CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Research Online @ ECU

Edith Cowan University
Research Online

Theses : Honours
Theses

2000

An investigation on sun microsystems Jini technology

Ferdina D. Soeyadi
Edith Cowan University

Follow this and additional works at: https://ro.ecu.edu.au/theses_hons

Part of the OS and Networks Commons

Recommended Citation
Soeyadi, F. D. (2000). *An investigation on sun microsystems Jini technology*. https://ro.ecu.edu.au/theses_hons/520

This Thesis is posted at Research Online.
https://ro.ecu.edu.au/theses_hons/520

Edith Cowan University

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

- Copyright owners are entitled to take legal action against persons who infringe their copyright.
- A reproduction of material that is protected by copyright may be a copyright infringement.
- A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

USE OF THESIS

This copy is the property of Edith Cowan University. However the literary rights of the author must also be respected. If any passage from this thesis is quoted or closely paraphrased in a paper or written work prepared by the user, the source of the passage must be acknowledged in the work. If the user desires to publish a paper or written work containing passages copied or closely paraphrased from this thesis, which passages would in total constitute an infringing copy for the purpose of the Copyright Act, he or she must first obtain the written permission of the author to do so.

AN INVESTIGATION ON SUN MICROSYSTEMS JINI™ TECHNOLOGY

By

Ferdina D. Soeyadi

0964841

Supervisor

Dr. James W. Millar

A Thesis Submitted in Partial Fulfilment of the

Requirements for the Award of

Bachelor of Science with Honours in Computer Science.

At the Faculty of Communication, Science and Health, Edith Cowan University, Mount Lawley Campus.

Date of submission:

March 31, 2000

ABSTRACT

Sun Microsystems introduced the Jini™ Technology as its vision of the future in networking, where services can be registered dynamically and be used easily regardless of their location in the network. This is an investigation of feasibility on such claims made by Sun regarding Jini ™ and comparisons with the directly similar Universal Plug and Play from Microsoft. The aim is to implement a simple application of the Jini™ technology in order to demonstrate its capabilities as a contribution to the distributed computing research.

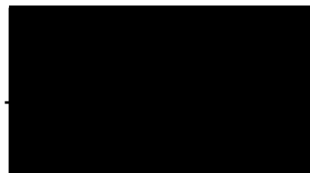
DECLARATION

I certify that this thesis does not, to the best of my knowledge and belief:

i. Incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education;

ii. Contain any material previously published or written by another person except where due reference is made in the text; or

iii. Contain any defamatory material.

Signature : 

Date : 4 April 2000

ACKNOWLEDGEMENT

First and foremost, I would like to thank God for everything. Thank you most of all to my supervisor, Dr. James W. Millar for his guidance and support throughout. To my family and friends back home, who could not wait for me to graduate. To both of my parents, especially. My dear departed father, Soeyadi Hadiwinoto, and my mother, Agustina Ferida, for their hard work that put me through school and make this possible. Last but not least, I would also like to extend my thanks to my best of friends, Robert Juhaniak, David Sutanto, and Jeshua Yee, for their endless encouragement, for giving me a lift, and for being my sleepless companion all this time. To everyone else that I forgot to mention, thank you.

TABLE OF CONTENTS

USE OF THESIS I

ABSTRACT III

DECLARATION IV

ACKNOWLEDGEMENT V

TABLE OF CONTENTS VI

CHAPTER ONE 1

Introduction 1

Background 1

Significance 2

Purpose 4

Research Questions 5

Question one. 5

Question two. 5

Definitions of Terms 6

Networking terminology. 6

Jini terminology. 7

UPnP terminology. 8

CHAPTER TWO 9

Literature Reviews 9

Jini™ Concept 11

Discovery, Join and Lookup 11

Jini Service 12

Jini Client 14

Note on Requirements 14
Leasing. 15
Remote events. 16
Transaction. 17
Universal Plug and Play Concept 20
Device discovery. 20
UPnP Service 20
UPnP Client 21
Requirements flexibility. 21
Service lifetime. 22
Events. 22
CHAPTER THREE 24
Materials and Methods 24
Procedures 24
Limitations 24
Equipment 25
CHAPTER FOUR 26
Project Results 26
A Simple Jini™ Implementation 26
Jini Service 26
Observation of Basic Service Requirements 27
Jini Client 29
Observation of Client requirements 29
An Experiment with Jini™ Lookup Discovery 31
Experiment Conclusions 41
Experiment Limitations 41
Jini and UPnP side by side 42

Language of implementation. 42

Networking requirements. 42

Service discovery. 43

Relaying query beyond the local network. 43

Device-to-device connectivity. 44

Device driver requirements. 44

Security Issues 44

CHAPTER FIVE 46

Conclusions 46

Question one. 47

Question two. 47

APPENDICES 49

Appendix A 49

Jini Packages 49

Legends 49

net.jini packages 49

net.jini packages 50

net.jini.discovery 51

net.jini.entry 52

net.jini.lookup 52

net.jini.lookup.entry 53

net.jini.admin 54

net.jini.space 54

net.jini.core packages 55

net.jini.core.discovery 56

net.jini.core.entry 56

net.jini.core.lookup 57

net.jini.core.event 58

net.jini.core.lease 59

net.jini.core.transaction 60

net.jini.core.transaction.server 61

LIST OF REFERENCES 62

CHAPTER ONE

Introduction

Background

The computing industry has recorded rapid advancement over the past 50 years. It started with mainframes and minicomputers where a single computer allowed sharing of applications and data for the whole department. Unfortunately, these computers are bulky and expensive. Then came the workstations and the personal computers (PC) giving computing power at lower cost and freedom to individuals but also isolating them from each other. This isolation was then overcome by connecting computers via networks. A whole new possibility was opened for individuals to make use of resources on different locations, as long as those resources are registered on the network. The problem is that the network is not for ordinary users to configure; the bulk of the work falls to the system administration to make sure that the resources on the network can be shared. The popularisation of the World Wide Web (WWW) gives further significance to the network computing technology. In addition, there is the embedded computer, which is a specialised computer system embedded as a part of a larger device, such as those within a microwave. The emergence of this technology allows for smart devices and the trend of recent years that demands computing portability and connectivity.

Sun Microsystems has been preaching about the idea that the network *is* the computer. The Web is already a model of exactly *one* single computer operating in the world being used by individuals all over the world. Obviously, connectivity and ease of use are bigger issues than ever for the current network computing.

Device connectivity, which is the ability for a device to link with another, necessitates that the different devices have a common protocol to be able to communicate. Between different computer systems, it is the task of the operating systems to manage this. Between devices, some of the latest technologies being offered

are the Universal Plug and Play from Microsoft, and the Jini™ Technology from Sun Microsystems. Both claim, not only to allow connectivity between different range of devices, but also between devices and computer systems.

Significance

The computing industry has been talking about how to make devices easy to connect and use without the need for the user's intervention [CNET, 2000; Intel, 2000; Lee, 1998; Shnier, 1996]. Microsoft, with the cooperation of Intel and other hardware manufacturer, then invented the Plug and Play (PnP) technology. Instead of the user re-configuring the computer to recognise a new device every time one is connected, PnP allows the computer to detect the new connection automatically. The operating system involved usually provides a set of device drivers from various hardware manufacturers and some generic ones needed by the computer to communicate with this new device. However, it still depends on the user to provide newer version of those device drivers when needed. This technology is also limited to computers, whereas the computing industry is moving towards different kinds of devices (entertainment devices, home electronic devices, etc.) to be connected and communicating with each other.

This is where Jini technology enters the picture. Jini manifests a new system architecture to allow a 'federation of devices', as termed by Sun, to be dynamically registered to the network whenever they are available and used by anyone, anywhere on the network with ease of administration. The federation of devices, resources, and users are called **djinn**. The devices in a Jini system are more aptly termed as **services** since they could be hardware, software or a combination of both. The Jini technology uses a special service called the **Lookup Service** that acts as a look up directory for the Jini system. It requires a bootstrap mechanism called the **Discovery and Join protocol** to access this lookup service. That is, when a service or a user first join the **djinn**, it must locate the lookup service (**discovery protocol**). Once a service finds a lookup service, it can register itself (**join protocol**) so that other services and the users can access the

service it has to offer. In the user's case, it locates the lookup service to find what services are available and to use the available services. Jini technology allows all these to be taken care of by the participating devices. All the user has to do is contact the lookup service and choose the service required. In addition, the services can provide a proxy object to act on its behalf that will be passed on to the user via the lookup service. This object downloading eliminates the need for platform-specific device drivers, which is one of the problems with connecting devices today.

As we can see, Jini is the offered solution from Sun that may contribute to the networked and distributed computing. Currently, it specifies that, since Jini is Java™ based, it requires a Java Virtual Machine (JVM) within the devices to be able to implement Jini. The choice is obvious because Java is a highly portable language that is also one of the important element that revolutionarise the Web. The fact that Java source code is compiled into the bytecode format allows this portability. Moreover, this bytecode is executed by a Java interpreter and a JVM, which exists for most operating systems today. Therefore, the use of Java allows Jini to run on different platforms that include JVM. Unfortunately, this may be a limitation to the range of Jini application. However, Sun is reported to be considering the use of a proxy agent for simple devices not capable of running a JVM [Middleton, 1999].

Although the Jini specification has been out as early as the start of 1999, it is of course not the only solution around. Microsoft has already been working for a while on an extension of its PnP technology called the Universal Plug and Play (UPnP). It has only provided some background documentation at the middle of the same year [Christensson, 1999; Microsoft, 1999a; Microsoft, 1999b]. Like its predecessor, UPnP aims to provide ease of use and connectivity, but this time extended to the network environment. So no longer is the system administrator needed to do all the configuring for every new device on the network. UPnP allows the device to obtain an **Internet Protocol** (IP) address from a **Dynamic Host Configuration Protocol** (DHCP) or try to configure its own IP address in the absent of a DHCP. The device will then announce itself to the computer or server of its availability at that address and listens for a device probes sent by the

users. Upon receiving a device request, its description is then sent out to the user in an Extended Markup Language (XML) format.

Again, this UPnP technology will largely contribute to the networked computing community. Although only aimed for different hardware devices. UPnP is based, unlike Jini, on many existing protocols and technology, such as the IP internetworking and the DHCP.

Purpose

This research is concerned with two of the new networking technologies available as the latest solutions to some of the infamous networking problems. These two technologies are the Jini™ Technology from Sun Microsystems and the Universal Plug and Play (UPnP) technology from Microsoft. They are chosen because their products offer close similarity in their solutions, thus, placing them in direct competition to each other.

The aim is to investigate what these technologies have to offer to the distributed computing scene. Both companies have their own claims regarding these technologies, but claims are not necessarily realisable. In addition, prospective developers are interested on how realisable are these technologies before they will take up Jini, UPnP, or both.

Research Questions

The questions that will be answered after this research are based on the claims made about Jini by Sun Microsystems, as follows:

Sun's claim:

"Jini technology promises to be a reality in the immediate future as architecture to enable connections between devices any time, anywhere." [Sun, 1999d]

Question one.

To what extent is the above claim immediately realisable?

Sun's claim:

"Jini technology provides simple mechanisms which enable devices to plug together to form an impromptu community—a community put together without any planning, installation, or human intervention." [Sun, 1999e]

The aim is to make connection of devices seamless to the users; however, this causes Jini to be comparable to the UPnP technology by Microsoft.

Question two.

How do the capabilities, ease of use, and reliability of Jini compare and contrast with the features of the UPnP technology?

Definitions of Terms

Both Jini and UPnP carry their own terminology, some of the important ones are reproduced here from their corresponding glossaries.

Networking terminology.

ARP: The Address Resolution Protocol (ARP) is used to translate an IP address to an Ethernet MAC address [Microsoft, 1999b]

DHCP: The Dynamic Host Configuration Protocol is a mechanism for providing devices with configuration information needed to access the Internet [Microsoft, 1999b]

DNS: The Domain Name System is a hierarchical and delegated database for the Internet host names and their mapping to IP addresses [Microsoft, 1999b]

IP: The Internet Protocol is the foundation protocol of the Internet that defines how a single message is sent from a source through zero or more routers to its final destination [Microsoft, 1999b].

UDP: The User Datagram Protocol is an IP-based protocol that provides support for the unreliable, unordered delivery of messages over IP [Microsoft, 1999b]

XML: Extensible Markup Language is a simplification of the Standard Generalised Markup Language (SGML), the textual, tag-based markup language intended for the creation of tags vocabularies that can be applied as the semantic markup to documents [Microsoft, 1999b].

Jini terminology.

discovering entity: Cooperating objects on the same host, that are starting, or are in the process of, obtaining references to Jini lookup services [Sun, 1999f].

discovery protocol: The protocol that rule the acquirement of a reference to one or more instances of the Jini lookup service [Sun, 1999f].

djinn: The group of devices, resources, and users joined by the Jini software infrastructure [Sun, 1999g].

join protocol: The protocol which allows entities to start communicating usefully with services in a djinn, through the Jini lookup service [Sun, 1999f].

lookup service: The Jini lookup service provides a central registry of service items, representing services, available within the djinn [Sun, 1999g]. It acts as a broker that allows users to locate and access the services in the djinn.

service registrar: A synonym for Jini Lookup service (see **lookup service**) [Sun, 1999g].

service: Something that can be used by a person, a program, or another service. Services will appear programmatically as objects in the Java programming language and have an interface, which defines the operations that can be requested of that service [Sun, 1999c].

UPnP terminology.

AutoIP: The enhancement to DHCP, allowing devices to configure an IP address for itself from a reserved range that is only used within a LAN [Microsoft, 1999b].

Multicast DNS: Rules for making normal DNS requests using multicast UDP [Microsoft, 1999b].

SSDP: The Simple Service Discover Protocol is the UPnP proposal for how to perform extremely simple discovery [Microsoft, 1999b].

CHAPTER TWO

Literature Reviews

What constitutes the Jini concept is not totally an innovation unique to the Jini technology. Jini is built on existing technologies, such as, the use of **Java Remote Method Invocation** (RMI) and **JavaSpace**.

Java RMI allows applications to use methods of other applications that exist on different machines. This is done by having a local object on the client side (called a *stub*) that takes care of the communication with the server-side object (called a *skeleton*) and handling of the data that are sent and received. The client actually invokes the local method on the stub, which maps the invocation to the remote method on a different machine. On the server side, the skeleton receives the request, invokes the requested method and returns the results to the client-side stub to unpack it for the client.

RMI allows transfer of code as well as data across the network by serialisation of the object to be transferred. When an object is serialised, it is converted into a byte sequence that can be sent over the wire. At its destination, the sequence of bytes is reconstitute or deserialised to make it into a whole object again. Being able to move codes around is the feature of RMI that enhances Jini. Although the Java RMI is available since JDK 1.1, Jini specially utilised the RMI enhancements that are only added in Java 1.2 [Edwards, 1999].

JavaSpace itself is based on Linda, a project from Yale University [Arnold, et al., 1999 p. 258-259; Clark, 1999; Edwards, 1999 p. 638-639; IEEE, 1998]. Linda provides a shared virtual space for processes in a parallel program so that all processes can exchange data by reading and writing them in the shared space [SCA, 1997].

JavaSpace extends on Linda by augmenting some of Java characteristics. For example, JavaSpace has the strong typing typical of Java and the ability to conduct searching based on class relationships, implemented interfaces, and known attributes. Although individual elements in Linda are also typed, as a whole unit they are not typed

like the objects in JavaSpace. In addition, because the entities that are stored in JavaSpace are objects, they not only contain data, but also their methods. Moreover, unlike Linda, several JavaSpace services are allowed to exist in a Jini environment, each with their own separate object storage area [Edwards, 1999]. JavaSpace supports the use of transaction in Jini.

Jini™ Concept

Using the above technologies, Jini expands and enhances them into a working model that comprise five key concepts:

- Discovery and Join
- Lookup
- Leasing
- Remote events
- Transaction

These components fall under three categories in the Jini model; the *infrastructure*, *programming model*, and the *services*. The infrastructure, or the core of the technology, consists of the **Discovery and Join** protocols, **Lookup,** and distributed security issues. The last three are what made up the programming model, which supports and made used by the infrastructure [Sun, 1999c]. Jini services are participants of the Jini network, enabled by the infrastructure and programming model, which have some resources to offer to the community. JavaSpaces in Jini are such service whose resources is the availability of storage space for used by other participants.

Discovery, Join and Lookup

The **Lookup** is a Jini service that acts like a broker that allows clients and services to see each other in a Jini federation network. Hence, without the existence of at least one lookup service, a Jini network will not function.

The **discovery protocol** describes the steps that any entity (either a client or a service) must initially take when connecting to the Jini network. Upon connecting to the network, if the entity want to participate in the Jini federation, it must first locate the lookup service by initiating the discovery process.

Employing either of the three discovery protocols, namely, **multicast request**, **multicast announcement**, and **unicast discovery** can be used to locate a lookup

service.

The *multicast request* is the attempt of a participating entity (client or service) in the djinn network to discover a lookup service by multicasting a request to all available lookup services in the network to announce itself. The *multicast announcement* is when a lookup service broadcasts its availability on the network. *Unicast discovery* protocol is used when an entity already knows the address of the lookup service it wants to join and, thus, it can directly query the lookup service. The unicast discovery is also employed by a lookup service after a multicast request to directly answer the entity's discovery request. Both multicast discovery protocols can be implemented by the use of the `LookupDiscovery` class from the `net.jini.discovery` package, whereas, the unicast discovery is implemented by the `LookupLocator` class from the `net.jini.core.discovery` package (see *Appendix A* for Jini packages diagram). The `LookupLocator` class uses the Uniform Resource Locator (URL) address of the target as an argument. The standard URL syntax takes the form of `protocol://host:port/data`. The protocol used is of course `jini`, the host is a Domain Name System (DNS) name or an IP address. The port is optional and defaults to 4160.

Jini Service

After a participating service entity locates the lookup service, the newly connected service will register itself with the lookup service. This is known as the **join protocol**. By registering with the lookup service, a service entity must provide the lookup service with its service proxy object and any attributes it has.

The lookup service maintains services registered to it through a set of **service items.** Programmatically, this set represents instances of the `ServiceItem` class. Each `ServiceItem` contains three elements, namely, the `serviceID` – its universal unique identifier (UUID), the service – either a Remote Method Invocation (RMI) stub or a proxy object, and the `attributeSets` – its set of attributes.

The `ServiceID` of a service item is initially generated by the lookup service when

the service first registers with the lookup. It is represented by the `ServiceID` class as a 128-bit value. Once obtained, the ID is reused whenever the service re-registers itself with the lookup.

A service object is an RMI stub if it is implemented as a remote object. Otherwise, it could be other object if the service uses the local proxy [Sun, 1999g]. The use of RMI technology allows the complete service object and its code to be passed on the network and downloaded for the client to use.

Jini services can attach attributes to its service proxy to associate extra descriptive info to the service. Service attributes are Java objects that implement the `Entry` interface from `net.jini.core.entry` package. The `Entry` interface is a subinterface of `Serializable` of `java.io` package. Because an attribute is a collection of Java objects, each field is serialised separately and independently, which allows for simpler searching [Edwards, 1999].

By attaching attributes, clients can search for a particular service based on certain criteria of the attributes. Attributes are extendable but the provided standards are `Name`, `Address`, `Location`, `Comment`, `ServiceInfo`, `ServiceType`, and `Status` attributes. The values of some attributes, such as the last three mentioned above, can not be changed by human intervention. That is, only the service can change them programmatically. For example, a printer that has run out of paper will change its `Status` attribute accordingly. Such attributes are said to be service-controlled and implement the `ServiceControlled` interface from `net.jini.lookup.entry` package. An administrator can change non-service-controlled attributes, such as, the `Location` attribute of the printer whenever it is moved, for example.

The set of attributes gives description of the service, such as, its name, owner, and location. This is represented by the `Entry` class. Upon registering itself with the lookup, a service provides its attributes to the lookup. When a user is looking for a particular service, giving attributes criteria allows narrowing down to those specific attributes. This is called service item matching and is implemented using an instance of the `ServiceTemplate` class, which has exactly the same elements as the `ServiceItem`

class. Hence, a service item matches a service template if all elements within the template match to the corresponding elements in the service item.

The elements of a service item are persistent across crashes and restarts. Two other elements of a service item that must survive such incidents are the set of groups the service is a member of and the set of specific lookup service with which it must registers itself whenever restarted [Sun, 1999f].

Jini Client

A participating client entity can, after locating the lookup service, query the lookup service for any services that match certain criteria, for instance, by querying the attributes or interface that a service might support. The client accomplishes this by providing a template of criteria attributes to pass as the query. This template is then checked for matches by the lookup service against the attributes of the Jini services that are registered with the lookup. Wild card is allowed by passing null fields in the template. On the other hand, non-null fields in the template must match the corresponding service attributes exactly. This means when serialised, they produced the same bytes [Edwards, 1999]

A match is found if the both the template and the service attributes are of the same class or subclass. When a match is found, the lookup service will pass on the service proxy object to the client so that the client can directly invoke the methods of the service. On the other hand, when a match is not found at the time of the query, the client has the option to ask the lookup service to notify it when a match occurs. This is done through the remote event notification mechanism, as will be discussed further.

Note on Requirements

To be able to participate in a Jini federation, there are several requirements that need to be met. A host must have an **IP** address, either a statically assigned to them or dynamically acquired through a **DHCP**. This a bit restricting for Jini, but UPnP has a work around for allowing IP address to be assigned temporarily when the DHCP is not

available, as will be discussed later. Jini also requires that there must be support for unicast **TCP** and multicast **UDP**, which are used during the discovery process. Unicast TCP is also used when utilising the Java RMI. Furthermore, a mechanism must be provided that allows for the dynamic downloading of RMI stubs or other codes needed. The typical mechanism is an **HTTP** server.

The reason for the last requirement is that the RMI, which is used by the Jini technology, allows for both data and code to be passed around the network. This is the solution provided by Jini, rather than upgrading device drivers manually, the drivers can be dynamically provided to its clients. In the case of a software service, it allows the client to obtain the interface 'driver' of the service needed to interact with that service [Venners, 1999]. In addition, with RMI, if an entity does not have all the classes that it needs it will download it automatically from the host's codebase. In Jini case, when a service version is upgraded, the client will automatically download the new stub for its use.

Leasing.

Again, the idea of leasing is not entirely new to network computing, thus not unique to Jini. A network participant uses the leasing concept when dynamically obtaining IP address from a DHCP. The DHCP then guarantees that the IP address will always be allocated to the requesting participant as long as the lease still holds [Droms, 1997; Wobus, 1998]. The lease for the IP address can be periodically renewed or released voluntarily.

Jini, on the other hand, expands on this concept of leasing to promote a self-healing network. The basic idea is the same, but the concept is applied to services and service consumers.

To obtain a lease means to show an interest in accessing and holding some kind of resource. Thus, whenever a Jini service registers with a lookup, it must negotiate a lease with the lookup to access the lookup registration. The negotiation for a lease is one

way. The service passes lease duration in milliseconds as a request. The lookup service, as the lease grantor, creates a lease object from the `net.jini.lease.Lease` interface. It then returns the proxy of the lease to the client. The returned lease is the agreed lease duration decided by the lease grantor, which is not necessarily what the client originally requested. The client, or in this case, the lease holder, can check the returned lease duration by calling the `getExpiration()` method of the lease object.

Like the IP address holder of DHCP client, a Jini client can continually show interest in holding the registration to a lookup service by periodically renewing its lease. It can also release the lease if needed be. This is achieved by calling the `renew()` and `cancel()` method of the lease object, respectively. The costs of the leasing model are that a lease holder must actively renew its leases, whereas, the grantor must actively check for expiration of leases [Edwards, 1999]. Fortunately, Jini allows leasing to be handled by a third party lease manager whose duty is to renew its clients' leases. In addition, leases can also be batched together using the `LeaseMap` interface of the `net.jini.lease` package, which extends `java.util.Map` from Java 1.2. Batched lease can be renewed and cancelled together, but whether particular lease can be batched or not depends on its implementation [Edwards, 1999].

The benefits of leasing outweigh the costs since it allows a self-healing network. That is, failure in the part of the client can be detected by the lookup service at utmost when the lease expires. Therefore, the shorter the duration of the lease, the faster a failure can be detected since the lease object that provides the connection between the lookup and the service is severed. Once the lease expires, the lookup service can reclaim back any resources used by the service, and forgets about irrelevant data or unwanted states left behind [Edwards, 1999; Sun, 1999i]. All this requires work only on the part of the lease grantor.

Remote events.

A Jini participant can register for remote event notification. Event notification in

Jini is based on Java. It differs greatly because Java events are mainly for local events on the same machine. This caused a lot of restriction, as discussed by Simon Roberts and Jon Byous [Roberts and Byous, 1999]. They stated that, Java assumes synchronous and reliable delivery of events when in distributed systems there is no guarantee. In addition, most Java source events are non-serialisable, thus, preventing the whole event object to pass as argument over the network [Roberts and Byous, 1999].

The Jini remote event model provides the solution and allows more simplicity. Unlike the Java model that requires different listener for different event type, any Jini event listener can receive any type of events. All of Jini remote events are subclasses of `net.jini.core.event.RemoteEvent`, which extends `Serializable`, and all Jini listeners implements the `net.jini.core.eventRemoteEventListener` interface. This characteristic allows Jini event listeners to be pipelined – one's output becomes another's input – allowing Jini to cater for specific application requirements when needed [Edwards, 1999]. In addition, like any Jini resources, Jini events are leased and thus carry the benefits of leasing as discussed above.

Transaction.

Transaction in computing is indispensable. By using transaction, multiple operations become one working unit, where all transaction participants can either succeed or fail together as a unit. The aim is to preserve the consistency of the operations by preventing partial success or failure.

The Jini specification provides a *two-phased commit protocol* for distributed transactions. The two-phased commit protocol guarantees that operations are consistently resolved by ensuring all transaction participants will eventually know whether to commit the transaction or abort it [Arnold, et al., 1999 p. 185-186]. Essentially the two-phased commit involves two stages called the '*prepare stage*' and the '*commit stage*'. At the first stage, all participants are made sure that they have finished computing and saved to a temporary storage, whatever results were requested before

going to the next stage. If all participants successfully completed the first stage, then they can proceed to the commit stage. At this stage, all participants copy their results to a permanent storage and report their success status. If all participants completed successfully the commit stage, then the transaction is also successful as a whole. A transaction manager coordinates all of these necessary steps.

Again, the transaction concept is neither new nor unique to Jini since database systems have been using transaction as well. Although, unlike a database transaction, Jini does not define the semantics of the implementations but left it to the individual participants involved [Edwards, 1999]. In Jini case, the participants are Jini services whose methods can be grouped as transaction. These services implement the `TransactionParticipant` interface of `net.jini.core.transaction.server` package. The transaction manager in Jini is a service as well, whose sole duty is to coordinate transaction. Transaction managers implement the `TransactionManager` interface instead of `TransactionParticipant` of the same package. A Jini transaction client is a Jini application which have a need to execute operations as transaction. The client must first retrieve a reference to a Jini transaction manager through the usual discovery and lookup process. Then it creates a `Transaction` object by calling the `TransactionFactory` from `net.jini.core.transaction`. The `Transaction` object is managed by the transaction manager and is passed to each participant as part of a method call.

As a service, Jini transaction managers are leased too, and therefore must be renewed until completion of the transaction. The lease in transaction only determine when transaction stops allowing participants to be added before attempting to start the two-phased commit protocol [Edwards, 1999].

Jini transactions are optional and only used by JavaSpace to coordinate operations across a common storage space. JavaSpace services in Jini implements the `net.jini.space.JavaSpace` interface in their proxies. As usual, these services are leased. The objects in JavaSpaces are of type `Entry`, which is also used as attribute objects in Jini services. Naturally, the mechanism of searching for particular objects in

JavaSpace is similar to querying service attributes. That is, template `Entry` object is also used, but multiple matches can not be returned. Where lookup service query attributes as a set of entries, JavaSpace only queries single entries.

JavaSpace has two types of storage, transient and persistent. In transient, the storage is only available as long as the service is running. For the latter, the storage data is recoverable because it is saved to a permanent storage.

Universal Plug and Play Concept

The specification for UPnP has been slow coming compared to Jini. Nonetheless, similar issues have been addressed and include the following concepts:

- Device discovery
- Service lifetime
- Events

Device discovery.

Similar to Jini, UPnP has some sort of a discovery protocol that allows discovery of devices on IP networks. The term used is Simple Service Discovery Protocol (SSDP).

With SSDP, the use of a lookup service mechanism such as used by Jini can be bypassed. However, such lookup mechanism still exists in UPnP to implement service discovery beyond the local area network. The term used is for such mechanism in UPnP is a **directory**. In the presence of a directory, it acts like the Jini lookup service, necessitating services to register with it so that it can act on its behalf, that is, listening, and answering requests from clients.

UPnP Service

A service must initially sends a multicast packet to announce its availability on the network. The packet contains information about the service, specifically, its identifier, location, and expiration time. Instead of using service ID and attributes as used by Jini, SSDP uses a single identifier, which is a unique pairing of a unique service name (USN) Uniform Resource Identifier (URI) and the service type URI. The location is the URL information of where to contact the service. The expiration time specifies the maximum time that information about the service is cached on the clients, which could be in any time units from seconds to years [Microsoft, 1999a; Microsoft, 1999c].

UPnP Client

A client looks for a service by sending a UDP multicast packet containing the service identifier of interest [Microsoft, 1999b]. Services can directly listens for and respond to such request if it has the specified identifier.

The UDP data is sent and received in HTTP format with special semantics [John, 1999]. Special message is embedded in the HTML, that is, either ANNOUNCE or OPTIONS message for announcement and querying, respectively.

The successful result of a query will return the URL of the XML file containing the service description. UPnP utilised XML to provide descriptive information of services and the capabilities information of smart objects. The features of a smart object are presented with XML, allowing the device to be manipulated. The use of XML style sheet (XLS) also allows different views to be presented to the client as required [Christensson, 1999]. The URL can be resolved to the service's IP address by the client and used to connect with the service.

UPnP enables the clients to control services through net browser if the browser learns how to talk to UPnP services. The service provided the learning process by uploading the XML file that describes the capabilities of the service. This means that with UPnP, invoking services can be done without any code being passed around, but only through the exchange of formatted data. To make it workable it would require that all devices communicate with common interfaces and protocols. Such service-driven auto-configuration capability is enabled by UPnP architectural component called the **Rehydrator,** whose job is convert between programming interfaces and protocols [Microsoft, 1999c].

Requirements flexibility.

Like Jini, UPnP uses the IP networking standard. In the case of Jini, it requires that the IP address of participating entities be assigned statically or dynamically using DHCP. UPnP requires its services to be able to operate in the absence of either. This is

where the **AutoIP** comes into the picture. It allows a service to automatically configure an IP address for itself in the absent of a DHCP. This is done because it has a range of IP addresses set aside that are use when DHCP is not available. It only keeps the current arbitrarily chosen address only after it uses the Address Resolution Protocol (ARP) to ensure that the address is not already being taken. Despite obtaining an IP address this way, it still continually checks for DHCP availability. If a DHCP is available, the IP address is obtained from the DHCP instead.

Of course, typically we prefer to use a DNS instead of an IP address to refer to a hostname. Because there is a possibility for a DNS server to be unavailable, UPnP uses **Multicast DNS** to allow services to listen for their names being requested and respond to such requests [Microsoft, 1999b].

Both of the above approaches are used to allow peer device-to-device connection. The network can still function without the assistance of a personal computer system [Microsoft, 1999b]. Neither of the above techniques is provided by Jini. Jini assumes IP address assignment mechanism is available. Further, it requires the existence of the lookup service to marshal interaction between the client and the service provider.

Service lifetime.

A service provides an expiration time as part of its announcement packet. Expiration information is analogous to the Jini leasing system, where the service must refresh the cache of the clients periodically to signal its continuing existence in the network. The difference is that the client can perform discovery rather than waiting for cache update [Microsoft, 1999a].

Events.

UPnP uses Generic Event Notification (GENA) over TCP/IP for its event notification purposes. It adds conventions for establishing relationships between devices,

addressing scheme of events delivery, and GENA leverages HTTP addressing and encapsulation [Microsoft, 1999c]. The GENA mechanism is similar to the Java event delivery mechanism but with different terminologies and underlying technology. For example, the event listener in GENA is called **subscription arbiter**, and hence also the term **subscriber** for the entity that subscribes to the event notification [Cohen, et al., 1999]. A subscriber negotiates subscription of event notification with a subscriber arbiter, which then will relay events of interest from a source to the subscriber.

In GENA all event-related communication is done through HTTP notification using multicast UDP. To subscribe to an event notification, a subscriber must send an HTTP SUBSCRIBE message, specifying the target subscriber arbiter, event type, and a callback information on how to contact the subscriber. Each event subscription has a timeout value and a unique subscription ID (SID) in the form of a URI, which is passed along in the HTTP notification header. The timeout value acts like the leasing mechanism in Jini event, which can be renewed by sending another SUBSCRIBE message with the particular SID to be renewed. Like Java event mechanism, the subscription arbiter can also relay events to another arbiter, if necessary, to be passed to multiple recipients.

CHAPTER THREE

Materials and Methods

Procedures

Both Jini and UPnP technology are still relatively new. Especially for UPnP, the major source of information is the Internet. A few books on Jini have been published and provided a great deal of help. In order to answer the research questions, as much relevant information as possible were collected and analysed as time constrained permits.

Development of a Jini application will require skill of the Java language and thorough understanding of the Jini specification. Proving some of Jini claims can be accomplished by running and experimenting with limited implementation of Jini technology, based on possible scenarios that should work as claimed.

Limitations

Jini devices require the presence of a JVM. At the time of research, no Jini-compatible JVM is available for the Windows CE™ operating system that is used by the PalmPilot, the initial implementation platform of choice. The reason is that the EmbeddedJava is yet to support Jini technology.

A Jini device may use a Jini chip to make it Jini-capable. Again, at the time of research, this chip is not yet on the market. There is a circulating idea on the Jini-Users mailing list on creating your own Jini-chip, but this requires hardware knowledge and more time.

Thus, this research taken the step of simulating a hardware service using software and/or development of software services that demonstrates the claimed capabilities of Jini.

Equipment

Equipment used for this research is listed below.

- Networked PCs, running Microsoft Windows 98 operating system.
- A text editor.
- A web server, Microsoft Personal Web Server 4.0.
- Java Development Kit (JDK) version 1.2.2 (the requirement for Jini technology is JDK 1.2 onward).
- Jini development kit version 1.0 (the latest available is version 1.1 alpha).

CHAPTER FOUR

Project Results

A Simple Jini™ Implementation

Jini Service

A Jini Service must supply an interface as a contract between the client and the service provider. The interface specifies all of the services that the Jini Service guarantees to provide to the client. Based on the service specification of the interface, the client can utilise the provided services without knowledge of the implementation details. As long as both sides agree on this interface, they can communicate together.

The service implementation is provided to the client by the proxy object. The service provider, upon performing the join protocol, provides this proxy object to the lookup service. The proxy object may carry out all the services implementation for the client or pass it to the actual service provider when necessary.

Basically, the minimum responsibilities that a service implementation must perform upon connecting to a Jini network is as follows:

- Declare that it implements the agreed service interface.
- Find one or more lookup services of interest to register with.
- Publish its proxy object, providing attributes as needed.
- Manage its leasing with the lookup service.

To be able to publish it proxy object, the service must also declare to implement `Serializable`. Being serialisable allows a service to be transported down a network socket as byte streams. This is a feature of RMI.

Managing leasing is optional for the service because it can be handled by a third-party service. Therefore, as long as there is another service in the Jini network that advertises it will manage other services' leasing for them, then individual services do no

need to manage it themselves.

As mentioned previously, an entity in a Jini network may register for event notification. This is useful because during a unicast discovery process, the service might not find a particular lookup service that it wants to join. Hence, another requirement that will prove useful is for the service to register for discovery event notification. That is, it can ask to be notified if the lookup service that was not running previously came online. To register for discovery event notification the service must have an instance that implements a `DiscoveryListener` and attach it to its own lookup discovery mechanism, which is an instance of a `LookupDiscovery`.

When performing lookup discovery, the discovering entity, either a client or a service provider, can specify the lookup service's group it is interested in. The discovering entity must have a permission to attempt discovery of each of the group it specified as part of its lookup discovery attempt. This discovery permission is controlled through a security policy file. Thus, it is necessary for the service to set its security manager within its implementation.

Observation of Basic Service Requirements

After reviewing the above requirements, the issues that must be tackled when implementing a Jini Service can be divided into three categories:

- What to do within the service's `main()` method.
- What to do within the service's constructor method.
- What to do when a discovery event occurs.

1. What to do within the service's main() method

The least that a `main()` method of a service should do is to create an instance of the service, and then start a thread so that it will not terminate at the end of `main()` execution.

2. What to do within the service's constructor method

The constructor method is responsible for creating an instance of the `ServiceItem`, that is the proxy object of the service. Next, it is important that it sets a security manager for the proxy. To perform the lookup discovery it must then create an instance of the `LookupDiscovery` class. Next, the service should create a discovery event listener that will notify the service whenever a lookup service of interest is found. The discovery event listener must implement `DiscoveryListener` and it must then be attached to the instance of `LookupDiscovery` created earlier.

3. What to do when a discovery event occurs

There are two types of discovery events that a service will be interested in. The events are when a lookup service is discovered, and when a lookup service needs to be discarded. As mentioned, the discovery listener must implement the `DiscoveryListener` interface. Accordingly, it has two methods that correspond to the two discovery events, `discovered()` and `discarded()` method. The `DiscoveryEvent` has a method called `getRegistrars()` that returns a set of lookup services of type `ServiceRegistrar` related to the either events.

When a discovered event occurs, the service should then register with each lookup service by calling the `register()` method of the `ServiceRegistrar`. The least that a service must provide when registering is its proxy object created previously within the constructor method. In addition, if it is managing its own leasing, the duration of the lease should also be provided.

The discarded event allows the service to discard each lookup service from the set returns by `getRegistrars()` that stop responding to it since discarding of a lookup service does not happen automatically.

Jini Client

A Jini Client must find the services it wants to use through the lookup service. Therefore, like the Jini Service, the client basically follows a common frame of work, as follows:

- Find one or more lookup services of interest.
- Query each lookup for the service(s) of interest.
- Start using the service(s).

As we can see, being client is obviously simpler than a service provider since it does not need to register with the lookup and does not need to provide the lookup with a proxy object. Like a service provider, it must also have permission to be able to attempt discovery of lookup service's group of interest.

Observation of Client requirements

Again, guided by the framework above, the issues that must be handled are the same with services:

- What to do within the client's `main()` method.
- What to do within the client's constructor method.
- What to do when a discovery event occurs.

1. What to do within the client's main() method

The client's `main()` method has the same responsibility as the service's `main()`. That is, to create an instance of the client and start the thread that runs the service.

2. What to do within the client's constructor method

Like a Jini Service, a client must set a security manager. In addition, it must create an instance of the `LookupDiscovery`. Again, it is useful for the client to install a discovery event listener so that it can react only when a lookup service of interest is

discovered. Once it has an instance of the listener, it can attach it to the `LookupDiscovery` instance.

The next thing to do is to find the exact match of the service(s) of interest by querying the lookup service. The more criteria it provides the narrower the search for the service becomes. The parameters to provide are the service ID of interest, the type of the service, and a set of attributes that the service should have. The client must thus create instances of one or combination of `ServiceID`, `ServiceType`, and `Entry` for each respective parameter, unless null is passed instead. Then, an instance of the `ServiceTemplate` must be created to contain those parameters.

3. What to do when a discovery event occurs

The only discovery event that a client might be interested in is the discovered event. When each lookup service of interest is discovered, it can query the lookup by calling its method `lookup()`, providing the criteria template. When matching service is found, the client can then start using the service. Typically, by creating an instance of the Java class `Object`, which is the superclass of most Jini classes. Then, this object is typed cast to the service's interface before being able to call the service's methods, as follows, `(ServiceInterface)InstanceOfClassObject.ServiceMethod()`.

An Experiment with Jini™ Lookup Discovery

To watch the discovery process in action, two Jini example programs are used. These are called the *DiscoveryExample* and the *Browser*, from the book *Core Jini* by W. Keith Edwards [Edwards, 1999] and the Jini development kit provided by Sun Microsystems, respectively.

The possible discovery scenarios are as follows:

- To discover an already running lookup service.
- To discover a newly available lookup service on the same host machine.
- To discover another newly available lookup service on a different host machine.

These scenarios are the very basic that the discovery protocol should be able to handle. Any Jini participants will not be able to work without first discovering a lookup service. Therefore, the least that the participant must be able to do is discover an existing lookup service anywhere in the local network. The aim is to test dynamic discovery of lookup services, that is, a Jini device should be able to be plugged in and detects online lookup services upon commencement of discovery process.

1. Discovering an existing lookup service

The screenshot below shows a service during a discovery process. The first window on the top left is the RMI daemon running after the web server is up and before a lookup service can be run. The second one in the middle is the lookup service itself, provided by the example that came with the Jini development kit from Sun. The last window at the bottom is the Jini service example, whose purpose is to continually attempt to discover available lookup services on the network until a key is pressed.

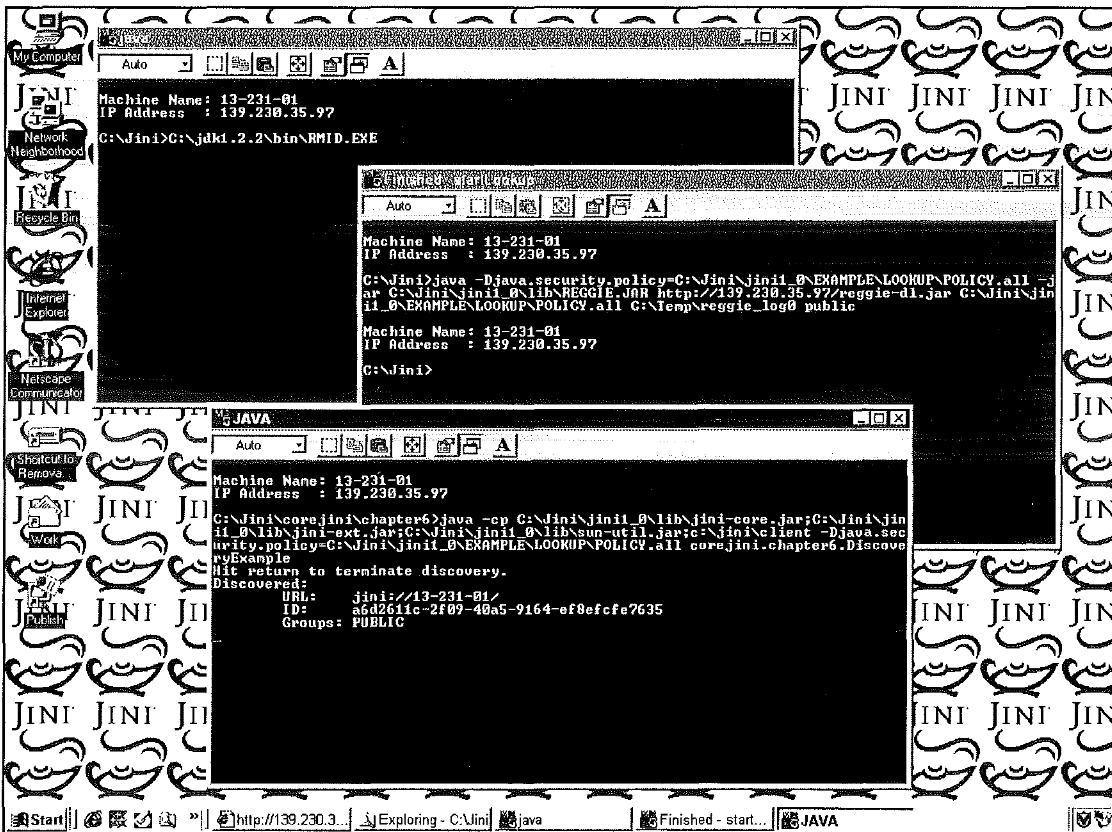


Figure 1 – The first lookup service is already running and consequently discovered by the discovering service entity.

A closer look at the last window shows the URL of the lookup service (in this case, the machine's name), the lookup service's unique service ID, and the groups it belongs to (in this case, the public group).

```
Machine Name: 13-231-01
IP Address   : 139.230.35.97

C:\Jini\corejini\chapter6>java -cp C:\Jini\jini1_0\lib\jini-core.jar;C:\Jini\jini1_0\lib\jini-ext.jar;C:\Jini\jini1_0\lib\sun-util.jar;c:\jini\client -Djava.security.policy=C:\Jini\jini1_0\EXAMPLE\LOOKUP\POLICY.all corejini.chapter6.DiscoveryExample
Hit return to terminate discovery.
Discovered:
        URL:    jini://13-231-01/
        ID:     a6d2611c-2f09-40a5-9164-ef8efcfe7635
        Groups: PUBLIC
_
```

Figure 3 – Existing lookup service is discovered instantly by this Jini service.

For the above first experiment, the first lookup service was already running before the discovery protocol commences. What would happen if a new lookup service went online? Since the example service above registers for discovery event notification, it should be able to discover new lookups too.

2. Discovering a newly online lookup service on the same host machine

The second scenario of the experiment is tested by running another lookup service on the same host while the service example (the *DiscoveryExample*) is still on its discovery mode, as shown below.

Figure 4 – The second lookup service is run and consequently discovered too by the discovering service entity.

The newly online lookup service has been successfully discovered by the example service. Note the URL of the lookup service. The first lookup service uses the default port number of 8080, which need not be specified. Every other lookup services that are on the same host will be automatically given an arbitrary free port number. At this instance, it is the port number 1156.

```
Machine Name: 13-231-01
IP Address   : 139.230.35.97

C:\Jini\corejini\chapter6>java -cp C:\Jini\jini1_0\lib\jini-core.jar;C:\Jini\jin
i1_0\lib\jini-ext.jar;C:\Jini\jini1_0\lib\sun-util.jar;c:\jini\client -Djava.sec
urity.policy=C:\Jini\jini1_0\EXAMPLE\LOOKUP\POLICY.all corejini.chapter6.Discove
ryExample
Hit return to terminate discovery.
Discovered:
        URL:    jini://13-231-01/
        ID:     a6d2611c-2f09-40a5-9164-ef8efcfe7635
        Groups: PUBLIC
Discovered:
        URL:    jini://13-231-01:1156/
        ID:     3e85150b-dd98-49ea-a2c5-9d4c3ae4aa5a
        Groups: PUBLIC
_
```

Figure 5 – The second lookup service is run on the same host and consequently assigned non-default port number.

At this point, another discovering entity from different service example program is started. This is to make sure that in the third scenario, the result is not based solely on the ability of the *DiscoveryExample* from *Core Jini* book to discover lookup services.

The second discovering entity program chosen is the one that comes with the Jini development kit from Sun. This program (the *Browser*) does the same thing as the *DiscoveryExample*, which is to discover online lookup services on the network.

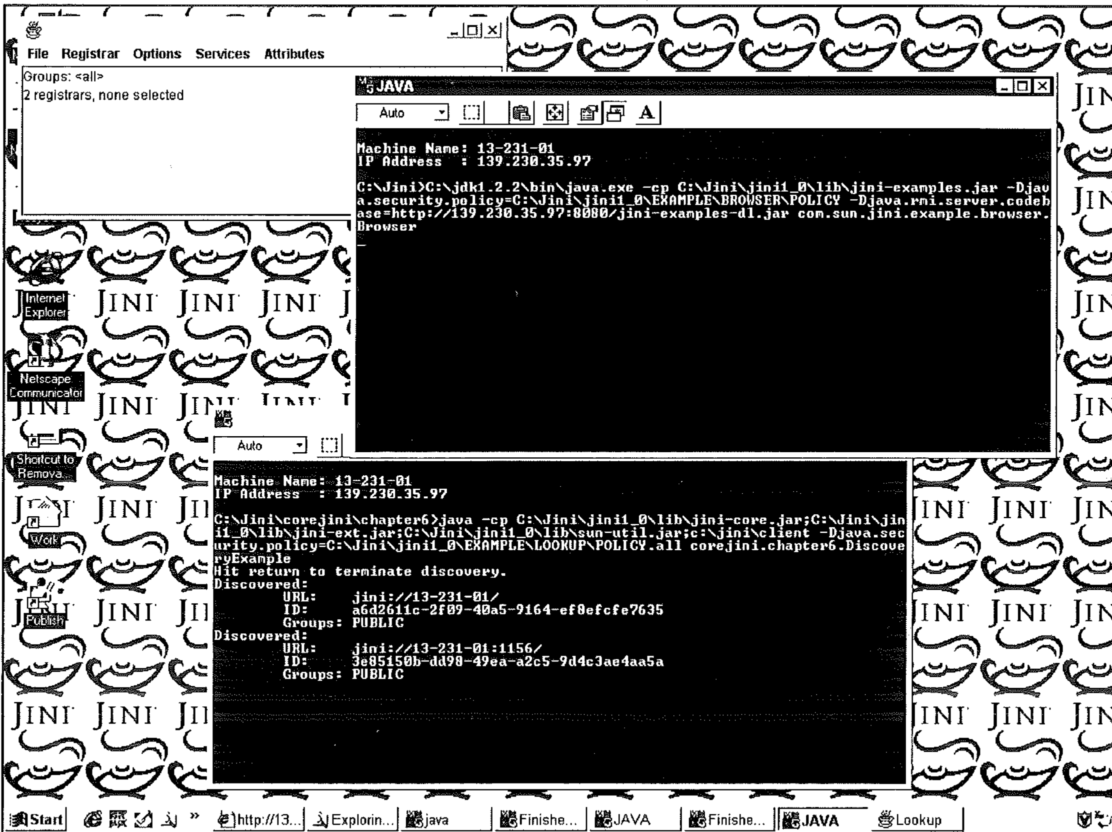


Figure 6 – The second discovering entity discovers two lookup services too.

A closer look shows that the *Browser* too discovers the same information about the lookup services as the *DiscoveryExample*.

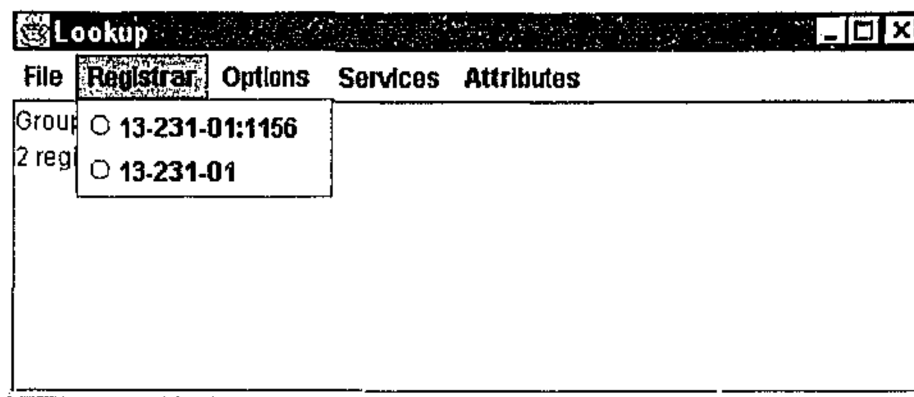


Figure 7 – Both the first lookup service on the default port and the second on port number 1156 is discovered by the second discovering entity.

The conclusion so far is that services commencing the discovery protocol will find any lookup services already running and newly run on the same host machine.

The last scenario is to test whether a newly run lookup service on another machine can also be dynamically discovered.

3. Discovering a newly online lookup service on a different host machine

When another lookup service is run, this time on a different host machine, it turns out that it was not discovered by the *DiscoveryExample* on the first host machine. A second instance of the *DiscoveryExample* is then run on the second machine and it successfully discovered all three lookup services currently running on the network, that is, on the first and second machines.

The figure below shows the snapshot of the second machine after running the third lookup service and the *DiscoveryExample* that detects all three lookups.

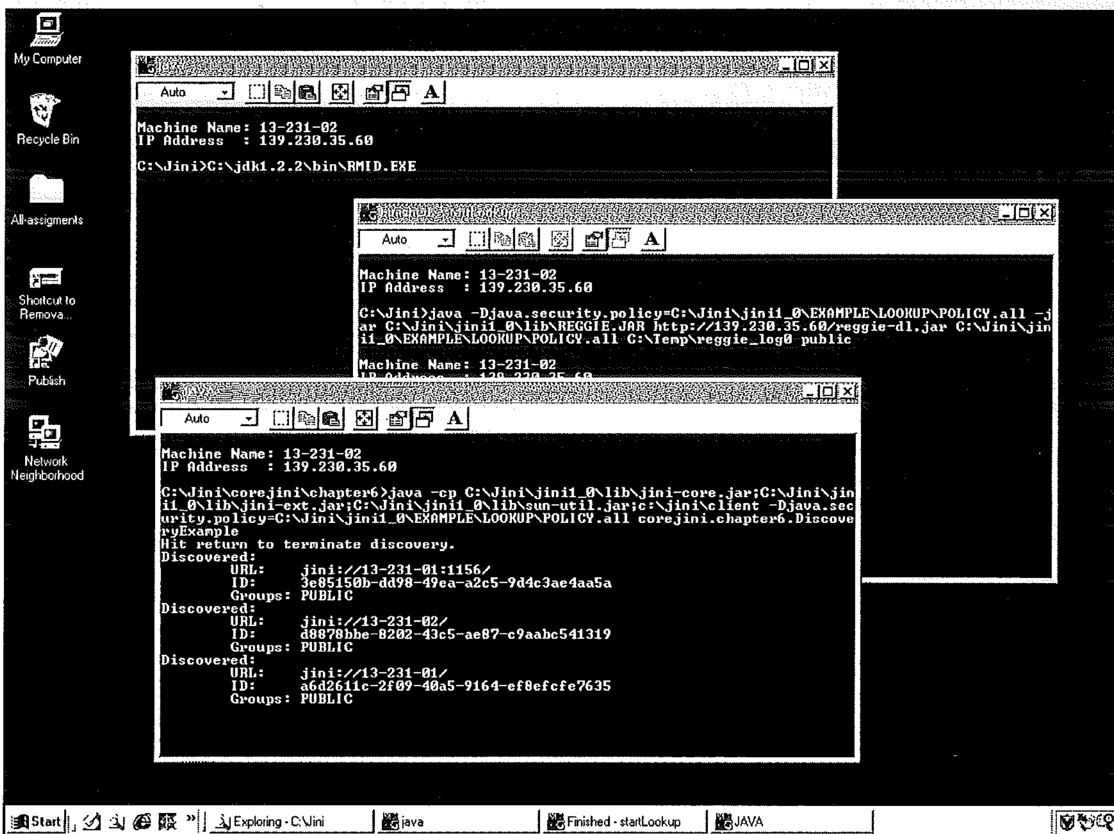


Figure 8 – The third lookup service is run on a different host machine and is only discovered by another instance of a discovering entity on that second machine.

The snapshot of the first machine is the same as the previous desktop snapshot, so there is no reason to reproduce it here.

The *DiscoveryExample*'s output on the second machine shows that it discovered the first two lookup services (both on machine '13-231-01') and the third lookup service (on machine '13-231-02') in an arbitrary order.

```
Machine Name: 13-231-02
IP Address  : 139.230.35.60

C:\Jini\corejini\chapter6>java -cp C:\Jini\jini1_0\lib\jini-core.jar;C:\Jini\jin
i1_0\lib\jini-ext.jar;C:\Jini\jini1_0\lib\sun-util.jar;c:\jini\client -Djava.sec
urity.policy=C:\Jini\jini1_0\EXAMPLE\LOOKUP\POLICY.all corejini.chapter6.Discove
ryExample
Hit return to terminate discovery.
Discovered:
        URL:    jini://13-231-01:1156/
        ID:     3e85150b-dd98-49ea-a2c5-9d4c3ae4aa5a
        Groups: PUBLIC
Discovered:
        URL:    jini://13-231-02/
        ID:     d8878bbe-8202-43c5-ae87-c9aabc541319
        Groups: PUBLIC
Discovered:
        URL:    jini://13-231-01/
        ID:     a6d2611c-2f09-40a5-9164-ef8efcfe7635
        Groups: PUBLIC
_
```

Figure 9 – The example service running on the second machine discovered all three already running lookup services on the network.

Note that the *DiscoveryExample* will detect all three lookup services when restarted regardless of the host machines.

Since the *DiscoveryExample* program from the *Core Jini* book can not restart the discovery process unless the program itself is restarted, it is up to the *Browser* from Sun to do so.

Only after restarting the discovery process that the Browser is then able to detect all three lookup services, also known as registrars.

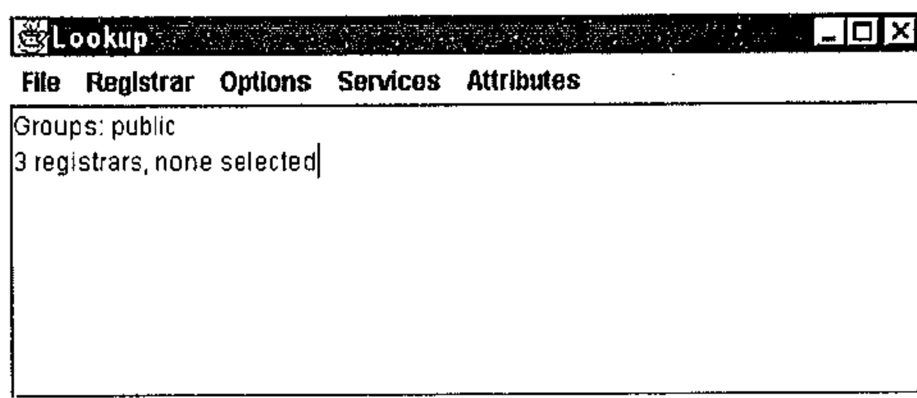


Figure 10 – After restarting the discovery process, all three lookup services on the network are detected accordingly.

The details of the lookup services are as shown on this next screen shot.

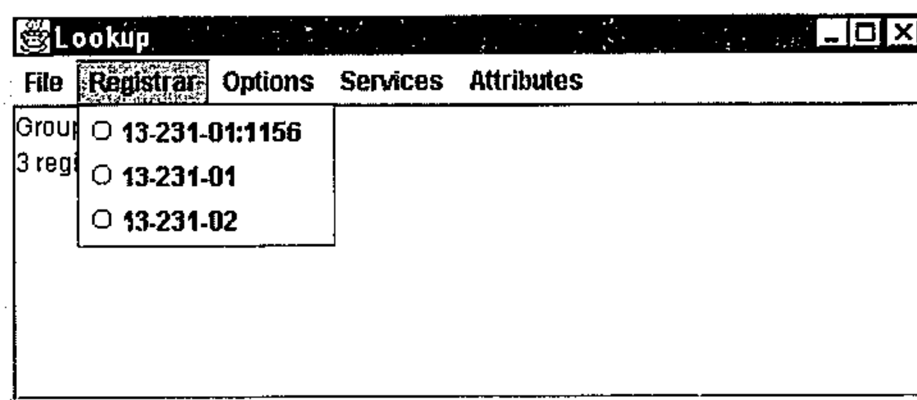


Figure 11 – Details of all three lookup services on the network.

Experiment Conclusions

This experiment shows that any well-configured Jini participants (both services and clients) will at least successfully discover all existing lookup services within a network and new ones only within the same host.

Unless a client can not find a service it is looking for within the lookup services it has already discovered, it does not need to detect new ones. Even if the need arise, restarting the discovery process may be done periodically within the implementation of the client as necessary.

W. Keith Edwards concurs with this view when discussing the need for lease renewal as a requirement for 'well-behaved' services. He stated that:

> *"If the lookup service was discovered through unicast discovery... then the service should try periodically to reconnect. ...since these lookup services are likely to be on a different network, they will not be discovered automatically via the serendipitous forms of discovery."* [Edwards, 1999, p. 278]

Note that in this experiment, although within the same network, multicast discovery only worked when the discovery process was restarted.

In conclusion, when Jini plays the role of the new plug-and-play device, it will run successfully with the above set up.

Experiment Limitations

Limitation of this experiment includes the fact that the flaws maybe on the part of the discovering service examples used. Undetected flaws in the software and/or network setup may also contribute to failures or misleading results.

Language of implementation.

UPnP utilised XML to provide descriptive information of services but apart from that it is language-neutral [Microsoft, 1999d]. The advantage of using XML is the ability to provide different views for clients using the XML style language XLS. Moreover, XML could become the next standard of markup language after HTML.

Jini technology is based on Java 1.2 technology otherwise also called Java 2[1]. In addition, it depends on the use of RMI and JavaSpaces. Although, this is not too much of a restriction since other programming languages can be used instead of or with Java [Sun, 1999h p. 5]. Specifically, by delegating the Java-specific functionality to a third party application if needed. This includes eliminating the need for a device to have its own Java virtual machine.

Both technologies use base languages that are considered highly portable. The difference is that XML might be relatively easier to learn than Java because of its close relation with the popular HTML.

Networking requirements.

The networking requirement for both technologies is the use of IP networking standard. Jini requires that the IP addresses of participating entities be assigned either statically or dynamically with DHCP. This reliance on DHCP is the gap that can be filled with the auto-configuration mechanisms provided by UPnP.

Although UPnP still primarily uses DHCP, in the absence of such IP-assigning authority, services can use the AutoIP to acquire IP addresses. With AutoIP, the IP addresses assigned are from a reserved range that is only be used in a local network [Cole, 1999]. The AutoIP also checks for the availability of a DHCP periodically to let it

---

[1] The Java 2 name applies to Java 1.2 product, as announced by Sun on December 1998[Sun, 1999b].

take over in assigning proper IP addresses.

Service discovery.

UPnP has SSDP as its own discovery protocol. For searching purposes, unlike Jini which use service attributes as criteria, SSDP uses a single Uniform Resource Identifier (URI), which specifies the profile of the service. The XML description used by the service is not even examined until discovery [John, 1999].

The client sends UDP multicast packet with the service identifier to find a service. If a directory exists it can act as a broker. Nevertheless, all services can directly listen for requests but only those with identifier that corresponds to the one specified in the request can respond. Microsoft maintains that the simple query mechanism is preferable than a full blown name-value pair searching such as provided by Jini attributes and interface support searching [Microsoft, 1999a].

The use of a lookup service mechanism as such used by Jini allows UPnP to implement service discovery beyond the local area network. Like the Jini lookup service, a UPnP directory acts like the broker between clients and services. Thus, a service has to register with the UPnP directory so that the directory can listen and answer requests made by clients.

Relaying query beyond the local network.

Jini does not specify how to the lookup service can relay a query to a neighbouring network if a sought service is unavailable locally. A solution would be to have the lookup service registered to other lookups outside the local network that are members of the same groups. The lookup can the register all services of the other lookups with itself to allow greater possibility of query match. W. K. Edwards provides a sample solution which is to have a dedicated service that does just that [Edwards, 1999 p. 330-338]. He calls this service a 'lookup service tunnel'. Such service actively finds all services registered in one lookup and registers them with another lookup.

Where Jini leaves it to the developer to provide the solution of relaying query,

UPnP provides its own. As mentioned above, UPnP uses the SSDP to query other directories beyond the local area network. This includes service query for wide area network and the Internet.

Device-to-device connectivity.

Jini and UPnP support peer device-to-device connection. In UPnP, the network can still function without the assistance of a personal computer system [Microsoft, 1999].

Jini, on the other hand, assumes that an IP address assignment mechanism is available. Jini also requires that a lookup service is available to marshal interaction between a client and a service provider.

Device driver requirements.

Jini does not eliminate the need for device driver altogether. Instead, it allows for dynamic downloading of the driver using RMI. Rekesh John [John, 1999] argues that this is not as simple as it sounds. Manufacturers must first agree on a standard for the methods in RMI interfaces with the device.

UPnP specified that the client can interact with a device through a net browser after the device-driven auto-configuration [Microsoft, 1999c]. But unless both client and device uses a standardised protocol then a device driver is still needed as noted by R. John [John, 1999] and Alec Saunders from Microsoft, who was quoted by John Charles [Charles, 1999].

Security Issues

The most critical issue in distributed computing is on security. Unfortunately, the UPnP has not provided sufficient details for a comparable presentation. Although, it does declare that since auto-configuration process only concerns with the exchange of formatted data, there is less chance of a breach through hostile code [Microsoft, 1999c].

On the other hand, security in Jini technology is based on Java 2 and RMI. Java 2 has a more fine-grained security approach with its security manager compared to its previous version. The execution of a code that is loaded into a machine is constrained by

its security policy. If an object attempts to do something that is not defined in its the security policy, a security exception will be thrown. A class can also be signed or unsigned. Signed classes uses the security policy defined in their certificate whereas, unsigned ones use the general security policy.

An article by Charles Crichton, et al. [Crichton, et al., 1999], identifies further the security loopholes in Jini. It said that the Java security manager is not adequate in a Jini environment. For example, when a user can not find a matching service within the local network, he might decide to look for the service through a lookup service on an external network. He then would inadvertently allow access to untrusted code through the network firewall. The result would be disastrous since untrusted code with local network access can scan local ports and send this information through the firewall. The solution for this is, of course, not to allow untrusted code access to the network at all.

RMI allows for passing code across the network. This is done through the serialisation mechanism, which converts an object into a byte stream that can be passed down the socket of a network. When deserialised at the target location, the object is converted back to its original form. Meanwhile, in a serialised form, the Java object's security restriction to private, package protected, or protected fields no longer exists [Sun, 1999a]. Anyone who has access to the stream can read, alter, and reconstitute the object without the Java security. This security loophole can not be dealt with the security manager because it is part of the runtime environment and therefore is dependent on the Java virtual machine. Because serialisation is to facilitate code mobility around different machines, it can not be tied down to such dependency. Other security breaches that can occur on object serialisation are also discussed by Charles Crichton, et al. [Crichton, et al., 1999].

CHAPTER FIVE

Conclusions

The innovation of Jini is found in the way that existing concepts and technologies were composed together.

Some of Jini strengths and weaknesses can be attributed because of its close relation to Java. In particular is Jini reliance on Java security system, portability, and object mobility with RMI. Further more, Jini performance will also depends on Java, whose performance has been questioned by some developers. Sun Microsystems is also yet to release embedded version of Java that supports Jini. Since Jini relies some of its core features on Java, any delay in the Java development means delay in Jini development as well.

Fortunately, Jini is relatively easier to understand and implement compared to UPnP. Its specification was released sooner which gives Jini a leverage in creating a pool of developer community. Therefore, Jini has already been under various implementation stages at this time by researchers and vendors alike [Flowwworks, 1999; MirrorWorlds, 2000]. The Jini community is already actively working on standardisation issues on Jini. This includes the interface standardisation of all common Jini services but to cover all possible services and get everyone to agree on a standard will take time.

All new technology needs time and supports to gain acceptance before it can become officially adopted as standards. Like Java, many of the industry experts believe that Jini will probably takes around three to five years before it truly becomes ubiquitous [Plummer, 1999a; Plummer, 1999b; Plummer, 1999c]. Nevertheless, because of Microsoft dominance in the industry, anyone with the Windows operating system will eventually live with UPnP. If the UPnP concept of invoking services through XML succeed, then the undeniable presence of worldwide Internet community will further leverage UPnP position. However, to have both technologies working and competing side by side is not an impossible scenario. The ChaiServer technology from Hewlett-

Packard claimed to be able to bridge between UPnP and Jini enabled devices[Brody, 1999; Gage, 1999]. Such bridging technology will let the competition between Jini and UPnP continue.

And so in conclusion, the research questions can be seen to be answered as the followings detailed.

Sun's claim:

"Jini technology promises to be a reality in the immediate future as architecture to enable connections between devices any time, anywhere." [Sun, 1999d]

Question one.

To what extent is the above claim immediately realisable?

Apart from the known Jini applications implemented by developers mentioned above, Jini services implementation has been slow, isolated and of a smaller scale than hoped. Aoption process is still slow and many in the industry is still waiting on how UPnP will be implemented because of the Microsoft influence. In reality, Jini technology is not realisable as immediate as it claims.

Sun's claim:

"Jini technology provides simple mechanisms which enable devices to plug together to form an impromptu community—a community put together without any planning, installation, or human intervention." [Sun, 1999e]

The aim is to make connection of devices seamless to the users; however, this causes Jini to be comparable to the UPnP technology by Microsoft.

Question two.

How do the capabilities, ease of use, and reliability of Jini compare and contrast with the features of the UPnP technology?

The capabilities of Jini are very parallel to that of UPnP as discussed under the comparison section. Both aim to network various kinds of devices, including household appliances that are easy to set up and maintain. The difference is how each implements their solution. Jini is Java-centric, whereas, UPnP is not tied to a particular programming language but uses standardised delivery mechanisms, such as XML and HTTP.

Reliability-wise they also depends on their underlying technologies. Java security mechanism, despite its weaknesses, seems more assuring for now because UPnP security issues are yet to be publicised.

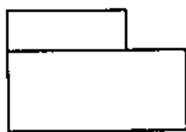
Ease of use, in term of implementation, Jini is more readily accessible at the moment because its development kit was published earlier and also because of its open source-type licensing.

APPENDICES

Appendix A

Jini Packages

Legends



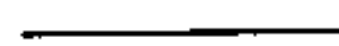
Package

Class

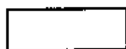
Interface

Inheritance

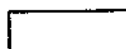
Dependency



Realisation

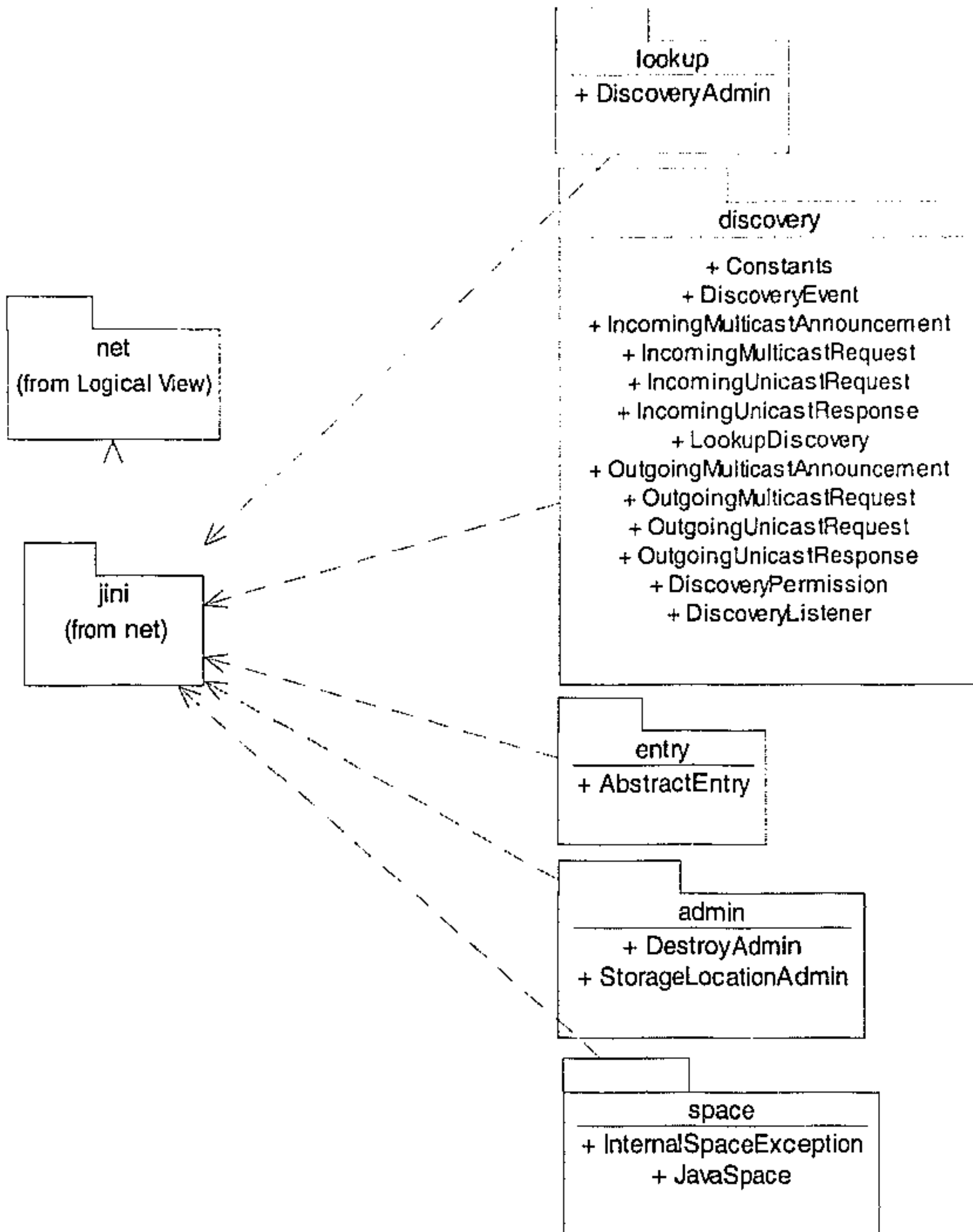


Jini package

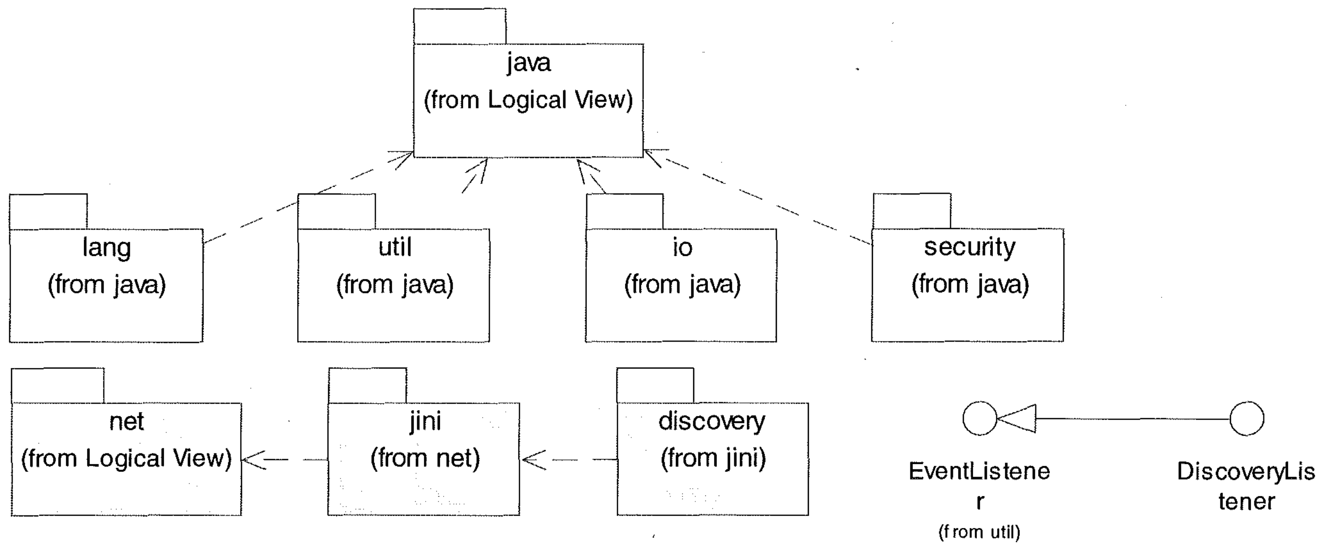


Java package

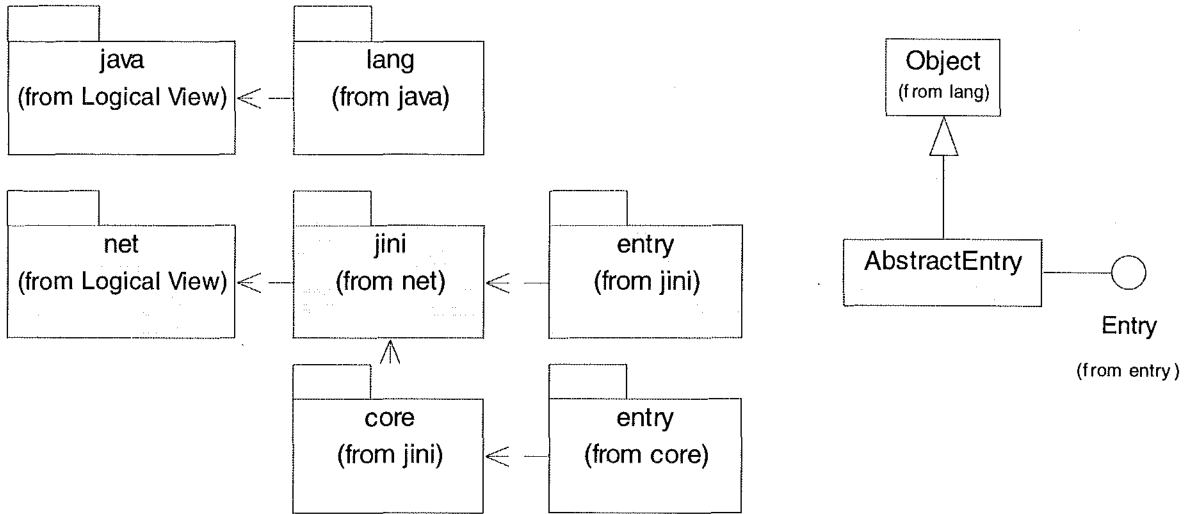
net.jini packages



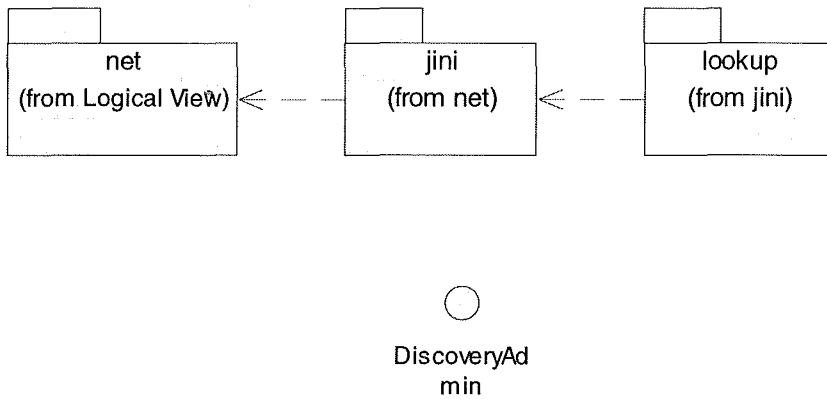
net.jini.discovery



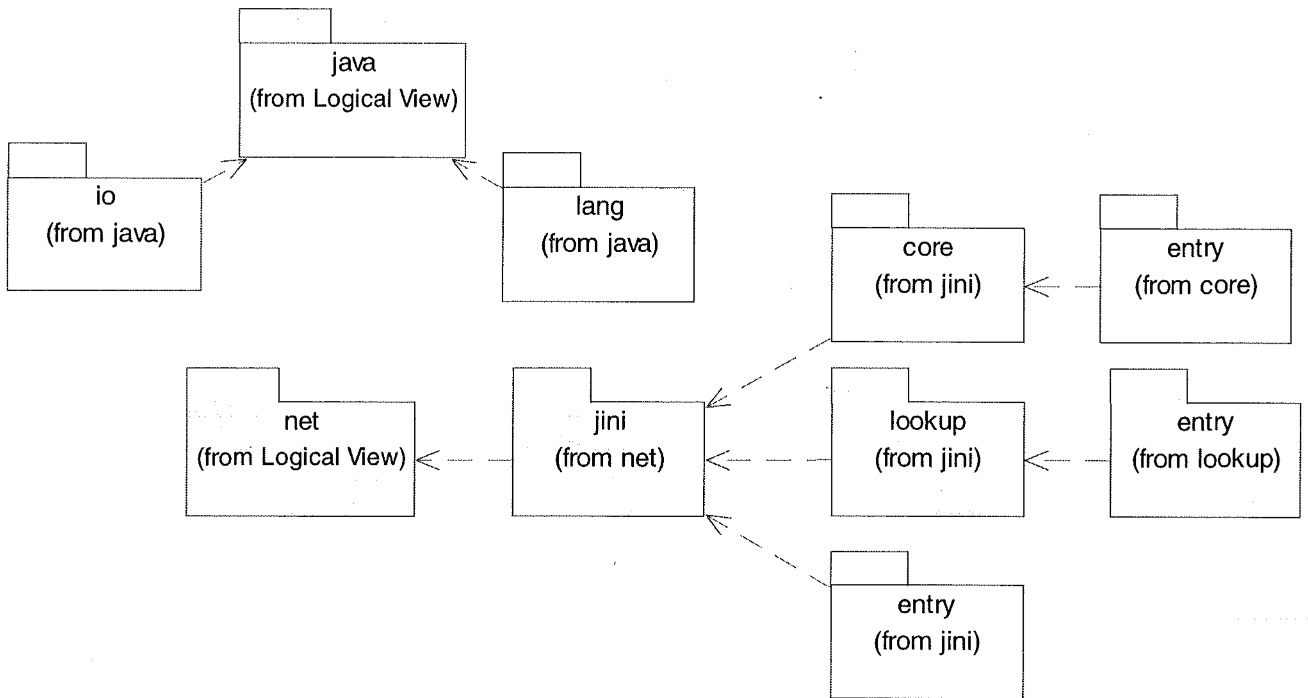
net.jini.entry



net.jini.lookup



net.jini.lookup.entry



ServiceControlled

Address
Comment
Location
Name
ServiceInfo
ServiceType
Status

AbstractEntry
(from entry)

Entry
(from entry)

AddressBean
EntryBean
Serializable

CommentBean
EntryBean
Serializable

LocationBean
EntryBean
Serializable

NameBean
EntryBean
Serializable

Object
(from lang)

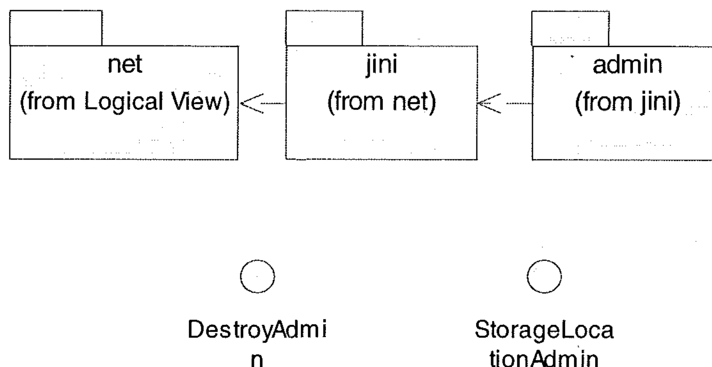
ServiceInfoBean
EntryBean
Serializable

StatusBean
EntryBean
Serializable
(from io)

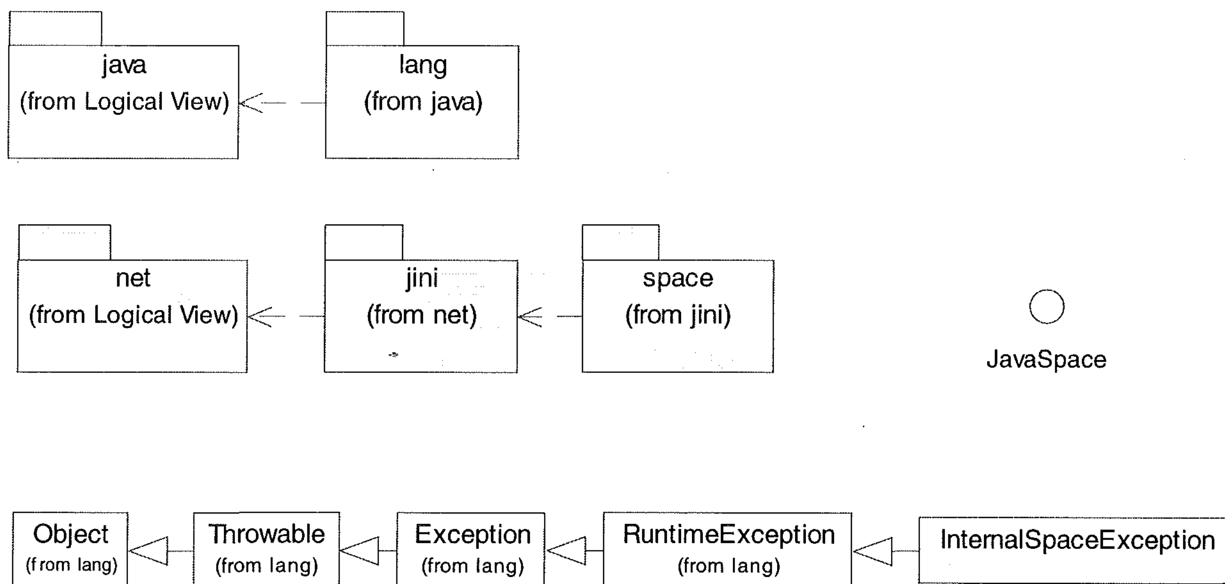
StatusType
Serializable
(from io)

EntryBeans

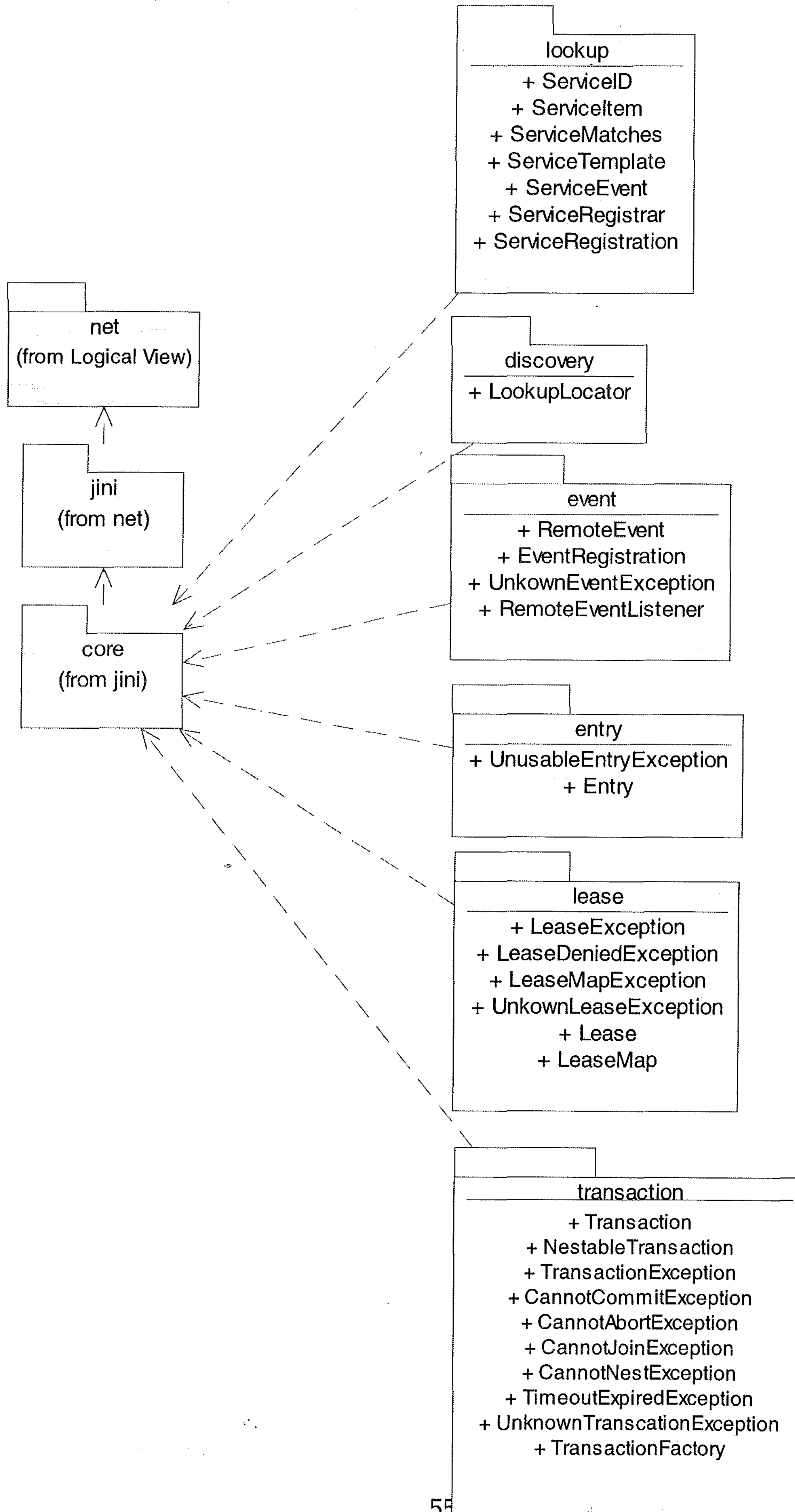
net.jini.admin



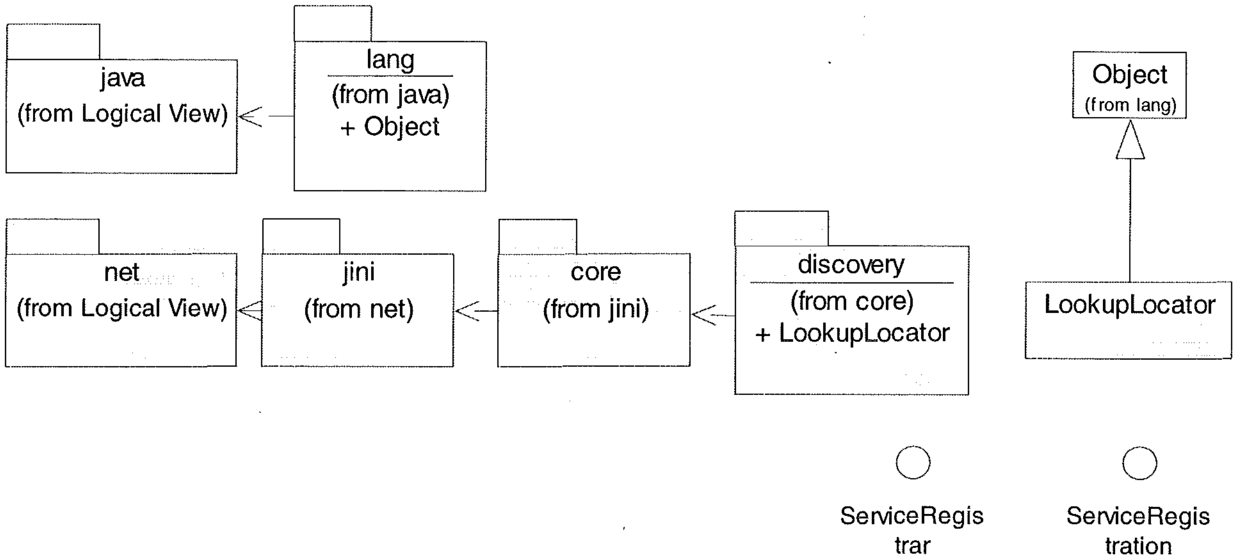
net.jini.space



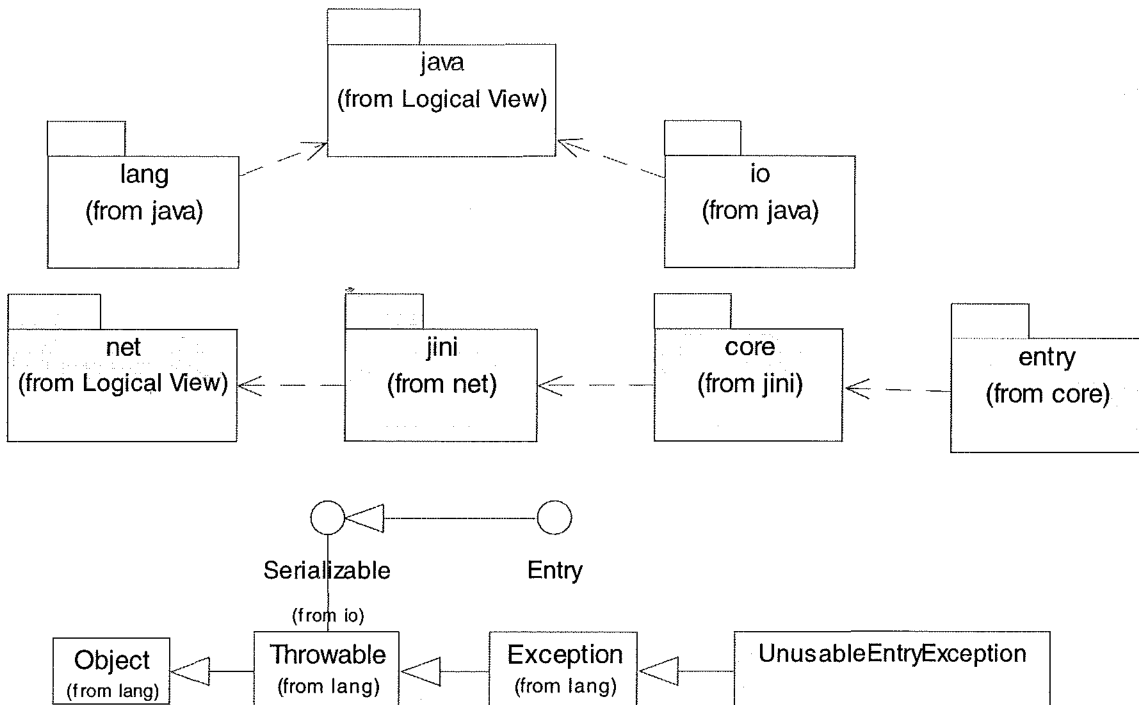
net.jini.core packages



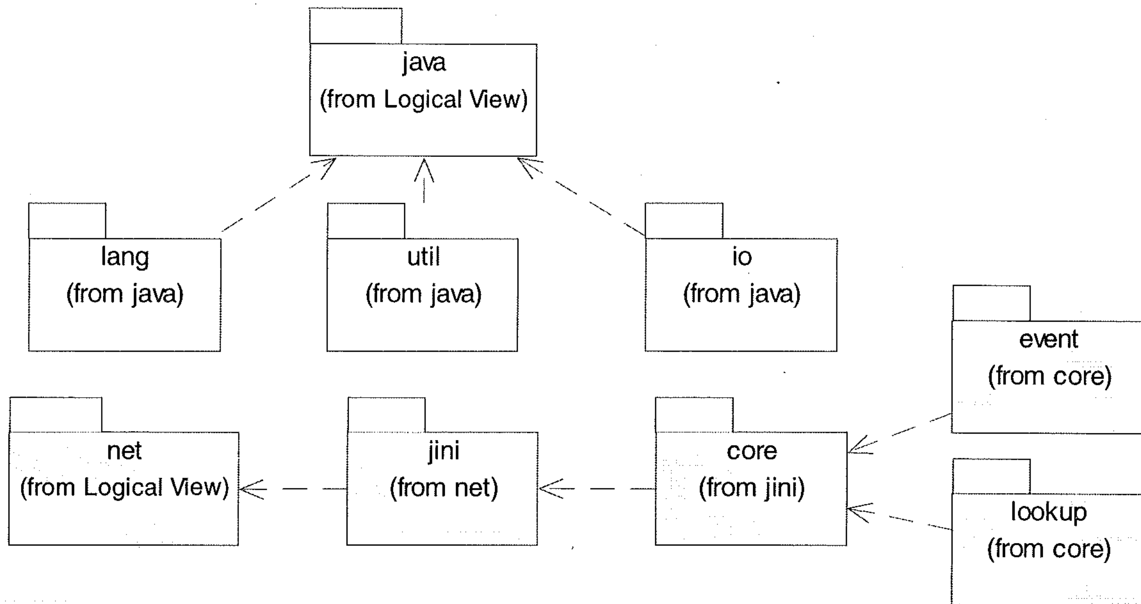
net.jini.core.discovery

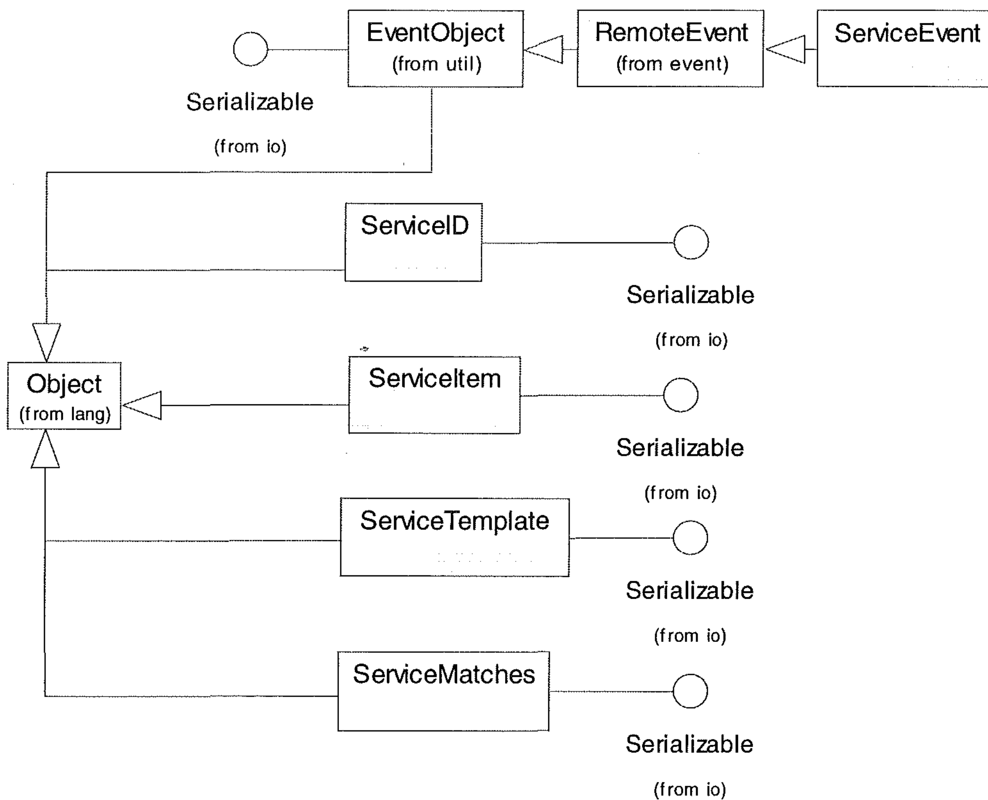


net.jini.core.entry

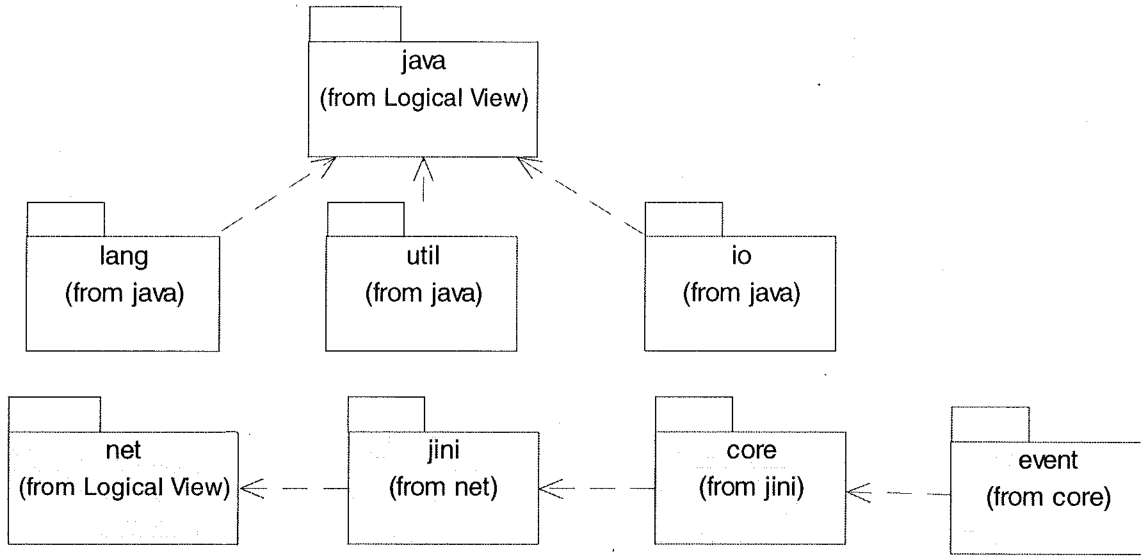


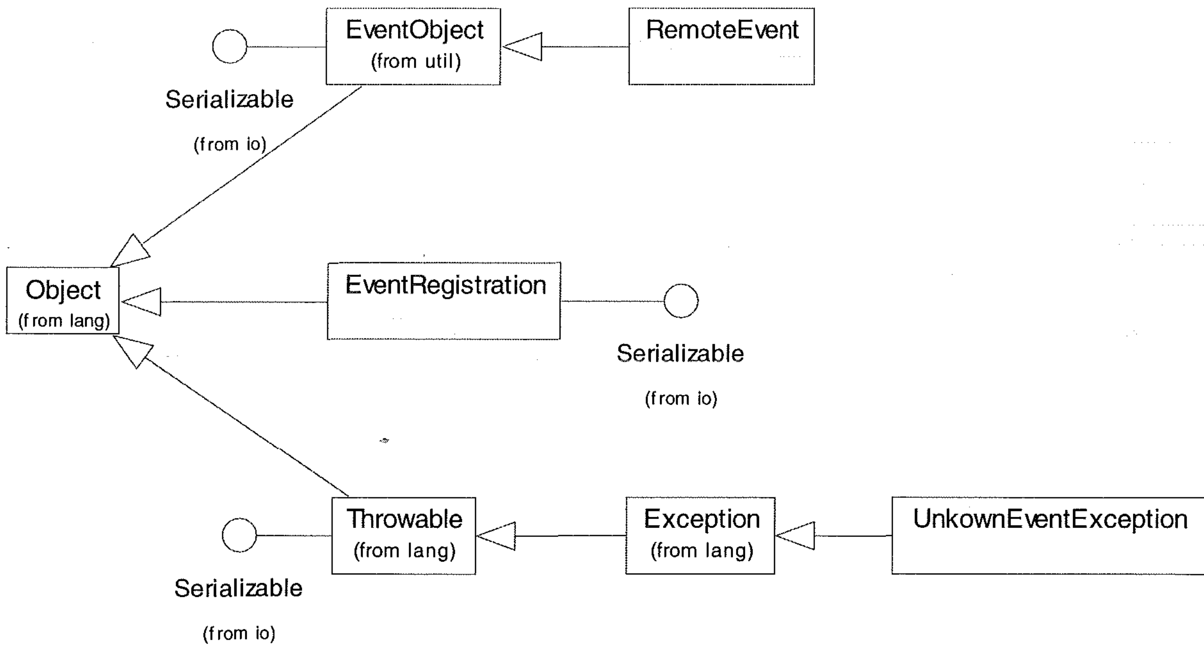
net.jini.core.lookup

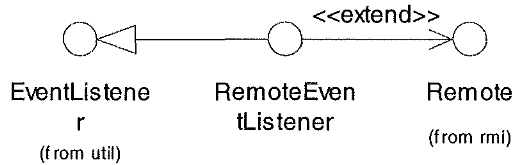


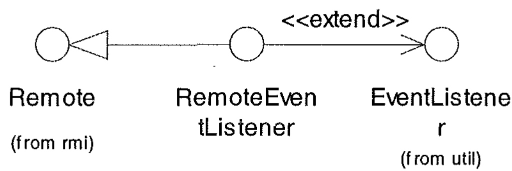


net.jini.core.event

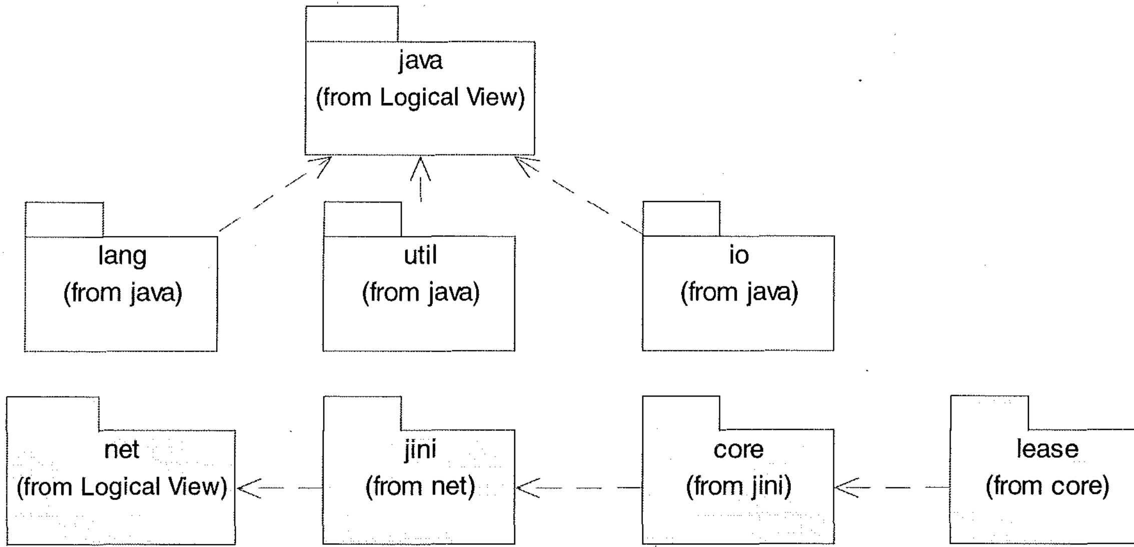


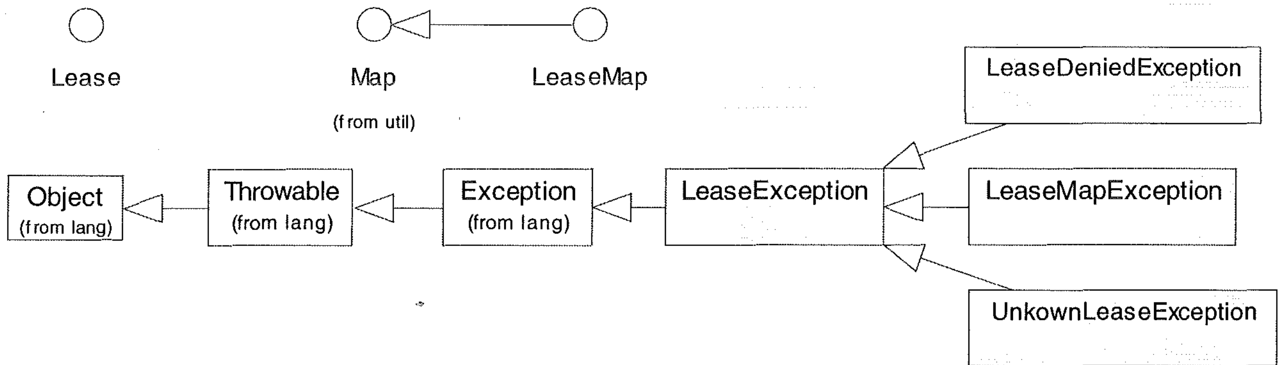




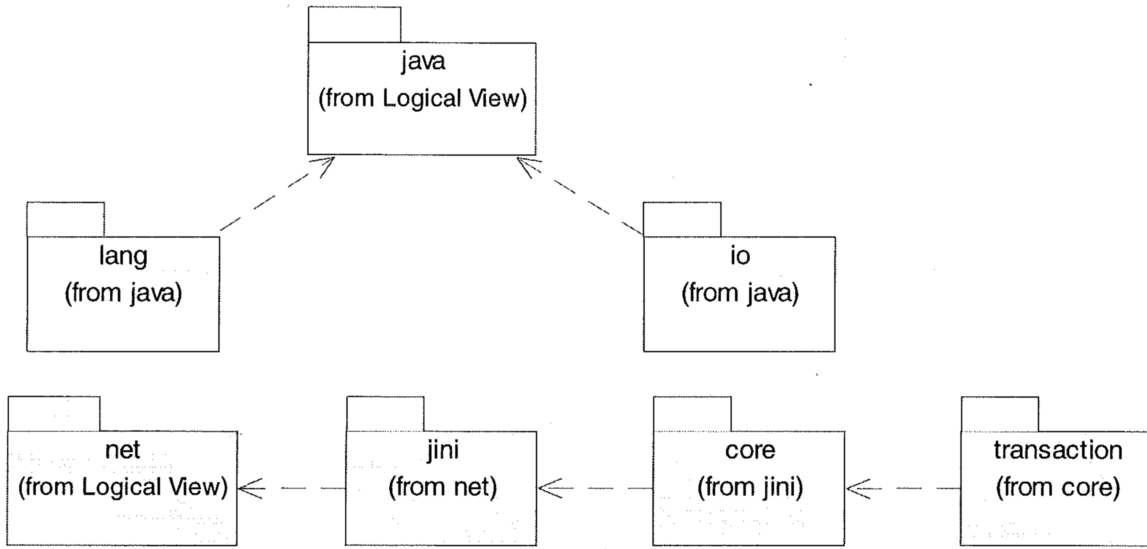


net.jini.core.lease





net.jini.core.transaction



TransactionFactory

CannotAbortException

CannotCommitException

Object
(from lang)

Throwable
(from lang)

Exception
(from lang)

TransactionException

CannotNestException

UnknownTranscationException

Serializable
(from io)

Created
(from Transaction)

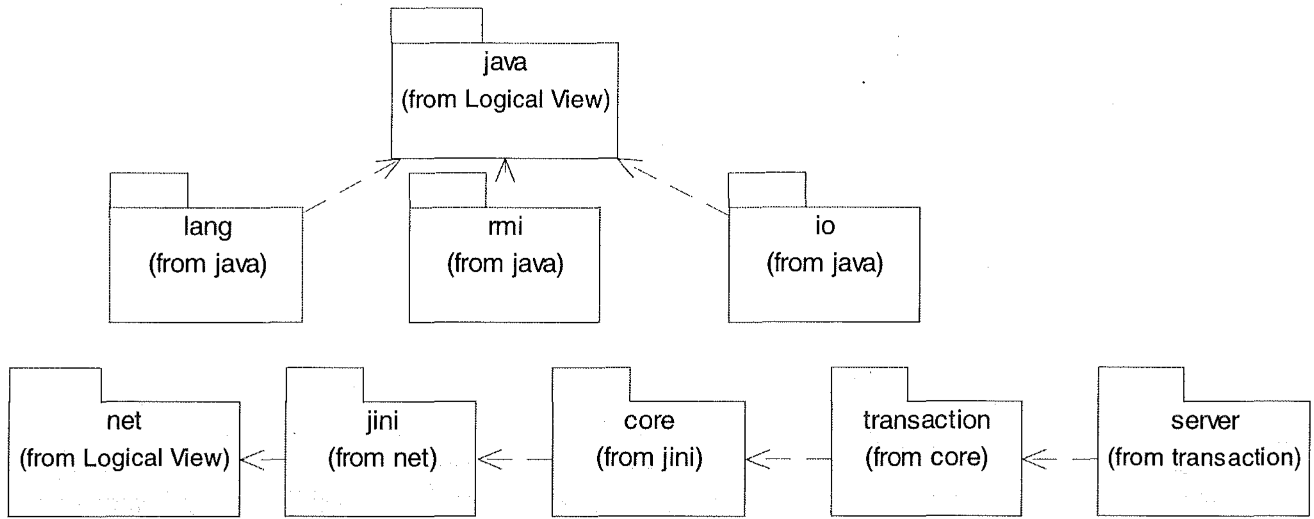
Transaction

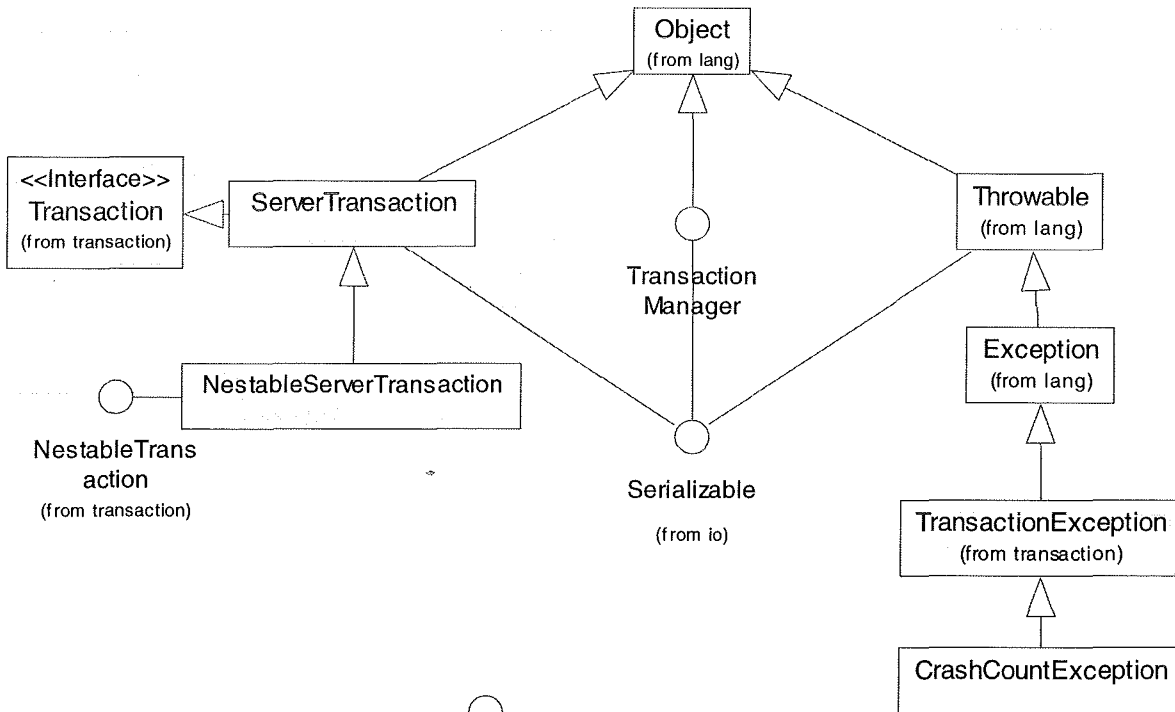
Created
(from NestableTransaction)

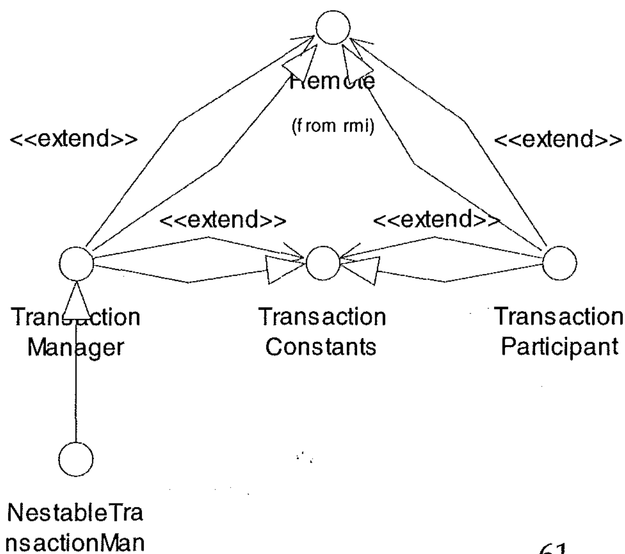
NestableTra
nsaction

Serializable
(from io)

net.jini.core.transaction.server







LIST OF REFERENCES

Arnold, Ken, O'Sullivan, Bryan, Scheifler, Robert W., Waldo, Jim and Wollrath, Ann & Friendly, Lisa. (1999). The Jini™ Specification: *The Jini™ Technology Series*. Palo Alto, CA: Addison-Wesley, Inc.

Brody, Steven. (1999). Anything to Everything: *Jini battles Microsoft and others for instant networking market*. Available at WWW: http://www.sunworld.com/swol-03-1999/swol-03-jini.html. SunWorld.

Charles, John, "Ubiquitous Computing Uncorked," *IEEE Software*, vol. 16, pp. 97-99, 1999.

Christensson, Bengt. (1999). WinHEC 99 White Paper: *Universal Plug and Play Connects Smart Devices*. Available at World Wide Web: http://www.axis.com/products/documentation/UPnP.doc. Axis Communications, Inc.

Clark, David, "Service with a (smart) smile: networks Jini-style," *IEEE Intelligent Systems*, vol. 14, pp. 81-83, 1999.

CNET. (2000). CNET Glossary: *Plug and Play*. Available at WWW: http://coverage.cnet.com/Resources/Info/Glossary/Terms/plugandplay.html. CNET, Inc.

Cohen, Josh, Aggarwal, Sonu and Goland, Yaron Y. (1999). General Event Notification Architecture Base: *Client to Arbiter*. Available at WWW: http://www.ietf.org/internet-drafts/draft-cohen-gena-client-00.txt. Microsoft Corp.

Cole, Bernard. (1999). Microsoft pushes Universal Plug and Play at WinHEC. Available at WWW: http://24.1.208.115/1394informer/990414B.htm. 1394 Informer.

Crichton, Charles, Davies, Jim and Woodcocky, Jim. (1999). When to trust mobile objects: access control in the Jini™ Software System: ***Proceedings of the Technology of Object-Oriented Languages and Systems***. Santa Barbara, California: IEEE.

Droms, Ralph. (1997). Dynamic Host Configuration Protocol. Available at WWW: http://www.dhcp.org/rfc2131.html. Network Working Group.

Edwards, W. Keith. (1999). Core Jini. Upper Saddle River, NJ: Prentice Hall PTR.

Flowwworks. (1999). EVOLVE: *Version Object Manager*. Available at WWW: http://flowwworks.com/evolve/. Flowwworks Workflow Systems.

Gage, Deborah. (1999). HP Takes Microsoft Closer to Jini. Available at WWW: http://www.zdnet.com/sr/stories/infopack/0,,2239858,00.html. ZD, Inc.

IEEE, "Sun's JavaSpaces and IBM's TSpaces," *IEEE Internet Computing*, vol. 2, 1998.

Intel. (2000). Intel Architecture Labs: *Plug and Play*. Available at WWW: http://developer.intel.com/ial/plugplay/index.htm. Intel Corp.

John, Rekesh. (1999). UPnP, Jini and Salutation - A look at some popular coordination frameworks for future networked devices. Available at WWW: http://www.cswl.com/whiteppr/tech/upnp.html. California Software Labs, Inc.

Lee, Yann-Hang. (1998). CDA 4102: Computer Architecture: *Plug and Play*. Available at WWW: http://www.hsi-lab.cise.ufl.edu/pc_arch/misc/pnp.html. CISE Dept., University of Florida.

Microsoft. (1999a). Simple Service Discovery Protocol/1.0: *Operating without an Arbiter*. Available at WWW: http://www.ietf.org/internet-drafts/draft-cai-ssdp-v1-03.txt. Microsoft, Corp.

Microsoft. (1999b). Universal Plug and Play: *Background*. Available at Word Wide Web: http://www.upnp.com/resources/UpnPbkgnd.htm. Microsoft, Corp.

Microsoft. (1999c). Universal Plug and Play Device Architecture Reference Specification: *Version 0.90*. Available at WWW: http://www.microsoft.com/hwdev/UPnP/. Microsoft, Corp.

Microsoft. (1999d). Universal Plug and Play to Make Network Configuration Easy and Convenient for Average Consumer. Available at WWW: http://research.microsoft.com/features/Uniplugplay.htm. Microsoft, Corp.

Middleton, Guy. (1999). Sun's Jini Appears To Be At Your Service. Available at World Wide Web: http://networkweek.com/applications/story/NWW19990201S0008. CMP Net.

MirrorWorlds. (2000). Learn about LifeStore: *Lifestream*™ + *Jini*™ = *LifeStore*™. Available at WWW: http://www.mirrorworlds.com/javaone/nist.html. Mirror Worlds Technologies, Inc.

Plummer, Daryl. (1999a). Internet & Intranet Development: Everything AND the Kitchen Sink: *GartnerGroup Symposium/ITxpo 99*. Brisbane, Australia: GartnerGroup.

Plummer, Daryl. (1999b). The Java Scenario: Reality Sets In: *GartnerGroup Symposium/ITxpo 99*. Brisbane, Australia: GartnerGroup.

Plummer, Daryl. (1999c). Web Projects in the Java Age: *GartnerGroup Symposium/ITxpo 99*. Brisbane, Australia: GartnerGroup.

Roberts, Simon and Byous, Jon. (1999). Distributed Events in Jini™ Technology. Available at WWW: http://developer.java.sun.com/developer/technicalArticles/ConsumerProducts/JiniEvents/index.html. Sun Microsystems, Inc.

SCA. (1997). Linda Overview: *The Linda Model*. Available at WWW: http://www.sca.com/LINDA_overview.html. Scientific Computing Associates, Inc.

Shnier, Mitchell. (1996). Dictionary of PC Hardware and Data Communications Terms: *Plug and Play*. Available at WWW: http://www.ora.com/reference/dictionary/terms/P/Plug_and_Play.htm. O'Reilly & Associates, Inc.

Sun. (1999a). Frequently Asked Questions: *RMI and Object Serialization*. Available at

WWW: http://java.sun.com/products/jdk/1.2/docs/guide/rmi/faq.html. Sun Microsystems, Inc.

Sun. (1999b). JavaTM 2 Name: *A note on the "JavaTM 2" Name*. Available at World Wide Web: http://www.java.sun.com/products/jdk/1.2/java2.html. Sun Microsystems, Inc.

Sun. (1999c). JiniTM Architecture Specification. Available at World Wide Web: http://www.sun.com/jini/specs/jini-spec.pdf. Sun Microsystems, Inc.

Sun. (1999d). JiniTM Connection Technology. Available at World Wide Web: http://www.sun.com/jini/. Sun Microsystems, Inc.

Sun. (1999e). JiniTM Connection Technology: *Executive Overview Revision 1.0*. Available at World Wide Web: http://www.sun.com/jini/overview/#998999. Sun Microsystems, Inc.

Sun. (1999f). JiniTM Discovery and Join Specification 1.0. Available at World Wide Web: http://www.sun.com/jini/specs/boot.pdf. Sun Microsystems, Inc.

Sun. (1999g). JiniTM Lookup Service Specification 1.0. Available at World Wide Web: http://www.sun.com/jini/specs/lookup.pdf. Sun Microsystems, Inc.

Sun. (1999h). JiniTM Technology Overview: *FAQs*. Available at WWW: http://www.sun.com/jini/faqs/jini_faqv4.pdf. Sun Microsystems, Inc.

Sun. (1999i). Leasing and Automatic Cleanup: JiniTM Technology "Under the Hood". Available at WWW: http://java.sun.com:8081/features/1999/01/cleanup.html. Sun Microsystems, Inc.

Venners, Bill. (1999). Frequently Asked Questions for Jini-Users Mailing List. Available at World Wide Web: http://www.artima.com/jini/faq.htm. Bill Venners.

Wobus, John. (1998). DHCP FAQ. Available at WWW: http://www.dhcp-handbook.com/dhcp_faq.html.