fields in the location, such as the floor number.

LS.3.2 Associating JavaBeans Components with Entry Classes

The pattern for establishing a link between an entry object and an instance of its JavaBeans component is simple enough, as this example illustrates:

```
package org.example.foo;

import java.io.Serializable;
import net.jini.lookup.entry.EntryBean;
import net.jini.entry.AbstractEntry;

public class Size {
    public int value;
}

public class Cavenewt extends AbstractEntry {
    public Cavenewt() {
    }
    public Cavenewt(Size anvilSize) {
        this.anvilSize = anvilSize;
    }
    public Size anvilSize;
}

public class CavenewtBean implements EntryBean, Serializable {
    protected Cavenewt assoc;
    public CavenewtBean() {
        super();
        assoc = new Cavenewt();
    }
    public void setAnvilSize(Size x) {
        assoc.anvilSize = x;
    }
    public Size getAnvilSize() {
        return assoc.anvilSize;
    }
    public void makeLink(Entry obj) {
         assoc = (Cavenewt) obj;
    }
    public Entry followLink() {
        return assoc;
    }
}
```

From the above, the pattern should be relatively clear:

- The name of a JavaBeans component is derived by taking the fully qualified entry class name and appending the string `Bean`; for example, the name of the JavaBeans component associated with the entry class `foo.bar.Baz` is `foo.bar.BazBean`. This implies that an entry class and its associated JavaBeans component must reside in the same package.
- The class has both a public no-arg constructor and a public constructor that takes each public object field of the class and its superclasses as parameter. The former constructs an empty instance of the class, and the latter initializes each field of the new instance to the given parameter.
- The class implements the `net.jini.core.entry.Entry` interface, preferably by extending the `net.jini.entry.AbstractEntry` class, and the JavaBeans component implements the `net.jini.lookup.entry.EntryBean` interface.
- There is a one-to-one link between a JavaBeans component and a particular entry object. The `makeLink` method establishes this link and will throw an exception if the association is with an entry class of the wrong type. The `followLink` method returns the entry object associated with a particular JavaBeans component.
- The no-arg public constructor for a JavaBeans component creates and makes a link to an empty entry object.
- For each public object field *`foo`* in an entry class, there exist both a `set`*`Foo`* and a `getFoo` method in the associated JavaBeans component. The `set`*`Foo`* method takes a single argument of the same type as the *`foo`* field in the associated entry and sets the value of that field to its argument. The `get`*`Foo`* method returns the value of that field.

LS.3.3 Supporting Interfaces and Classes

The following classes and interfaces provide facilities for handling entry classes and their associated JavaBeans components.

```
package net.jini.lookup.entry;

public class EntryBeans {
    public static EntryBean createBean(Entry e)
        throws ClassNotFoundException, java.io.IOException {…}
```

```
    public static Class getBeanClass(Class c)
        throws ClassNotFoundException {…}
}

public interface EntryBean {
    void makeLink(Entry e);
    Entry followLink();
}
```

The `EntryBeans` class cannot be instantiated. Its sole method, `createBean`, creates and initializes a new JavaBeans component and links it to the entry object it is passed. If a problem occurs creating the JavaBeans component, the method throws either `java.io.IOException` or `ClassNotFoundException`.

The `createBean` method uses the same mechanism for instantiating a JavaBeans component as the `java.beans.Beans.instantiate` method. It will initially try to instantiate the JavaBeans component using the same class loader as the entry it is passed. If that fails, it will fall back to using the default class loader.

The `getBeanClass` method returns the class of the JavaBeans component associated with the given attribute class. If the class passed in does not implement the `net.jini.core.entry.Entry` interface, an `IllegalArgumentException` is thrown. If the given attribute class cannot be found, a `ClassNotFoundException` is thrown.

The `EntryBean` interface must be implemented by all JavaBeans components that are intended to be linked to entry objects. The `makeLink` method establishes a link between a JavaBeans component object and an entry object, and the `followLink` method returns the entry object linked to by a particular JavaBeans component. Note that objects that implement the `EntryBean` interface should not be assumed to perform any internal synchronization in their implementations of the `makeLink` or `followLink` methods, or in the `setFoo` or `getFoo` patterns.

LS.4 Generic Attribute Classes

We will now describe some attribute classes that are generic to many or all services, and the JavaBeans components that are associated with each. Unless otherwise stated, all classes defined here live in the `net.jini.lookup.entry` package. The definitions assume the following classes to have been imported:

```
java.io.Serializable
net.jini.entry.AbstractEntry
```

LS.4.1 Indicating User Modifiability

To indicate that certain entry classes should only be modified by the service that registered itself with instances of these entry classes, we annotate them with the `ServiceControlled` interface.

```
public interface ServiceControlled {
}
```

Authors of administrative tools that modify fields of attribute objects at the lookup service should not permit users to either modify any fields or add any new instances of objects that implement this interface.

LS.4.2 Basic Service Information

The `ServiceInfo` attribute class provides some basic information about a service.

```
public class ServiceInfo extends AbstractEntry
    implements ServiceControlled
{
    public ServiceInfo() {…}
    public ServiceInfo(String name, String manufacturer,
                       String vendor, String version,
                       String model, String serialNumber) {…}
```

```
    public String name;
    public String manufacturer;
    public String vendor;
    public String version;
    public String model;
    public String serialNumber;
}

public class ServiceInfoBean
    implements EntryBean, Serializable
{
    public String getName() {…}
    public void setName(String s) {…}
    public String getManufacturer() {…}
    public void setManufacturer(String s) {…}
    public String getVendor() {…}
    public void setVendor(String s) {…}
    public String getVersion() {…}
    public void setVersion(String s) {…}
    public String getModel() {…}
    public void setModel(String s) {…}
    public String getSerialNumber() {…}
    public void setSerialNumber(String s) {…}
}
```

Each service should register itself with only one instance of this class. The fields of the `ServiceInfo` class have the following meanings:

- The `name` field contains a specific product name, such as `"Ultra 30"` (for a particular workstation) or `"JavaSafe"` (for a specific configuration management service). This string should not include the name of the manufacturer or vendor.
- The `manufacturer` field provides the name of the company that "built" this service. This might be a hardware manufacturer or a software authoring company.
- The `vendor` field contains the name of the company that sells the software or hardware that provides this service. This may be the same name as is in the `manufacturer` field, or it could be the name of a reseller. This field exists so that in cases in which resellers relabel products built by other companies, users will be able to search based on either name.

- The `version` field provides information about the version of this service. It is a free-form field, though we expect that service implementors will follow normal version-naming conventions in using it.
- The `model` field contains the specific model name or number of the product, if any.
- The `serialNumber` field provides the serial number of this instance of the service, if any.

LS.4.3 More Specific Information

The `ServiceType` class allows an author of a service to deliver information that is specific to a particular instance of a service, rather than to services in general.

```
public class ServiceType extends AbstractEntry
        implements ServiceControlled
{
    public ServiceType() {…}
    public java.awt.Image getIcon(int iconKind) {…}
    public String getDisplayName() {…}
    public String getShortDescription() {…}
}
```

Each service may register itself with multiple instances of this class, usually with one instance for each type of service interface it implements.

This class has no public fields and, as a result, has no associated JavaBeans component.

The `getIcon` method returns an icon of the appropriate kind for the service; it works in the same way as the `getIcon` method in the `java.beans.BeanInfo` interface, with the value of `iconKind` being taken from the possibilities defined in that interface. The `getDisplayName` and `getShortDescription` methods return a localized human-readable name and description for the service, in the same manner as their counterparts in the `java.beans.FeatureDescriptor` class. Each of these methods returns `null` if no information of the appropriate kind is defined.

In case the distinction between the information this class provides and that provided by a JavaBeans component's metainformation is unclear, the class `ServiceType` is meant to be used in the lookup service as one of the entry classes with which a service registers itself, and so it can be customized on a per-service basis. By contrast, the `FeatureDescriptor` and `BeanInfo` objects for all `EntryBean` classes provide only generic information about those classes and none about specific instances of those classes.

LS.4.4 Naming a Service

People like to associate names with particular services and may do so using the Name class.

```
public class Name extends AbstractEntry {
    public Name() {...}
    public Name(String name) {...}

    public String name;
}

public class NameBean implements EntryBean, Serializable {
    public String getName() {...}
    public void setName(String s) {...}
}
```

Services may register themselves with multiple instances of this class, and either services or administrators may add, modify, or remove instances of this class from the attribute set under which a service is registered.

The name field provides a short name for a particular instance of a service (for example, "Bob's toaster").

LS.4.5 Adding a Comment to a Service

In cases in which some kind of comment is appropriate for a service (for example, "this toaster tends to burn bagels"), the Comment class provides an appropriate facility.

```
public class Comment extends AbstractEntry {
    public Comment() {...}
    public Comment(String comment) {...}

    public String comment;
}

public class CommentBean implements EntryBean, Serializable {
    public String getComment() {...}
    public void setComment(String s) {...}
}
```

A service may have more than one comment associated with it, and comments may be added, removed, or edited by either a service itself, administrators, or users.

LS.4.6 Physical Location

The `Location` and `Address` classes provide information about the physical location of a particular service.

Since many services have no physical location, some have one, and a few may have more than one, it might make sense for a service to register itself with zero or more instances of either of these classes, depending on its nature.

The `Location` class is intended to provide information about the physical location of a service in a single building or on a small, unified campus. The `Address` class provides more information and may be appropriate for use with the `Location` class in a larger, more geographically distributed organization.

```
public class Location extends AbstractEntry {
    public Location() {…}
    public Location(String floor, String room,
                    String building) {…}

    public String floor;
    public String room;
    public String building;
}

public class LocationBean implements EntryBean, Serializable {
    public String getFloor() {…}
    public void setFloor(String s) {…}
    public String getRoom() {…}
    public void setRoom(String s) {…}
    public String getBuilding() {…}
    public void setBuilding(String s) {…}
}

public class Address extends AbstractEntry {
    public Address() {…}
    public Address(String street, String organization,
                   String organizationalUnit, String locality,
                   String stateOrProvince, String postalCode,
```

```
                    String country) {…}

    public String street;
    public String organization;
    public String organizationalUnit;
    public String locality;
    public String stateOrProvince;
    public String postalCode;
    public String country;
}

public class AddressBean implements EntryBean, Serializable {
    public String getStreet() {…}
    public void setStreet(String s) {…}
    public String getOrganization() {…}
    public void setOrganization(String s) {…}
    public String getOrganizationalUnit() {…}
    public void setOrganizationalUnit(String s) {…}
    public String getLocality() {…}
    public void setLocality(String s) {…}
    public String getStateOrProvince() {…}
    public void setStateOrProvince(String s) {…}
    public String getPostalCode() {…}
    public void setPostalCode(String s) {…}
    public String getCountry() {…}
    public void setCountry(String s) {…}
}
```

We believe the fields of these classes to be self-explanatory, with the possible exception of the `locality` field of the `Address` class, which would typically hold the name of a city.

LS.4.7 Status Information

Some attributes of a service may constitute long-lived status, such as an indication that a printer is out of paper. We provide a class, `Status`, that implementors can use as a base for providing status-related entry classes.

```
public abstract class Status extends AbstractEntry {
    protected Status() {…}
    protected Status(StatusType severity) {…}
```

```
    public StatusType severity;
}

public class StatusType implements Serializable {
    private final int type;
    private StatusType(int t) { type = t; }
    public static final StatusType ERROR =  new StatusType(1);
    public static final StatusType WARNING =
                                            new StatusType(2);
    public static final StatusType NOTICE = new StatusType(3);
    public static final StatusType NORMAL = new StatusType(4);
}

public abstract class StatusBean
    implements EntryBean, Serializable
{
    public StatusType getSeverity() {...}
    public void setSeverity(StatusType i) {...}
}
```

We define a separate `StatusType` class to make it possible to write a property editor that will work with the `StatusBean` class (we do not currently provide a property editor implementation).

LS.4.8 Serialized Forms

Class	serialVersionUID	Serialized Fields
Address	2896136903322046578L	*all public fields*
AddressBean	4491500432084550577L	Address asoc
Comment	7138608904371928208L	*all public fields*
CommentBean	5272583409036504625L	Comment asoc
Location	–3275276677967431315L	*all public fields*
LocationBean	–4182591284470292829L	Location asoc
Name	27432151480713072O1L	*all public fields*
NameBean	–6026791845102735793L	Name asoc
ServiceInfo	–1116664185758541509L	*all public fields*

Class	serialVersionUID	Serialized Fields
ServiceInfoBean	8352546663361067804L	ServiceInfo asoc
ServiceType	–6443809721367395836L	*all public fields*
Status	–5193075846115040838L	*all public fields*
StatusBean	–1975539395914887503L	Status asoc
StatusType	–8268735508512712203L	int type

Lookup Schema (LS)

THE JAVASPACES SPECIFICATION describes the JavaSpaces service defined in the package `net.jini.javaSpace`. *A JavaSpaces service provides a simple yet powerful persistent coordination tool for transactionally governed cooperation between loosely coupled players in distributed protocols.*

JS

The JavaSpaces Specification

JS.1 Introduction

DISTRIBUTED systems are hard to build. They require careful thinking about problems that do not occur in local computation. The primary problems are those of partial failure, greatly increased latency, and language compatibility. The Java programming language has a remote method invocation system called RMI that lets you approach general distributed computation in the Java programming language using techniques natural to the Java programming language and application environment. This is layered on the Java platform's object serialization mechanism to marshal parameters of remote methods into a form that can be shipped across the wire and unmarshalled in a remote server's Java virtual machine (JVM).

This specification describes the architecture of JavaSpaces technology, which is designed to help you solve two related problems: distributed persistence and the design of distributed algorithms. JavaSpaces services use RMI and the serialization feature of the Java programming language to accomplish these goals.

JS.1.1 The JavaSpaces Application Model and Terms

A JavaSpaces service holds *entries*. An entry is a typed group of objects, expressed in a class for the Java platform that implements the interface `net.jini.core.entry.Entry`. Entries are described in detail in the *Jini Entry Specification.*

An entry can be *written* into a JavaSpaces service, which creates a copy of that entry in the space[1] that can be used in future lookup operations.

[1] The term "space" is used to refer to a JavaSpaces service implementation.

You can look up entries in a JavaSpaces service using *templates,* which are entry objects that have some or all of its fields set to specified *values* that must be matched exactly. Remaining fields are left as *wildcards*—these fields are not used in the lookup.

There are two kinds of lookup operations: *read* and *take*. A *read* request to a space returns either an entry that matches the template on which the read is done, or an indication that no match was found. A *take* request operates like a read, but if a match is found, the matching entry is removed from the space.

You can request a JavaSpaces service to *notify* you when an entry that matches a specified template is written. This is done using the distributed event model contained in the package `net.jini.core.event` and described in the *Jini Distributed Event Specification.*

All operations that modify a JavaSpaces service are performed in a transactionally secure manner with respect to that space. That is, if a write operation returns successfully, that entry was written into the space (although an intervening take may remove it from the space before a subsequent lookup of yours). And if a take operation returns an entry, that entry has been removed from the space, and no future operation will read or take the same entry. In other words, each entry in the space can be taken at most once. Note, however, that two or more entries in a space may have exactly the same value.

The architecture of JavaSpaces technology supports a simple transaction mechanism that allows multi-operation and/or multi-space updates to complete atomically. This is done using the two-phase commit model under the default transaction semantics, as defined in the package `net.jini.core.transaction` and described in the *Jini Transaction Specification.*

Entries written into a JavaSpaces service are governed by a lease, as defined in the package `net.jini.core.lease` and described in the *Jini Distributed Lease Specification.*

JS.1.1.1 Distributed Persistence

Implementations of JavaSpaces technology provide a mechanism for storing a group of related objects and retrieving them based on a value-matching lookup for specified fields. This allows a JavaSpaces service to be used to store and retrieve objects on a remote system.

JS.1.1.2 Distributed Algorithms as Flows of Objects

Many distributed algorithms can be modeled as a flow of objects between participants. This is different from the traditional way of approaching distributed com-

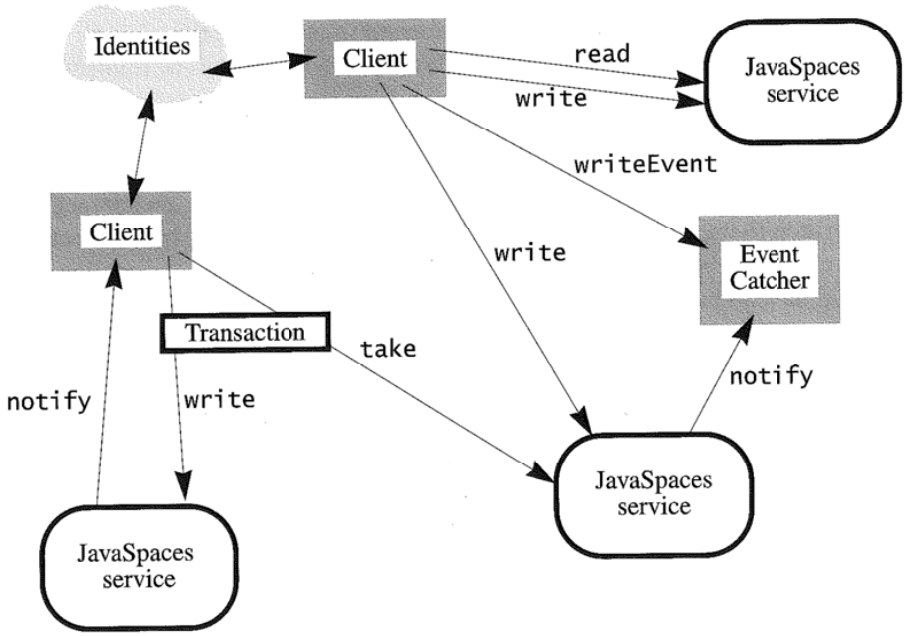
puting, which is to create method-invocation-style protocols between participants. In this architecture's "flow of objects" approach, protocols are based on the movement of objects into and out of implementations of JavaSpaces technology.

For example, a book-ordering system might look like this:

- A book buyer wants to buy 100 copies of a book. The buyer writes a request for bids into a particular public JavaSpaces service.
- The broker runs a server that takes those requests out of the space and writes them into a JavaSpaces service for each book seller who registered with the broker for that service.
- A server at each book seller removes the requests from its JavaSpaces service, presents the request to a human to prepare a bid, and writes the bid into the space specified in the book buyer's request for bids.
- When the bidding period closes, the buyer takes all the bids from the space and presents them to a human to select the winning bid.

A method-invocation-style design would create particular remote interfaces for these interactions. With a "flow of objects" approach, only one interface is required: the `net.jini.space.JavaSpace` interface.

In general, the JavaSpaces application world looks like this:



Clients perform operations that map entries or templates onto JavaSpaces services. These can be singleton operations (as with the upper client), or contained in transactions (as with the lower client) so that all or none of the operations take place. A single client can interact with as many spaces as it needs to. Identities are accessed from the security subsystem and passed as parameters to method invocations. Notifications go to event catchers, which may be clients themselves or proxies for a client (such as a store-and-forward mailbox).

JS.1.2 Benefits

JavaSpaces services are tools for building distributed protocols. They are designed to work with applications that can model themselves as flows of objects through one or more servers. If your application can be modeled this way, JavaSpaces technology will provide many benefits.

JavaSpaces services can provide a reliable distributed storage system for the objects. In the book-buying example, the designer of the system had to define the protocol for the participants and design the various kinds of entries that must be passed around. This effort is akin to designing the remote interfaces that an equivalent customized service would require. Both the JavaSpaces system solution and the customized solution would require someone to write the code that presented requests and bids to humans in a GUI. And in both systems, someone would have to write code to handle the seller's registrations of interest with the broker.

The server for the model that uses the JavaSpaces API would be implemented at that point.

The customized system would need to implement the servers. These servers would have to handle concurrent access from multiple clients. Someone would need to design and implement a reliable storage strategy that guaranteed the entries written to the server would not be lost in an unrecoverable or undetectable way. If multiple bids needed to be made atomically, a distributed transaction system would have to be implemented.

All these concerns are solved in JavaSpaces services. They handle concurrent access. They store and retrieve entries atomically. And they provide an implementation of the distributed transaction mechanism.

This is the power of the JavaSpaces technology architecture—many common needs are addressed in a simple platform that can be easily understood and used in powerful ways.

JavaSpaces services also help with data that would traditionally be stored in a file system, such as user preferences, e-mail messages, and images. Actually, this is not a different use of a JavaSpaces service. Such uses of a file system can equally be viewed as passing objects that contain state from one external object

(the image editor) to another (the window system that uses the image as a screen background). And JavaSpaces services enhance this functionality because they store objects, not just data, so the image can have abstract behavior, not just information that must be interpreted by some external application(s).

JavaSpaces services can provide distributed *object* persistence with objects in the Java programming language. Because code written in the Java programming language is downloadable, entries can store objects whose behavior will be transmitted from the writer to the readers, just as in an RMI using Java technology. An entry in a space may, when fetched, cause some active behavior in the reading client. This is the benefit of storing objects, not just data, in an accessible repository for distributed cooperative computing.

JS.1.3 JavaSpaces Technology and Databases

A JavaSpaces service can store persistent data which is later searchable. But a JavaSpaces service is not a relational or object database. JavaSpaces services are designed to help solve problems in distributed computing, not to be used primarily as a data repository (although there are many data storage uses for JavaSpaces applications). Some important differences are:

- Relational databases understand the data they store and manipulate it directly via query languages. JavaSpaces services store entries that they understand only by type and the serialized form of each field. There are no general queries in the JavaSpaces application design, only "exact match" or "don't care" for a given field. You design your flow of objects so that this is sufficient and powerful.
- Object databases provide an object oriented image of stored data that can be modified and used, nearly as if it were transient memory. JavaSpaces systems do not provide a nearly transparent persistent/transient layer, and work only on copies of entries.

These differences exist because JavaSpaces services are designed for a different purpose than either relational or object databases. A JavaSpaces service can be used for simple persistent storage, such as storing a user's preferences that can be looked up by the user's ID or name. JavaSpaces service functionality is somewhere between that of a filesystem and a database, but it is neither.

JS.1.4 JavaSpaces System Design and Linda[2] Systems

The JavaSpaces system design is strongly influenced by Linda systems, which support a similar model of entry-based shared concurrent processing. In Section JS.4.1 you will find several references that describe Linda-style systems.

No knowledge of Linda systems is required to understand this specification. This section discusses the relationship of JavaSpaces systems with respect to Linda systems for the benefit of those already familiar with Linda programming. Other readers should feel free to skip ahead.

JavaSpaces systems are similar to Linda systems in that they store collections of information for future computation and are driven by value-based lookup. They differ in some important ways:

- Linda systems have not used rich typing. JavaSpaces systems take a deep concern with typing from the Java platform type-safe environment. In JavaSpaces systems, entries themselves, not just their fields, are typed—two different entries with the same field types but with different data types for the Java programming language are different entry types. For example, an entry that had a string and two double values could be either a named point or a named vector. In JavaSpaces systems these two entry types would have specific different classes for the Java platform, and templates for one type would never match the other, even if the values were compatible.
- Entries are typed as objects in the Java programming language, so they may have methods associated with them. This provides a way of associating behavior with entries.
- As another result of typed entries, JavaSpaces services allow matching of subtypes—a template match can return a type that is a subtype of the template type. This means that the read or take may return more states than anticipated. In combination with the previous point, this means that entry behavior can be polymorphic in the usual object-oriented style that the Java platform provides.
- The fields of entries are objects in the Java programming language. Any object data type for the Java programming language can be used as a template for matching entry lookups as long as it has certain properties. This means that computing systems constructed using the JavaSpaces API are

[2] "Linda" is the name of a public domain technology originally propounded by Dr. David Gelernter of Yale University. "Linda" is also claimed as a trademark for certain goods by Scientific Computing Associates, Inc. This discussion refers to the public domain "Linda" technology.

object-oriented from top to bottom, and behavior-based (agent-like) applications can use JavaSpaces services for co-ordination.

- Most environments will have more than one JavaSpaces service. Most Linda tuple spaces have one tuple space for all cooperating threads. So transactions in the JavaSpaces system can span multiple spaces (and even non-JavaSpaces system transaction participants).
- Entries written into a JavaSpaces service are leased. This helps keep the space free of debris left behind due to system crashes and network failures.
- The JavaSpaces API does not provide an equivalent of "eval" because it would require the service to execute arbitrary computation on behalf of the client. Such a general compute service has its own large number of requirements (such as security and fairness).

On the nomenclature side, the JavaSpaces technology API uses a more accessible set of terms than the traditional Linda terms. The term mappings are "entry" for "tuple", "value" for "actual", "wildcard" for "formal", "write" for "out", and "take" for "in". So the Linda sentence "When you 'out' a tuple make sure that actuals and formals in 'in' and 'read' can do appropriate matching" would be translated to "When you write an entry make sure that values and wildcards in 'take' and 'read' can do appropriate matching."

JS.1.5 Goals and Requirements

The goals for the design of JavaSpaces technology are:

- Provide a platform for designing distributed computing systems that simplifies the design and implementation of those systems.
- The client side should have few classes, both to keep the client-side model simple and to make downloading of the client classes quick.
- The client side should have a small footprint, because it will run on computers with limited local memory.
- A variety of implementations should be possible, including relational database storage and object-oriented database storage.
- It should be possible to create a replicated JavaSpaces service.

The requirements for JavaSpaces application clients are:

- It must be possible to write a client purely in the Java programming language.
- Clients must be oblivious to the implementation details of the service. The same entries and templates must work in the same ways no matter which implementation is used.

JS.1.6 Dependencies

This document relies upon the following other specifications:

- *Java Remote Method Invocation Specification*
- *Java Object Serialization Specification*
- *Jini Entry Specification*
- *Jini Entry Utilities Specification*
- *Jini Distributed Event Specification*
- *Jini Distributed Leasing Specification*
- *Jini Transaction Specification*

JS.2 Operations

THERE are four primary kinds of operations that you can invoke on a JavaSpaces service. Each operation has parameters that are entries, including some that are templates, which are a kind of entry. This chapter describes entries, templates, and the details of the operations, which are:

- `write`: Write the given entry into this JavaSpaces service.
- `read`: Read an entry from this JavaSpaces service that matches the given template.
- `take`: Read an entry from this JavaSpaces service that matches the given template, removing it from this space.
- `notify`: Notify a specified object when entries that match the given template are written into this JavaSpaces service.

As used in this document, the term "operation" refers to a single invocation of a method; for example, two different `take` operations may have different templates.

JS.2.1 Entries

The types `Entry` and `UnusableEntryException` that are used in this specification are from the package `net.jini.core.entry` and are described in detail in the *Jini Entry Specification.* In the terminology of that specification `write` is a store operation; `read` and `take` are combination search and fetch operations; and `notify` sets up repeated search operations as entries are written to the space.

JS.2.2 net.jini.space.JavaSpace

All operations are invoked on an object that implements the JavaSpace interface. For example, the following code fragment would write an entry of type AttrEntry into the JavaSpaces service referred to by the identifier space:

```
JavaSpace space = getSpace();
AttrEntry e = new AttrEntry();
e.name = "Duke";
e.value = new GIFImage("dukeWave.gif");
space.write(e, null, 60 * 60 * 1000);// one hour
// lease is ignored -- one hour will be enough
```

The JavaSpace interface is:

```
package net.jini.space;

import java.rmi.*;
import net.jini.core.event.*;
import net.jini.core.transaction.*;
import net.jini.core.lease.*;

public interface JavaSpace {
    Lease write(Entry e, Transaction txn, long lease)
        throws RemoteException, TransactionException;
    public final long NO_WAIT = 0; // don't wait at all
    Entry read(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry readIfExists(Entry tmpl, Transaction txn,
                       long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry take(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry takeIfExists(Entry tmpl, Transaction txn,
                       long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    EventRegistration notify(Entry tmpl, Transaction txn,
              RemoteEventListener listener, long lease,
```

```
                MarshalledObject handback)
            throws RemoteException, TransactionException;
        Entry snapshot(Entry e) throws RemoteException;
    }
```

The `Transaction` and `TransactionException` types in the above signatures are imported from `net.jini.core.transaction`. The `Lease` type is imported from `net.jini.core.lease`. The `RemoteEventListener` and `EventRegistration` types are imported from `net.jini.core.event`.

In all methods that have the parameter, `txn` may be `null`, which means that no `Transaction` object is managing the operation (see Section JS.3).

The `JavaSpace` interface is not a remote interface. Each implementation of a JavaSpaces service exports proxy objects that implement the `JavaSpace` interface locally on the client, talking to the actual JavaSpaces service through an implementation-specific interface. An implementation of any `JavaSpace` method may communicate with a remote JavaSpaces service to accomplish its goal; hence, each method throws `RemoteException` to allow for possible failures. Unless noted otherwise in this specification, when you invoke `JavaSpace` methods you should expect `RemoteExceptions` on method calls in the same cases in which you would expect them for methods invoked directly on an RMI remote reference. For example, invoking `snapshot` might require talking to the remote JavaSpaces server, and so might get a `RemoteException` if the server crashes during the operation.

The details of each `JavaSpace` method are given in the sections that follow.

JS.2.2.1 `InternalSpaceException`

The exception `InternalSpaceException` may be thrown by a JavaSpaces service that encounters an inconsistency in its own internal state or is unable to process a request because of internal limitations (such as storage space being exhausted). This exception is a subclass of `RuntimeException`. The exception has two constructors: one that takes a `String` description and another that takes a `String` and a nested exception; both constructors simply invoke the `RuntimeException` constructor that takes a `String` argument.

```
package net.jini.space;

public class InternalSpaceException extends RuntimeException {
    public final Throwable nestedException;
    public InternalSpaceException(String msg) {…}
    public InternalSpaceException(String msg, Throwable e) {…}
```

```
    public printStackTrace() {…}
    public printStackTrace(PrintStream out) {…}
    public printStackTrace(PrintWriter out) {…}
}
```

The `nestedException` field is the one passed to the second constructor, or `null` if the first constructor was used. The overridden `printStackTrace` methods print out the stack trace of the exception and, if `nestedException` is not `null`, print out that stack trace as well.

JS.2.3 `write`

A `write` places a copy of an entry into the given JavaSpaces service. The `Entry` passed to the `write` is not affected by the operation. Each `write` operation places a new entry into the specified space, even if the same `Entry` object is used in more than one `write`.

Each `write` invocation returns a `Lease` object that is `lease` milliseconds long. If the requested time is longer than the space is willing to grant, you will get a lease with a reduced time. When the lease expires, the entry is removed from the space. You will get an `IllegalArgumentException` if the lease time requested is negative.

If a `write` returns without throwing an exception, that entry is committed to the space, possibly within a transaction (see Section JS.3). If a `RemoteException` is thrown, the `write` may or may not have been successful. If any other exception is thrown, the entry was not written into the space.

Writing an entry into a space might generate notifications to registered objects (see Section JS.2.7).

JS.2.4 `readIfExists` and `read`

The two forms of the `read` request search the JavaSpaces service for an entry that matches the template provided as an `Entry`. If a match is found, a reference to a copy of the matching entry is returned. If no match is found, `null` is returned. Passing a `null` reference for the template will match any entry.

Any matching entry can be returned. Successive read requests with the same template in the same JavaSpaces service may or may not return equivalent objects, even if no intervening modifications have been made to the space. Each invocation of read may return a new object even if the same entry is matched in the JavaSpaces service.

A `readIfExists` request will return a matching entry, or `null` if there is currently no matching entry in the space. If the only possible matches for the template have conflicting locks from one or more other transactions, the `timeout` value specifies how long the client is willing to wait for interfering transactions to settle before returning a value. If at the end of that time no value can be returned that would not interfere with transactional state, `null` is returned. Note that, due to the remote nature of JavaSpaces services, `read` and `readIfExists` may throw a `RemoteException` if the network or server fails prior to the timeout expiration

A `read` request acts like a `readIfExists` except that it will wait until a matching entry is found or until transactions settle, whichever is longer, up to the timeout period.

In both read methods, a timeout of `NO_WAIT` means to return immediately, with no waiting, which is equivalent to using a zero timeout.

JS.2.5 `takeIfExists` and `take`

The `take` requests perform exactly like the corresponding `read` requests (see Section JS.2.4), except that the matching entry is removed from the space. Two `take` operations will never return copies of the same entry, although if two equivalent entries were in the JavaSpaces service the two `take` operations could return equivalent entries.

If a `take` returns a non-`null` value, the entry has been removed from the space, possibly within a transaction (see Section JS.3). This modifies the claims to once-only retrieval: A `take` is considered to be successful only if all enclosing transactions commit successfully. If a `RemoteException` is thrown, the `take` may or may not have been successful. If an `UnusableEntryException` is thrown, the `take` removed the unusable entry from the space; the contents of the exception are as described in the *Jini Entry Specification.* If any other exception is thrown, the `take` did not occur, and no entry was removed from the space.

With a `RemoteException`, an entry can be removed from a space and yet never returned to the client that performed the `take`, thus losing the entry in between. In circumstances in which this is unacceptable, the `take` can be wrapped inside a transaction that is committed by the client when it has the requested entry in hand.

JS.2.6 `snapshot`

The process of serializing an entry for transmission to a JavaSpaces service will be identical if the same entry is used twice. This is most likely to be an issue with

templates that are used repeatedly to search for entries with `read` or `take`. The client-side implementations of `read` and `take` cannot reasonably avoid this duplicated effort, since they have no efficient way of checking whether the same template is being used without intervening modification.

The `snapshot` method gives the JavaSpaces service implementor a way to reduce the impact of repeated use of the same entry. Invoking `snapshot` with an `Entry` will return another `Entry` object that contains a *snapshot* of the original entry. Using the returned snapshot entry is equivalent to using the unmodified original entry in all operations on the same JavaSpaces service. Modifications to the original entry will not affect the snapshot. You can `snapshot` a `null` template; `snapshot` may or may not return `null` given a `null` template.

The entry returned from `snapshot` will be guaranteed equivalent to the original unmodified object only when used with the space. Using the snapshot with any other JavaSpaces service will generate an `IllegalArgumentException` unless the other space can use it because of knowledge about the JavaSpaces service that generated the snapshot. The snapshot will be a different object from the original, may or may not have the same hash code, and `equals` may or may not return `true` when invoked with the original object, even if the original object is unmodified.

A snapshot is guaranteed to work only within the virtual machine in which it was generated. If a snapshot is passed to another virtual machine (for example, in a parameter of an RMI call), using it—even with the same JavaSpaces service—may generate an `IllegalArgumentException`.

We expect that an implementation of JavaSpaces technology will return a specialized `Entry` object that represents a pre-serialized version of the object, either in the object itself or as an identifier for the entry that has been cached on the server. Although the client may cache the snapshot on the server, it must guarantee that the snapshot returned to the client code is always valid. The implementation may not throw any exception that indicates that the snapshot has become invalid because it has been evicted from a cache. An implementation that uses a server-side cache must therefore guarantee that the snapshot is valid as long as it is reachable (not garbage) in the client, such as by storing enough information in the client to be able to re-insert the snapshot into the server-side cache.

No other method returns a snapshot. Specifically, the return values of the `read` and `take` methods are not snapshots and are usable with any implementation of JavaSpaces technology.

JS.2.7 `notify`

A `notify` request registers interest in future incoming entries to the JavaSpaces service that match the specified template. Matching is done as it is for `read`. The

`notify` method is a particular registration method under the *Jini Distributed Event Specification.* When matching entries are written, the specified `RemoteEventListener` will eventually be notified. When you invoke `notify` you provide an upper bound on the lease time, which is how long you want the registration to be remembered by the JavaSpaces service. The service decides the actual time for the lease. You will get an `IllegalArgumentException` if the lease time requested is not `Lease.ANY` and is negative. The lease time is expressed in the standard millisecond units, although actual lease times will usually be of much larger granularity. A lease time of `Lease.FOREVER` is a request for an indefinite lease; if the service chooses not to grant an indefinite lease, it will return a bounded (non-zero) lease.

Each `notify` returns a `net.jini.core.event.EventRegistration` object. When an object is written that matches the template supplied in the `notify` invocation, the listener's `notify` method is eventually invoked, with a `RemoteEvent` object whose `evID` is the value returned by the `EventRegistration` object's `getEventID` method, `fromWhom` being the JavaSpaces service, `seqNo` being a monotonically increasing number, and whose `getRegistrationObject` being that passed as the `handback` parameter to `notify`. If you get a notification with a sequence number of 103 and the `EventRegID` object's current sequence number is 100, there will have been three matching entries written since you invoked `notify`. You may or may not have received notification of the previous entries due to network failures or the space compressing multiple matching entry events into a single call.

If the transaction parameter is `null`, the listener will be notified when matching entries are written either under a `null` transaction or when a transaction commits. If an entry is written under a transaction and then taken under that same transaction before the transaction is committed, listeners registered under a `null` transaction will not be notified of that entry.

If the transaction parameter is not `null`, the listener will be notified of matching entries written under that transaction in addition to the notifications it would receive under a `null` transaction. A `notify` made with a non-`null` transaction is implicitly dropped when the transaction completes.

The request specified by a successful `notify` is as persistent as the entries of the space. They will be remembered as long as an untaken entry would be, until the lease expires, or until any governing transaction completes, whichever is shorter.

The service will make a "best effort" attempt to deliver notifications. The service will retry at most until the notification request's lease expires. Notifications may be delivered in any order.

See the *Jini Distributed Event Specification* for details on the event types.

JS.2.8 Operation Ordering

Operations on a space are unordered. The only view of operation order can be a thread's view of the order of the operations it performs. A view of inter-thread order can be imposed only by cooperating threads that use an application-specific protocol to prevent two or more operations being in progress at a single time on a single JavaSpaces service. Such means are outside the purview of this specification.

For example, given two threads *T* and *U*, if *T* performs a `write` operation and *U* performs a `read` with a template that would match the written entry, the `read` may not find the written entry even if the `write` returns before the `read`. Only if *T* and *U* cooperate to ensure that the `write` returns before the `read` commences would the `read` be ensured the opportunity to find the entry written by *T* (although it still might not do so because of an intervening `take` from a third entity).

JS.2.9 Serialized Form

Class	`serialVersionUID`	Serialized Fields
`InternalSpaceException`	-4167507833172939849L	*all public fields*

JS.3 Transactions

THE JavaSpaces API uses the package `net.jini.core.transaction` to provide basic atomic transactions that group multiple operations across multiple JavaSpaces services into a bundle that acts as a single atomic operation. JavaSpaces services are actors in these transactions; the client can be an actor as well, as can any remote object that implements the appropriate interfaces.

Transactions wrap together multiple operations. Either all modifications within the transactions will be applied or none will, whether the transaction spans one or more operations and/or one or more JavaSpaces services.

The transaction semantics described here conform to the default transaction semantics defined in the *Jini Transaction Specification.*

JS.3.1 Operations under Transactions

Any `read`, `write`, or `take` operations that have a `null` transaction act as if they were in a committed transaction that contained exactly that operation. For example, a `take` with a `null` transaction parameter performs as if a transaction was created, the `take` performed under that transaction, and then the transaction was committed. Any `notify` operations with a `null` transaction apply to `write` operations that are committed to the entire space.

Transactions affect operations in the following ways:

- `write`: An entry that is written is not visible outside its transaction until the transaction successfully commits. If the entry is taken within the transaction, the entry will never be visible outside the transaction and will not be added to the space when the transaction commits. Specifically, the entry will not generate notifications to listeners that are not registered under the writing transaction. Entries written under a transaction that aborts are discarded.
- `read`: A `read` may match any entry written under that transaction or in the entire space. A JavaSpaces service is not required to prefer matching entries written inside the transaction to those in the entire space. When read, an

entry is added to the set of entries read by the provided transaction. Such an entry may be read in any other transaction to which the entry is visible, but cannot be taken in another transaction.

- `take`: A `take` matches like a `read` with the same template. When taken, an entry is added to the set of entries taken by the provided transaction. Such an entry may not be read or taken by any other transaction.
- `notify`: A `notify` performed under a `null` transaction applies to `write` operations that are committed to the entire space. A `notify` performed under a non-`null` transaction additionally provides notification of writes performed within that transaction. When a transaction completes, any registrations under that transaction are implicitly dropped. When a transaction commits, any entries that were written under the transaction (and not taken) will cause appropriate notifications for registrations that were made under a `null` transaction.

If a transaction aborts while an operation is in progress under that transaction, the operation will terminate with a `TransactionException`. Any statement made in this chapter about `read` or `take` apply equally to `readIfExists` or `takeIfExists`, respectively.

JS.3.2 Transactions and ACID Properties

The ACID properties traditionally offered by database transactions are preserved in transactions on JavaSpaces systems. The ACID properties are:

- *Atomicity:* All the operations grouped under a transaction occur or none of them do.
- *Consistency:* The completion of a transaction must leave the system in a consistent state. Consistency includes issues known only to humans, such as that an employee should always have a manager. The enforcement of consistency is outside of the transaction—a transaction is a tool to allow consistency guarantees, and not itself a guarantor of consistency.
- *Isolation:* Ongoing transactions should not affect each other. Any observer should be able to see other transactions executing in some sequential order (although different observers may see different orders).
- *Durability:* The results of a transaction should be as persistent as the entity on which the transaction commits.

The timeout values in `read` and `take` allow a client to trade full isolation for liveness. For example, if a `read` request has only one matching entry and that entry is currently locked in a `take` from another transaction, `read` would block indefinitely if the client wanted to preserve isolation. Since completing the transaction could take an indefinite amount of time, a client may choose instead to put an upper bound on how long it is willing to wait for such isolation guarantees, and instead proceed to either abort its own transaction or ask the user whether to continue or whatever else is appropriate for the client.

Persistence is not a required property of JavaSpaces technology implementations. A transient implementation that does not preserve its contents between system crashes is a proper implementation of the `JavaSpace` interface's contract, and may be quite useful. If you choose to perform operations on such a space, your transactions will guarantee as much durability as the JavaSpaces service allows for all its data, which is all that any transaction system can guarantee.

JS.4 Further Reading

JS.4.1 Linda Systems

1. How to Write Parallel Programs: A Guide to the Perplexed, Nicholas Carriero and David Gelernter, *ACM Computing Surveys*, Sept., 1989.
2. Generative Communication in Linda, David Gelernter, *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 1, pp. 80–112 (January 1985).
3. Persistent Linda: Linda + Transactions + Query Processing, Brian G. Anderson and Dennis Shasha, *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems,* 1994.
4. Adding Fault-tolerant Transaction Processing to LINDA, Scott R. Cannon and David Dunn, *Software—Practice and Experience,* Vol. 24(5), pp. 449–446 (May 1994).
5. *ActorSpaces: An Open Distributed Programming Paradigm,* Gul Agha, Christian J. Callsen, University of Illinois at Urbana-Champaign, UILU-ENG-92-1846.

JS.4.2 The Java Platform

6. *The Java Programming Language, Second Edition,* Ken Arnold and James Gosling, Addison Wesley, 1998.
7. *The Java Language Specification,* James Gosling, Bill Joy, and Guy Steele, Addison Wesley, 1996.
8. *The Java Virtual Machine Specification, Second Edition,* Tim Lindholm and Frank Yellin, Addison Wesley, 1999.
9. *The Java Class Libraries, Second Edition,* Patrick Chan, Rosanna Lee, and Doug Kramer, Addison Wesley, 1998.

JS.4.3 Distributed Computing

10. *Distributed Systems,* Sape Mullender, Addison Wesley, 1993.
11. *Distributed Systems: Concepts and Design,* George Coulouris, Jean Dollimore, and Tim Kindberg, Addison Wesley, 1998.
12. *Distributed Algorithms,* Nancy A. Lynch, Morgan Kaufmann Publishers, 1997.

JavaSpaces Technology (JS)

THE JINI DEVICE ARCHITECTURE SPECIFICATION describes several ways in which a device (or any other service) can participate in a Jini system without the device (or service) being a general Jini service. The possibilities listed are not exhaustive—there could be other interesting models as well. The main point to pay attention to here is that any service can participate in the Jini architecture, even with no modification of the service provider itself. This "device architecture" applies equally well to legacy systems and other software services.

DA

The Jini Device Architecture Specification

DA.1 Introduction

THE Jini technology infrastructure is built around the model of clients looking for services. The notion of a service encompasses access to information, computation, software that performs particular tasks, and in general any component that helps a user accomplish some goal. Services can themselves be clients of other services, and can be grouped together to provide higher-level functionality.

The Jini architecture requires a service to be defined in terms of a data type for the Java programming language that can then be implemented in different ways by different instances of the service. A service can be a member of many different types, allowing a single service instance to provide a variety of functionality to clients. This is a standard practice in object-oriented software. However, the distributed nature of the Jini system allows data types for the Java programming language to be implemented in a combination of software and hardware in a way that is unique.

The core of the idea that enables this implementation flexibility is quite simple. Services are defined via an interface, and the implementation of a proxy supporting the interface that will be seen by the service client will be uploaded into the lookup service by the service provider. This implementation is then downloaded into the client as part of that client finding the service. This service-specific implementation needs to be code written in the Java programming language (to ensure portability). However, since this code comes from the actual instance of the service being used, it can know in great detail the specifics of the particular service implementation for which it is the proxy. Not only can the code that is downloaded know about the software used to implement the service, the code can know

specifics about the hardware on which the service resides. In the limit case of this, the hardware could be all that there is to the service, and the downloaded software could act as a network-level device driver, taking method calls in the Java programming language from the client and generating specific, hard-coded requests to the hardware on the other end of the network wire.

This approach to services requires that there be a piece of code written in the Java programming language that can be downloaded by the client of the service and some hardware that ultimately runs the service. Between these two points, however, there are a number of options concerning the software structure, hardware structure, and location of components that can be chosen by the service provider. These options allow trade-offs to be made in the functionality provided and the cost of the underlying hardware.

In what follows we begin by discussing in more detail the requirements placed on a service to be part of the Jini system. We then discuss some examples of combinations of software and hardware that can be used to implement Jini-capable services once the specialized implementations in hardware begin to play a role.

DA.1.1 Requirements from the Jini Lookup Service

The actual offering of a service places very few requirements on the entity that makes the offer; indeed, it is possible to implement a device using Jini software services that offers a service in such a way that the code written in the Java programming language that is downloaded by the client transmits bit patterns to the hardware that are directly interpreted. In such cases the amount of intelligence needed for a Jini device is minimal. The code written in the Java programming language could talk directly to the device controller in much the same way that the device would be talked to if it were on the local computer's bus (with, of course, some modifications for dealing with the network-centric aspects of the communication).

Unfortunately, providing a service is only part of what is needed to be a Jini service. To be part of a Jini system grouping, a service must also be able to participate in the Jini Discovery protocol and register itself into the local Jini Lookup service. This is how a service makes itself known to the djinn, and how the service is accessed by other members of the djinn.

These two requirements are intimately connected. The major goal of the Jini Discovery protocol is to allow a device or service to obtain a Java Remote Method Invocation (RMI) reference to the local Jini Lookup service. Once this reference has been obtained, the service needs to register itself in that Jini Lookup service, allowing other participants in the djinn to find and use the service.

The interface to the Jini Lookup service is a full RMI interface, and the implementation of that service uses all of the mechanisms of RMI, including the distributed garbage collection and the dynamic downloading of code. As such, there is an implicit assumption that the service that holds a reference to the Jini Lookup service lives inside a full Java™ virtual machine (JVM) that is at least capable of running the full RMI system.

This assumption is most evident if we consider the possibility of alternate implementations of the Jini Lookup service, which might support remote interfaces beyond that specified by the Jini Lookup service itself (currently the interface `net.jini.core.lookup.ServiceRegistrar`). Such an implementation would have a different RMI proxy than the current implementation, which would be downloaded if the device had a full JVM and RMI runtime. Devices without a full JVM and RMI runtime would need a different way of dealing with such implementations of the service.

In addition to the need to download the stub code for the Jini Lookup service, registering with the service requires the creation of an object of type `net.jini.core.lookup.ServiceItem`, which is itself made up of a set of objects in the Java programming language. Maintenance of these entries in the Jini Lookup service can require the creation of other objects in the Java programming language of the type `net.jini.core.entry.Entry`. All of these objects are most easily constructed by using a running JVM.

Finally, registrations with the Jini Lookup service are leased, with the lease that is returned requiring renewal for the service to continue to be shown in the lookup service. The specification of the lookup service does not include a specification of the lease object that is returned by a registration. All that is specified is an interface written in the Java programming language that must be supported by the (local) object that is returned as the lease. Thus the design of the Jini Lookup service requires that the code that implements the class that in turn implements the `net.jini.core.lease.Lease` interface be downloaded into the service that registers so that the lease can be renewed.

DA.2 Basic Device Architecture Examples

Now we will look at three different approaches for implementing a Jini service in hardware. Each of the approaches will look the same to a client of the service. Each approach takes a different route to interacting with the Jini Lookup service and in providing an interface written in the Java programming language to clients of that service. In each case, a different trade-off was made between the complexity of the device, the flexibility of the device, and the directness of the communication between the client wanting to use the service and the device that implements the service.

All but the first of the examples make use of *interposition*, that is, the ability of a service to add a proxy between itself and the client of the service. The service can use this proxy as an agent to the Jini technology infrastructure, off-loading from the service some of the work needed to join the Jini system federation.

The examples given in this chapter are not the only options available to the service designer who wishes to produce a service that includes a hardware component. Rather, the examples are meant to show some samples of the range of implementation possibilities that are open to such designers. In effect, this document is meant to show that, within the overall Jini architecture, there is no single Jini device architecture. Instead, the device space is freed up, allowing different services to have hardware implementations with different price, performance, functionality, and flexibility design points.

DA.2.1 Devices with Resident Java Virtual Machines

An obvious design for a device that can become part of a Jini system federation is one that includes the computing power, memory, and nonvolatile store necessary to have a full JVM and those parts of the Java application environment necessary to support the Jini infrastructure (in particular, those parts needed for code loading, RMI, and any required security). This would make the device into a specialized computing entity, with part of the device dedicated to the parts of the Java API required by the Jini architecture. On this approach, the hardware implementation is abstracted behind a device-local software abstraction, which in turn is

abstracted behind the proxy code used by the client to contact the service. This sort of architecture is shown in Figure DA.2.1.

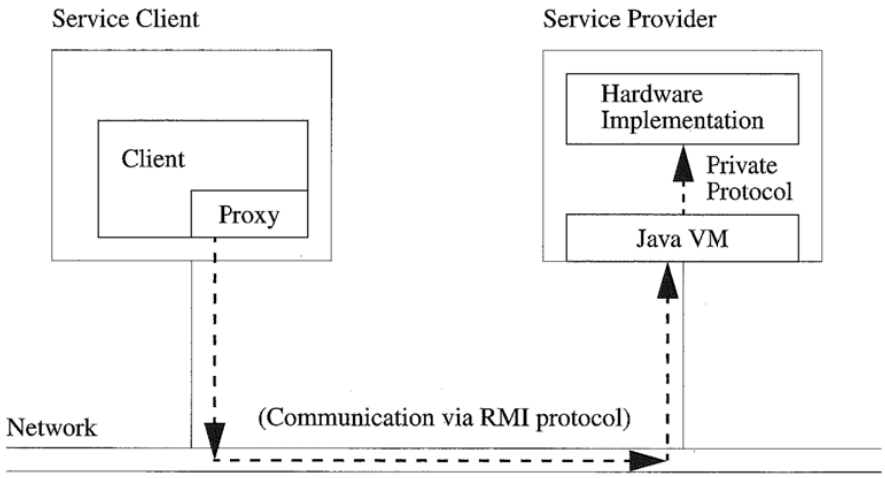


FIGURE DA.2.1: ***A Full Jini-Capable Device***

Such a device would be able to make full use of Jini and Java technology, uploading code that is used to communicate with the device and downloading code that might be needed for the service provided by the device. Such a device can make use of the native RMI protocol for communication over the network, and has a loose tie between the communication protocol and the particular software protocol governing the running of the device itself. On this approach, the device becomes a specialized network appliance offering a particular service (or set of services) via an embedded Java platform.

In effect, this approach uses a hardware implementation for the local implementation of an RMI server, isolating the hardware behind two levels of indirection. The first is that provided by the local proxy code that is uploaded into the Jini Lookup service and then downloaded into the client of the service. Additionally, the local JVM and code written in the Java programming language resident on the service device allow mediation between the client proxy and the hardware itself.

A device that took this approach could easily have multiple services implemented on the device in a way that was mediated by the JVM on the device. Further, such a device could be evolved with no impact on the client or the network

protocol used between the client and the service, since any change in the hardware would be seen only by the JVM and any server-side code that talked directly to the hardware.

While simple and flexible, this approach does add some cost to the device. In particular, the device would need to have a microprocessor capable of running the JVM, some memory in which to create and store classes, and some nonvolatile store (either disk or NVRAM) from which to load the JVM and Java™ Development Kit (JDK) software classes. All of these are in addition to the hardware needed to implement the Jini service that the device provides. This extra hardware will increase the cost of producing the device.

Meeting these requirements does not call for a hosted version of the JVM or a full version of the JDK running on the device. The JVM could run on any form of microkernel or directly on the hardware of the device. Further, there are large parts of the JDK that would not be required for the minimal device—such things as the graphics and user interface classes, which form a significant chunk of the current release, would not be needed. Other parts of that release could also be dropped, allowing a stripped-down JDK to suffice for Jini devices. It would be worthwhile to determine the exact definition of such a subset of the JDK and size that component; it would be something close to the definition of embedded Java technology with the additional classes needed to support RMI.

What is important for this kind of approach is for the device to be able to download any code written in the Java programming language (although whether that code is run could depend on the local security manager), utilize the RMI communication system, and handle the requirements of a general virtual machine. By presenting a standard JVM, the device gets full membership in a Jini system federation and complete flexibility in the ways in which the machine communicates between the proxy it provides other members of the federation and the device itself.

DA.2.2 Devices Using Specialized Virtual Machines

We can lower the barrier to entry for a device manufacturer if that manufacturer is willing to give up some of the flexibility given by the Jini distribution architecture. This can be done by allowing the device to become part of a Jini system federation with a specialized virtual machine that is tuned to allow only those operations needed by the Jini Discovery protocol and Jini Lookup service.

To do this, the device manufacturer would need to implement the interfaces to the Jini Discovery and Jini Lookup service in the device itself, include specialized knowledge of the kind of leases that are handed out by the Jini Lookup service and be able to renew those leases directly, and have sufficient functionality to

download and use the stubs for these services. This is a particular set of functionalities that is considerably smaller than that required by the whole of the JVM, and should be possible to implement in much less code. For example, such a JVM would not need to contain a security manager, a code verifier, or a number of the other components that are required for a full JVM.

Such a device would contain a JVM specialized for the Jini environment, allowing the Jini Discovery and Jini Lookup services to be accessed and leases of a particular sort to be renewed. This would limit the flexibility of such a device, as the device would not be able to have software changes made over time to the protocol used by the proxy for the device. The specialized knowledge of the kind of lease that is handed out by the lookup service would also tie such a device to a particular implementation of the lookup service. However, this penalty in serviceability might not outweigh the simplicity of the overall device.

DA.2.3 Clustering Devices with a Shared Virtual Machine (Physical Option)

A third approach uses a full JVM, but amortizes the cost of the JVM (both software and hardware) over a number of different devices. In this approach, a group of devices each uses a physically co-located JVM as an intermediate layer between the device and the Jini system grouping. The device loads code written in the Java programming language into this local virtual machine, allowing that local machine to interact with the device, and then delegates to the local JVM the requirements of interacting with the Jini Lookup service, Jini Discovery, and Jini Leasing.

This approach is very much like the first one discussed in this section, except that the JVM used by the devices is shared. It is still a full JVM, allowing the downloading of code and complete Java platform functionality. However, the most likely implementation of such a device would allow multiple (and perhaps different) kinds of physical devices to be plugged into the overall device to get the sharing of the Java application environment.

Such a device might best be thought of as a "Jini device bay." This bay could provide power, a network connection, and a processor running a JVM and appropriate parts of the JDK. Physical devices that are used to provide a particular kind of Jini service could be plugged into the device bay and announce themselves to the bay in whatever way the two decided was appropriate. This could be using a proprietary protocol (allowing a device manufacturer to produce both the basic device or devices and the device bay) or some other industry standard, local-device identification scheme.

As part of the local announcement, a new device would tell the device bay where to find the code written in the Java programming language that is needed by a client of the service, and (possibly) where to find code that would allow the device bay to interact with the device. This allows devices to carry their own "drivers," both for the local machine and at the network level.

Upon detection of the new local device, the Jini device bay would register the services provided by the new device (previously known by the device bay) with the Jini Lookup service. It would be the role of the device bay to renew leases on the Jini Lookup service entries, and to detect removal of any of the devices for which it was acting as proxy. The device bay would provide the Jini Lookup service with the code handed to it by the device so that service clients could download that code.

The client of the device service would believe that it is talking to the device registered in the Jini Lookup service, but would actually be talking to the device bay. The device bay would act as a dispatcher to the particular device for which it was acting as a proxy, along with any translation of protocol between the network protocol used by the service proxy and the protocol used between the device bay and the actual device. Graphically, the architecture of such an approach is shown in Figure DA.2.2.

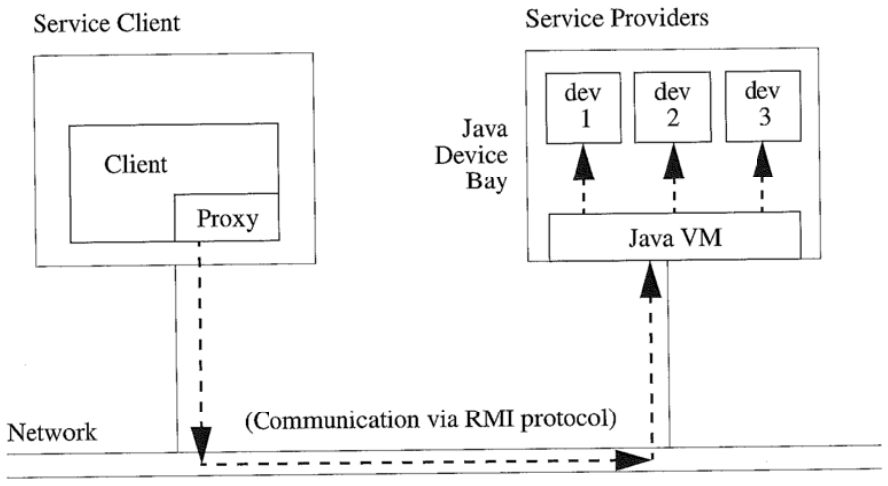


FIGURE DA.2.2: ***Clustering Multiple Devices With a Single Proxy in One Device***

The savings for the device manufacturer in this case comes from the ability of multiple physical devices to share a device bay, which contains the intelligence, memory, and perhaps other components (such as the power supply). By sharing these resources among multiple devices, the extra cost and engineering needed to interact with the Jini system federation can be amortized over a large number of devices.

The cost of this approach to the device manufacturers is that the protocol between the device acting as the Jini device bay and the devices that are placed in that bay must be defined in advance and cannot change over time. Because there is no way of introducing dynamic behavior in the particular devices, the pairing of device and Jini device bay must be controlled and known beforehand.

It should be noted that the Jini device bay itself is a Jini device, which can be thought of as providing services to those devices housed within it. As such, it could be a revenue item in its own right. Variations in the implementation could be provided to support various internal announcement protocols (device bay, jetsend, etc.) or hardware buses (including network-like buses such as firewire).

DA.2.4 Clustering Devices with a Shared Virtual Machine (Network Option)

A variation on the device bay approach uses the network rather than a physical enclosure and backplane. On this alternative, a proxy for the JVM used by the various service devices would exist on the network. Service devices could be added to the network, discover the existence of such a proxy device, and register with that proxy. Such a registration could include the code written in the Java programming language needed by a client of the device (either directly or as a URL to use to obtain the code) and code needed by the proxy to communicate with the service device.

When a service device registers with such a network proxy, the proxy device would register with the Jini Lookup service on behalf of the service device, thus allowing the service device to become a part of the Jini system federation. Requests to the new service would go first to the proxy for that device, which could then forward the requests (after appropriate protocol translation) to the particular service device. In addition, the proxy could handle the Jini-specific tasks such as renewing leases for the service. This alternative is shown in Figure DA.2.3.

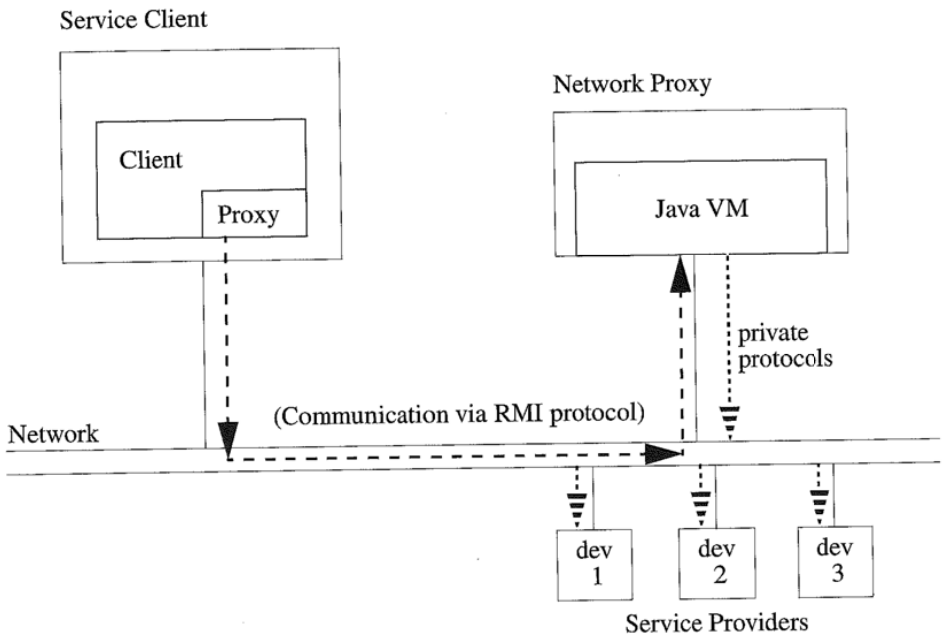


FIGURE DA.2.3: ***Clustering Devices With a Jini-capable Proxy on the Network***

This alternative requires somewhat more hardware for the individual device, as it requires each service device using such a proxy to be able to be placed on the network and have its own power supply and network connection. However, the devices would not need individual CPUs, memory, or persistent store; all of that would be provided by the networked Jini device proxy.

Devices using this option would need to have a protocol parallel to the Jini Discovery protocol between the individual service devices and the network proxy for those devices. This could be a specialized code on the network, known in advance, that the devices can use to identify themselves to the network proxy. This will have to be particular to the device and the proxy for that device. However, once this protocol has been decided upon, no other intelligence needs to be built into the device. All of the intelligence can be built into the network proxy, perhaps uploaded into the proxy by the service device (which could easily carry code written in the Java programming language, even though it cannot execute that code).

The protocol the network proxy uses to talk to the devices for which it is a proxy also needs to be statically defined in advance and cannot be changed. However, it can be any protocol the particular device needs.

In this approach, the individual devices will be more complex than they would be in the Jini device bay approach. However, the number of devices that can be served by a network available proxy is not limited by the physical constraints of the proxy device. Nor is there any requirement that the devices and the proxy device be co-located, which is a requirement on the physical clustering scheme.

This is also the approach that can be taken to build "gateways" between the Jini devices and other network-managed devices. Such devices, which already speak a particular protocol, can be spliced into the Jini system federation by providing a network proxy that speaks the Jini protocol on behalf of such devices, and the existing specialized protocol to such devices. This is the approach that can be used to add consumer electronic devices, factory controls, or home environment controls into the Jini system grouping.

DA.2.5 Jini Software Services over the Internet Inter-Operability Protocol

A final method for connecting devices or services that are not purely based on Java software into a Jini system centers on using the Object Management Group (OMG)'s Internet Inter-Operability Protocol (IIOP). This protocol defines a standard for data transmission that will be supported by a subset of RMI.

This approach relies on the ability of a device to read an IIOP stream directly, either because the device includes an implementation of a Common Object Request Broker Architecture (CORBA) Object Request Broker (ORB) or because the device knows what IIOP streams to expect and can interpret streams of these known forms directly.

This approach requires the Jini Lookup service to supply implementations of its interfaces over both the native RMI protocol and the IIOP protocol. This is supported by RMI over IIOP as long as the interfaces conform to any subsetting requirements established by the OMG. At the present time it appears that the Jini Lookup service interfaces are in conformance with the RMI over IIOP subset.

Devices that contain a CORBA ORB could directly interact with the Jini Lookup service using the IIOP protocol. The fact that the Jini Lookup service generated this protocol via RMI would be transparent to the service itself, and the fact that the service was using a method other than RMI to reply to the Jini Lookup service (to renew leases, for example) would be transparent to the Jini Lookup service. Current differences between the RMI programming model and the CORBA programming model would need to be dealt with by the device itself; for example,

the device would not be able to download the implementation of the stub for the Jini Lookup service, and would need an implementation of the Jini Lease class used by the Jini Lookup service.

Devices that do not include a CORBA ORB could directly interpret the IIOP stream and attempt to interact with the Jini Lookup service. This approach requires very little software support on the side of the device (since the bitstream from the wire is being directly interpreted). However, it is an approach that will work only with known versions of the Jini Lookup service that exports known implementations of a Jini Lease. Any alteration of either the Jini Lease implementation or the protocol used by the Jini Lookup service, even those that would be invisible to other clients of the service, would make it impossible for the device directly interpreting the IIOP protocol to interact with the new version of the service. Hence this alternative, while lowest in cost with respect to the hardware and software needed by the device, is also the least reliable in the face of implementations that can change over time or that are open to alternate implementations.

PART 3

Supplemental Material

The Jini Technology Glossary

activation

The process of transforming a passive object into an active object. Activation requires that an object be associated with a Java™ virtual machine (JVM), which may entail loading the class for that object into a JVM and the object restoring its persistent state (if any). (*Java Remote Method Invocation Specification,* Section 7.1.1)

activation descriptor

A class instance that holds an activatable object's group identifier (specifies the JVM in which it is activated), the object's class name, a location from where to load the object's class code, and object-specific initialization data in marshalled form. (*Java Remote Method Invocation Specification,* Section 7.2)

activation group

The entity that receives a request to activate an object in the JVM and returns the activated object back to the activator. (*Java Remote Method Invocation Specification,* Section 7.2) A separate JVM is spawned for each activation group. (Section 7.4.7)

activator

The entity that supervises activation by being both (1) a database of information that maps activation identifiers to the information necessary to activate an object and (2) a manager of JVMs, that starts up a JVM (when necessary) and forwards requests for object activation (along with the necessary information) to the correct activation group inside a remote JVM. There is usually only one activator per host, started by `rmid`. (*Java Remote Method Invocation Specification,* Section 7.2)

active object

A remote object that is instantiated and exported in a JVM on some system.(*Java Remote Method Invocation Specification,* Section 7.1.1)

ancestor transaction

A transaction that is the parent of a specific nested transaction (a transaction in which all its operations are contained, or executed, from within another transaction), or the parent of such a parent, recursively (a grandparent, a great-grandparent, and so on). (*Jini Transaction Specification,* Section TX.3.5)

attribute set

A strongly-typed set of fields in a service item (represented by a `net.jini.core.entry.Entry`) that describes the service or provide secondary interfaces to the service. A single attribute is a public field of an `Entry`. (*Jini Lookup Service Specification,* Section LU.1.2)

channel

The abstraction for a conduit between two address spaces in the RMI transport layer. As such, it is responsible for managing connections between the local address space and the remote address space for which it is a channel. (*Java Remote Method Invocation Specification,* Section 3.5)

connection

The stream-oriented (*Java Remote Method Invocation Specification,* Section 3.4) abstraction for transferring data (performing input/output) in the RMI transport layer. (Section 3.5)

discovering entity

One or more cooperating objects in the Java programming language on the same host that are about to start, or are in the process of, obtaining references to one or more Jini Lookup services. (*Jini Discovery and Join Specification,* Section DJ.1.1)

discovery request service

A service that runs on a host in the djinn and accepts requests for a remote reference to an instance of the Jini Lookup service. There are really two discovery request services; one accepts multicast requests, and the other accepts unicast requests. Both instances of the discovery request service are present on every system in a djinn that hosts an instance of the Jini Lookup service.

discovery response service
A remote object that runs on a discovering entity and accepts references to instances of the Jini Lookup service. An instance of the discovery response service is hosted on every system that wishes to establish communications with a djinn.

distributed event adapter
An event adapter in which the event generator and the event listener instances may exist in different virtual machines, possibly on different hosts. The distributed event adapter is at least a remote event listener, but may also be a remote event generator (see *local event, remote event*). (*Jini Distributed Event Specification,* Section EV.3)

djinn ***(pronounced "gin")***
The group of devices, resources, and users joined by the Jini software infrastructure. (*Jini Lookup Service Specification,* Section LU.1.1) This group, controlled by the Jini system, agrees on basic notions of trust, administration, identification, and policy.

dynamic class loading
The capability of the Java application environment to download files (classes for the Java platform, audio, and images) from an HTTP server at runtime if they are not already available to the client JVM. Dynamic class loading may be used by the RMI runtime to download: stub classes; skeleton classes; classes that are passed as subtypes of declared method parameters; and classes that are passed as subtypes of declared method return types. (See *dynamic stub loading*)

dynamic stub loading
A subset of dynamic class loading, used to support client-side stubs that implement the same set of remote interfaces as a remote object itself. (*Java Remote Method Invocation Specification,* Section 3.1)

endpoint
The abstraction used to denote an address space or JVM in the RMI transport layer. In the implementation an endpoint can be mapped to its transport. That is, given an endpoint, a specific transport instance can be obtained. (*Java Remote Method Invocation Specification,* Section 3.5)

entry

An entry is a typed group of object references, expressed as a class for the Java platform that implements the `net.jini.core.entry.Entry` interface. Entry fields must all be references to `Serializable` objects. (*Jini Entry Specification,* Section EN.1)

event

Something that happens in an object, corresponding to some change in the abstract state of the object. Events are abstract occurrences that are not directly observed outside of an object, and may not correspond to a change in the actual state of the object that advertises the ability to register interest in the event. (*Jini Distributed Event Specification,* Section EV.2.1)

event generator

An object that has some kinds of abstract state changes that might be of interest to other objects and allows other objects to register interest in those events. This is the object that will generate notifications when events of this kind occur, sending those notifications to the event listeners that were indicated as targets in the calls that registered interest in that kind of event. (*Jini Distributed Event Specification,* Section EV.2.1)

event listener

An object that has an interest in being notified when a particular event type occurs. The event listener (1) implements the appropriate interface, and (2) registers with an event generator. (See *remote event listener*)

export, -ed, -ing

The process of making a remote object available to accept incoming calls on a specific port. An object can be exported (1) if the object is a subclass of `java.rmi.server.UnicastRemoteObject`, through the constructor; (2) if the object is a subclass of `java.rmi.activation.Activatable`, through the constructor; (3) by passing the object to the static `exportObject` method of `UnicastRemoteObject` (*Java Remote Method Invocation Specification,* Section 5.3.1); or (4) by passing the object to the static `exportObject` method of `Activatable`. (Section 7.3)

faulting remote reference

A faulting remote reference to a remote object, sometimes referred to as a fault block, "faults in" the active object's reference upon the first method invocation to the object executed via the faulting reference. Each faulting reference, contained in the remote object's stub, holds both a persistent

object handle (a `java.rmi.activation.ActivationID`) and a transient remote reference to the target remote object. (*Java Remote Method Invocation Specification,* Section 7.1.2)

host

A hardware device that may be connected to one or more networks. An individual host may house one or more JVMs. (*Jini Discovery and Join Specification,* Section DJ.1.2)

idempotent

A method that is idempotent can be called multiple times and produce only the result as though it were called only a single time.

inferior transaction

The inverse of the transactional ancestor relationship: Transaction T_i is an inferior of T_a if and only if T_a is an ancestor of T_i. (*Jini Transaction Specification,* Section TX.3.5)

joining entity

One or more cooperating objects in the Java programming language on the same host that have just received a reference to the Jini Lookup service and are in the process of obtaining services from, and possibly exporting services to, a djinn. (*Jini Discovery and Join Specification,* Section DJ.1.1)

join protocol

The protocol that allows entities to start communicating usefully with services in a djinn, through the Jini Lookup service. (*Jini Discovery and Join Specification,* Section DJ.1.3)

JVM

A common abbreviation for "Java Virtual Machine."

lazy activation

The activation mechanism that the RMI system uses, which defers activating an object until a client's first use (that is, the first method invocation). Lazy activation of remote objects is implemented using a *faulting remote reference*. (*Java Remote Method Invocation Specification,* Section 7.1.1)

lease

A grant to use a resource, offered by one object in a distributed system, to another object in that system for a certain period of time. The duration of

the lease is negotiated by the two objects when access to the resource is first requested and given. (*Jini Distributed Leasing Specification,* Section LE.1) A lease ensures that the lease holder will have access to some resource for a period of time. During the period of a lease, a lease can be cancelled by the entity holding the lease. A lease holder can request that a lease be renewed, or a lease can expire. (*Jini Distributed Leasing Specification,* Section LE.2.1) In the current implementation of RMI, a lease term is not negotiated, as described by the *Jini Distributed Leasing Specification;* the lease term is mandated by the implementation server. Another difference is that in RMI there is no notion of explicit lease cancellation; lease cancellation is implicit when a remote reference becomes unreferenced by a specific client. (*Java Remote Method Invocation Specification,* Section 9.1)

lease grantor

The object granting access to a resource for some period of time. (*Jini Distributed Leasing Specification,* Section LE.2)

lease holder

The object asking for the leased resource. (*Jini Distributed Leasing Specification,* Section LE.2)

live reference

The concrete representation of a remote object reference (in the RMI transport layer), which consists of an endpoint and an object identifier. Given a live reference for a remote object, a transport can use the endpoint to set up a connection to the address space in which the remote object resides. On the server side, the transport uses the object identifier to look up the target of the remote call. (*Java Remote Method Invocation Specification,* Section 3.5)

local event

An event object that is fired from an event generator to an event listener, where both the generator and the listener instances exist in the same virtual machine. (See *event, remote event*) (*Jini Distributed Event Specification,* Section EV.1.1)

lookup discovery protocol

The protocol that governs the acquisition of a reference to one (or more) instances of the Jini Lookup service. (*Jini Discovery and Join Specification,* Section DJ.1.3)

lookup service

The Jini Lookup service provides a central registry of service items, representing services, available within the djinn. This Jini Lookup service is a primary means for programs to find services within the djinn, and is the foundation for providing user interfaces through which users and administrators can discover and interact with services in the djinn. (*Jini Lookup Service Specification,* Section LU.1)

marshal streams

Input/output streams, used by the RMI remote reference layer, that employ *object serialization* to enable objects in the Java programming language to be transmitted between address spaces. (*Java Remote Method Invocation Specification,* Section 3.3)

marshalled object

A container for an object that allows that object to be passed as a parameter in an RMI call, but postpones deserializing the object at the receiver until the application explicitly requests the object (via a call to the container object). The *serializable* object contained in the `MarshalledObject` is serialized and deserialized (when requested) with the same semantics as parameters passed in RMI calls (*Java Remote Method Invocation Specification,* Section 7.4.8), which means that any remote object in the `MarshalledObject` is represented by a serialized instance of its stub. The object contained by the `MarshalledObject` may be a remote object, a non-remote object, or an entire graph of remote and non-remote objects.

notification filter

A distributed event adapter that can be used by either the generator of a notification or the recipient to intercept notification calls, do processing on those calls, and act in accord with that processing (perhaps forwarding the notification, or even generating new notifications). (*Jini Distributed Event Specification,* Section EV.3.2) This filter may be used as an event multiplexer or demultiplexer.

notification mailbox

A distributed event adapter that can be used to store the notifications sent to an object until such time as the object for which the notifications were intended desires delivery. Such delivery can be in a single batch, with the mailbox storing any notifications received after the request for delivery until the next request is given. Alternatively, a notification mailbox can be viewed as a faucet, with notifications turned on (delivering any that have

arrived since the notifications were last turned off) and then delivering any subsequent notifications to an object immediately, until told to hold the notifications. (*Jini Distributed Event Specification,* Section EV.3.3)

object serialization

The system that allows a bytestream to be produced from a graph of objects, sent out of the Java application environment (either saved to disk or sent over the network) and then used to re-create an equivalent set of objects with the same state. (*Java Object Serialization Specification,* Section A.1) In RMI, objects transmitted using the object serialization system are passed by copy to the remote address space, unless they are remote objects, in which case they are passed by reference. (*Java Remote Method Invocation Specification,* Section 3.3)

passive object

A remote object that is not yet instantiated (or exported) in a JVM, but that can be brought into an active state (see *active object*). (*Java Remote Method Invocation Specification,* Section 7.1.1)

pure transaction

A transaction in which all access to shared mutable state is performed under transactional control. (*Jini Transaction Specification,* Section TX.3.5)

reference list

A reference list for a remote object is a list of client JVMs that hold references to that remote object. A client JVM is removed from the object's reference list when that client no longer references that object. (*Java Remote Method Invocation Specification,* Section 9.1)

registry

A remote object that maps names to remote objects. The `java.rmi.Naming` class provides methods for lookup, binding, rebinding, unbinding, and listing the contents of a registry. A registry can be used in a virtual machine shared with other server classes or in a standalone JVM. The methods of `java.rmi.registry.LocateRegistry` may be used to get a registry operating on a particular host or host and port. (*Java Remote Method Invocation Specification,* Section 6)

remote event

An object that is passed from an event generator to a remote event listener to indicate that an event of a particular kind has occurred. The remote event

generator and the remote event listener instances may exist in different virtual machines, possibly on different hosts. (*Jini Distributed Event Specification,* Section EV.2.1)

remote event generator
An object that is the source of remote events.

remote event listener
An object implementing the `net.jini.core.event.RemoteEventListener` interface, which is interested in the occurrence of remote events in some other object. The major function of a remote event listener is to receive notifications of the occurrence of a remote event in some other object (or set of objects). (*Jini Distributed Event Specification,* Section EV.2.1)

remote interface
An interface written in the Java programming language that extends `java.rmi.Remote`, either directly or indirectly, which declares the methods of a remote object. (*Java Remote Method Invocation Specification,* Section 2.1)

remote method invocation (RMI)
The action of invoking a method of a remote interface on a remote object. (*Java Remote Method Invocation Specification,* Section 2.1)

remote object
An object whose methods can be invoked from another JVM, potentially on a different host. An object of this type is described by one or more *remote interfaces.* (*Java Remote Method Invocation Specification,* Section 2.1)

remote reference layer (RRL)
The layer of the RMI system that supports remote reference behavior (such as invocation to a single object or to a replicated object) and carries out the semantics of method invocation. This layer sits between the RMI stub/skeleton layer and the RMI transport layer. Also handled by the remote reference layer are the reference semantics for the server. (*Java Remote Method Invocation Specification,* Section 3.2)

rmic
The stub and skeleton compiler used to generate the appropriate stubs and skeletons for a specific remote object implementation. The compiler is invoked with the package-qualified class name of the remote object class.

The class must previously have been compiled successfully. (*Java Remote Method Invocation Specification,* Section 5.11)

rmid

The activation system daemon which provides an implementation of the activation system interfaces. To use activation, you must first run `rmid`. This is the JVM with which activation descriptions get registered. (*Java Remote Method Invocation Specification,* Section 7.2)

rmiregistry

The RMI system command that provides an implementation of the `java.rmi.registry.Registry` interface. The rmiregistry, run on a remote host, can be accessed by calling methods of the `java.rmi.Naming` class.

semantic transaction

A *transaction* with specific, associated semantics, as opposed to the protocol specified by the `TransactionManager` interface, which does not specify transaction semantics. A semantic transaction is contractual in nature and implies a particular usage pattern, so if a program operates within the constraints of the contract, assumptions can be safely made about the transaction's behavior or state. (*Jini Transaction Specification,* Section TX.1.1)

serializable

Any data type that may be read from `java.io.ObjectInputStreams` and written to `java.io.ObjectOutputStreams`. This includes primitive data types in the Java programming language, remote objects in the Java programming language, and non-remote objects in the Java programming language that implement the `java.io.Serializable` interface. (*Java Remote Method Invocation Specification,* Section 2.6)

service

Something that can be used by a person, a program, or another service. It can be computational, storage, a communication channel to another user, or another service. Examples of services include devices such as printers, displays, disks, software (such as applications or utilities), information (such as databases and files), and users of the system. Services will appear programmatically as objects in the Java programming language, perhaps made up of other objects in the Java programming language. A service will have an interface, which defines the operations that can be requested of that service. The type of the service determines the interfaces that make up that service. (*Jini Architecture Specification,* Section AR.2.1.1)

service items

Each service item represents an instance of a service available within the djinn. The item contains the stub (if the service is implemented as a remote object) or serialized object (if the service makes use of a local proxy) that programs use to access the service, and an extensible collection of attribute sets that describe the service or provide secondary interfaces to the service. A new service item is created in the Jini Lookup service when a new service is added to the djinn. (*Jini Lookup Service Specification,* Section LU.1.1)

service registrar

A synonym for Jini Lookup service. (See *lookup service*) (*Jini Lookup Service Specification,* Section LU.2.5)

skeleton

The server-side entity that reads parameters from incoming method requests and dispatches calls to the actual remote object implementation. Note that in the Java Development Kit 1.2, skeleton functionality is now handled by the remote object stub, but skeletons may still be used for compatibility with earlier releases of the JDK. (*Java Remote Method Invocation Specification,* Section 3.3)

store-and-forward agent

A distributed event adapter that enables the object generating a notification to hand the actual notification of those who have registered interest off to a separate object. This agent can implement various policies for reliability. (*Jini Distributed Event Specification,* Section EV.3.1)

stub

The proxy for a remote object, which implements all the interfaces that are supported by the remote object implementation and forwards method invocations to the actual remote object instance. (*Java Remote Method Invocation Specification,* Section 3.3)

stub/skeleton layer

The layer of the RMI system that aids in carrying out method invocation. The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. (*Java Remote Method Invocation Specification,* Section 3.3) This layer does not deal with specifics of any transport, but transmits data to the remote reference layer via the abstraction of *marshal streams*. This layer contains client-side stubs (proxies) and server-side skeletons. (Section 3.2)

template

An entry object that has some or all of its fields set to specified *values*. Templates may be used to find matching entries. A template will match an entry if and only if the template's non-`null` public fields match the entry's non-`null` public fields exactly. Remaining fields (those set to `null`) are not used in the matching process but are left as *wildcards*. (*Jini Entry Specification,* Section EN.1.5)

transaction

In general, a transaction is a tool that allows a set of operations to be grouped in such a way as to make them all appear to either all succeed or all fail; further, the operations in the set appear from outside the transaction to occur simultaneously. In the Jini architecture model, the concrete representation of a transaction is encapsulated in an object. (*Jini Transaction Specification,* Section TX.1.1)

transaction client

An object that does either or both of the following: (1) requests that a transaction manager create a transaction, (2) invokes the `commit` or `abort` method to complete a transaction. A single transaction may have more than one client, since the object that completes a transaction may be different from the object that requested its creation. An object that is a transaction client may also be a transaction manager or participant. (*Jini Transaction Specification,* Section TX.1.1)

transaction manager

An object that (1) services requests from transaction clients to create transactions and (2) tracks and manages the completion state of those transactions by implementing the `TransactionManager` interface. An object that is a transaction manager may also be a transaction client or participant. (*Jini Transaction Specification,* Section TX.1.1)

transaction participant

An object that executes operations of a transaction and is able to interact with the manager to complete transactions properly. An object providing this service may implement the `TransactionParticipant` interface. An object that is a transaction participant may also be a transaction manager or client. (*Jini Transaction Specification,* Section TX.1.1)

transport

The abstraction that manages channels in the RMI transport layer. Each channel is a virtual connection between two address spaces. Within a transport, only one channel exists per pair of address spaces (the local address space and a remote address space). Given an endpoint to a remote address space, a transport sets up a channel to that address space. The transport abstraction is also responsible for accepting calls on incoming connections to the address space, setting up a connection object for the call, and dispatching to higher layers in the system. (*Java Remote Method Invocation Specification,* Section 3.5)

transport layer

The layer of the RMI system that is responsible for connection set up, connection management, and remote object tracking. (*Java Remote Method Invocation Specification,* Section 3.2) The transport layer sits below the *remote reference layer.*

weak reference

When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the JVM's garbage collector to discard the object if no other strong references to the object exist. The distributed garbage collection algorithm interacts with the local JVM's garbage collector in the usual ways by holding normal or weak references to objects; thus, a weak reference allows the RMI runtime to reference a remote object, but not prevent the object from being garbage collected. (*Java Remote Method Invocation Specification,* Section 3.7)

NOTE ON DISTRIBUTED COMPUTING describes the environment for which the Jini architecture is designed—one of failure characteristics unknown in local computing. The Jini architecture takes these differences into account in its original design principles, which is one reason why the overall Jini architecture works.

This note was originally published as a Sun Microsystems Laboratories technical report (SMLI TR-94-29). The note has been reformatted for this book. Two observations have been added, marked as [A] and [B] in the text, and presented at the end of the note.

APPENDIX A

A Note on Distributed Computing

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall

A.1 Introduction

MUCH of the current work in distributed, object-oriented systems is based on the assumption that objects form a single ontological class. This class consists of all entities that can be fully described by the specification of the set of interfaces supported by the object and the semantics of the operations in those interfaces. The class includes objects that share a single address space, objects that are in separate address spaces on the same machine, and objects that are in separate address spaces on different machines (with, perhaps, different architectures). On the view that all objects are essentially the same kind of entity, these differences in relative location are merely an aspect of the implementation of the object. Indeed, the location of an object may change over time, as an object migrates from one machine to another or the implementation of the object changes.

It is the thesis of this note that this unified view of objects is mistaken. There are fundamental differences between the interactions of distributed objects and the interactions of non-distributed objects. Further, work in distributed object-oriented systems that is based on a model that ignores or denies these differences is doomed to failure, and could easily lead to an industry-wide rejection of the notion of distributed object-based systems.

A.1.1 Terminology

In what follows, we will talk about local and distributed computing. By *local computing* (local object invocation, etc.), we mean programs that are confined to a single address space. In contrast, we will use the term *distributed computing* (remote object invocation, etc.) to refer to programs that make calls to other address spaces, possibly on another machine. In the case of distributed computing, nothing is known about the recipient of the call (other than that it supports a particular interface). For example, the client of such a distributed object does not know the hardware architecture on which the recipient of the call is running, or the language in which the recipient was implemented.

Given the above characterizations of "local" and "distributed" computing, the categories are not exhaustive. There is a middle ground, in which calls are made from one address space to another but in which some characteristics of the called object are known. An important class of this sort consists of calls from one address space to another on the same machine; we will discuss these later in the paper.

A.2 The Vision of Unified Objects

There is an overall vision of distributed object-oriented computing in which, from the programmer's point of view, there is no essential distinction between objects that share an address space and objects that are on two machines with different architectures located on different continents. While this view can most recently be seen in such works as the Object Management Group's Common Object Request Broker Architecture (CORBA)[1], it has a history that includes such research systems as Arjuna[2], Emerald[3], and Clouds[4].

In such systems, an object, whether local or remote, is defined in terms of a set of interfaces declared in an interface definition language. The implementation of the object is independent of the interface and hidden from other objects. While the underlying mechanisms used to make a method call may differ depending on the location of the object, those mechanisms are hidden from the programmer who writes exactly the same code for either type of call, and the system takes care of delivery.

This vision can be seen as an extension of the goal of remote procedure call (RPC) systems to the object-oriented paradigm. RPC systems attempt to make cross-address space function calls look (to the client programmer) like local function calls. Extending this to the object-oriented programming paradigm allows papering over not just the marshalling of parameters and the unmarshalling of results (as is done in RPC systems) but also the locating and connecting to the tar-

get objects. Given the isolation of an object's implementation from clients of the object, the use of objects for distributed computing seems natural. Whether a given object invocation is local or remote is a function of the implementation of the objects being used, and could possibly change from one method invocation to another on any given object.

Implicit in this vision is that the system will be "objects all the way down"; that is, that all current invocations or calls for system services will be eventually converted into calls that might be to an object residing on some other machine. There is a single paradigm of object use and communication used no matter what the location of the object might be.

In actual practice, of course, a local member function call and a cross-continent object invocation are not the same thing. The vision is that developers write their applications so that the objects within the application are joined using the same programmatic glue as objects between applications, but it does not require that the two kinds of glue be implemented the same way. What is needed is a variety of implementation techniques, ranging from same-address-space implementations like Microsoft's Object Linking and Embedding[5] to typical network RPC; different needs for speed, security, reliability, and object co-location can be met by using the right "glue" implementation.

Writing a distributed application in this model proceeds in three phases. The first phase is to write the application without worrying about where objects are located and how their communication is implemented. The developer will simply strive for the natural and correct interface between objects. The system will choose reasonable defaults for object location, and depending on how performance-critical the application is, it may be possible to alpha test it with no further work. Such an approach will enforce a desirable separation between the abstract architecture of the application and any needed performance tuning.

The second phase is to tune performance by "concretizing" object locations and communication methods. At this stage, it may be necessary to use as yet unavailable tools to allow analysis of the communication patterns between objects, but it is certainly conceivable that such tools could be produced. Also during the second phase, the right set of interfaces to export to various clients—such as other applications—can be chosen. There is obviously tremendous flexibility here for the application developer. This seems to be the sort of development scenario that is being advocated in systems like Fresco[6], which claim that the decision to make an object local or remote can be put off until after initial system implementation.

The final phase is to test with "real bullets" (e.g., networks being partitioned, machines going down). Interfaces between carefully selected objects can be beefed up as necessary to deal with these sorts of partial failures introduced by distribution by adding replication, transactions, or whatever else is needed. The

exact set of these services can be determined only by experience that will be gained during the development of the system and the first applications that will work on the system.

A central part of the vision is that if an application is built using objects all the way down, in a proper object-oriented fashion, the right "fault points" at which to insert process or machine boundaries will emerge naturally. But if you initially make the wrong choices, they are very easy to change.

One conceptual justification for this vision is that whether a call is local or remote has no impact on the correctness of a program. If an object supports a particular interface, and the support of that interface is semantically correct, it makes no difference to the correctness of the program whether the operation is carried out within the same address space, on some other machine, or off-line by some other piece of equipment. Indeed, seeing location as a part of the implementation of an object and therefore as part of the state that an object hides from the outside world appears to be a natural extension of the object-oriented paradigm.

Such a system would enjoy many advantages. It would allow the task of software maintenance to be changed in a fundamental way. The granularity of change, and therefore of upgrade, could be changed from the level of the entire system (the current model) to the level of the individual object. As long as the interfaces between objects remain constant, the implementations of those objects can be altered at will. Remote services can be moved into an address space, and objects that share an address space can be split and moved to different machines, as local requirements and needs dictate. An object can be repaired and the repair installed without worry that the change will impact the other objects that make up the system. Indeed, this model appears to be the best way to get away from the "Big Wad of Software" model that currently is causing so much trouble.

This vision is centered around the following principles that may, at first, appear plausible:

- There is a single natural object-oriented design for a given application, regardless of the context in which that application will be deployed;
- Failure and performance issues are tied to the implementation of the components of an application, and consideration of these issues should be left out of an initial design; and
- The interface of an object is independent of the context in which that object is used.

Unfortunately, all of these principles are false. In what follows, we will show why these principles are mistaken, and why it is important to recognize the fundamental differences between distributed computing and local computing.

A.3 Déjà Vu All Over Again

For those of us either old enough to have experienced it or interested enough in the history of computing to have learned about it, the vision of unified objects is quite familiar. The desire to merge the programming and computational models of local and remote computing is not new.

Communications protocol development has tended to follow two paths. One path has emphasized integration with the current language model. The other path has emphasized solving the problems inherent in distributed computing. Both are necessary, and successful advances in distributed computing synthesize elements from both camps.

Historically, the language approach has been the less influential of the two camps. Every ten years (approximately), members of the language camp notice that the number of distributed applications is relatively small. They look at the programming interfaces and decide that the problem is that the programming model is not close enough to whatever programming model is currently in vogue (messages in the 1970s[7,8], procedure calls in the 1980s[9,10,11], and objects in the 1990s[1,2]). A furious bout of language and protocol design takes place and a new distributed computing paradigm is announced that is compliant with the latest programming model. After several years, the percentage of distributed applications is discovered not to have increased significantly, and the cycle begins anew.

A possible explanation for this cycle is that each round is an evolutionary stage for both the local and the distributed programming paradigm. The repetition of the pattern is a result of neither model being sufficient to encompass both activities at any previous stage. However, (this explanation continues) each iteration has brought us closer to a unification of the local and distributed computing models. The current iteration, based on the object-oriented approach to both local and distributed programming, will be the one that produces a single computational model that will suffice for both.

A less optimistic explanation of the failure of each attempt at unification holds that any such attempt will fail for the simple reason that programming distributed applications is not the same as programming non-distributed applications. Just making the communications paradigm the same as the language paradigm is insufficient to make programming distributed programs easier, because communicating between the parts of a distributed application is not the difficult part of that application.

The hard problems in distributed computing are not the problems of how to get things on and off the wire. The hard problems in distributed computing concern dealing with partial failure and the lack of a central resource manager. The hard problems in distributed computing concern insuring adequate performance and dealing with problems of concurrency. The hard problems have to do with dif-

ferences in memory access paradigms between local and distributed entities. People attempting to write distributed applications quickly discover that they are spending all of their efforts in these areas and not on the communications protocol programming interface.

This is not to argue against pleasant programming interfaces. However, the law of diminishing returns comes into play rather quickly. Even with a perfect programming model of complete transparency between "fine-grained" language-level objects and "larger-grained" distributed objects, the number of distributed applications would not be noticeably larger if these other problems have not been addressed.

All of this suggests that there is interesting and profitable work to be done in distributed computing, but it needs to be done at a much higher-level than that of "fine-grained" object integration. Providing developers with tools that help manage the complexity of handling the problems of distributed application development as opposed to the generic application development is an area that has been poorly addressed.

A.4 Local and Distributed Computing

The major differences between local and distributed computing concern the areas of latency, memory access, partial failure, and concurrency.[1] The difference in latency is the most obvious, but in many ways is the least fundamental. The often overlooked differences concerning memory access, partial failure, and concurrency are far more difficult to explain away, and the differences concerning partial failure and concurrency make unifying the local and remote computing models impossible without making unacceptable compromises.

A.4.1 Latency

The most obvious difference between a local object invocation and the invocation of an operation on a remote (or possibly remote) object has to do with the latency of the two calls. The difference between the two is currently between four and five orders of magnitude, and given the relative rates at which processor speed and network latency speeds are changing, the difference in the future promises to be at best no better, and will likely be worse. It is this disparity in efficiency that is often seen as the essential difference between local and distributed computing.

[1] We are not the first to notice these differences; indeed, they are clearly stated in [12].