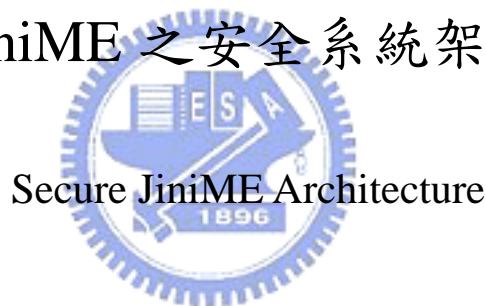


國立交通大學

資訊科學系

碩士論文

JiniME 之安全系統架構



研究生：蕭榮名 (Chi-Ming Hsiao)


指導教授：何慎諾教授 (Luc Claesen)

莊仁輝教授 (Jen-Hui Chuang)

中華民國 九十四 年 六月

Chapter 1 Introduction

More and more, information appliances can now interact with each other anytime and anywhere in the world as computing becomes more sophisticated and powerful. At the same time, decreasing processor costs and size let engineers endow ever more devices with application-specific processing power. In addition, mobile devices such as PDA, Tablet PC and smart phone not only will provide wireless inter-connection mechanism which easily coupled with all services client required, but also exchange information with high speed. These trends are moving toward the vision of pervasive computing. With all conditions of ubiquitous intelligent devices getting more and more mature, it seems to announce the advent of the new generation of intelligent digital network.



An interoperable networked environment provides multiple services for all members. People can easily use services they needed via network appliances and facilely add services to them within this connected community. For instance, people can acquire weather forecast, read news from information kiosk; the household of digital home can turn off light, or control room temperature through home center with their voiceprint. In order to reach this goal, we need a highly *reliable, scalable, simple* and *secure* architecture which is an end-to-end solution that forms a network of devices on the fly for delivering managed services on networked devices.

There are several related technologies, *Jini, Universal Plug and Play (UPnP), Home Audio Video Interoperability (HAVI), Bluetooth* and so on. But here we focus on Jini [1] for the sake of the fact that it meets the most of our requirements:

- *Simplicity*— Jini connection technology provides a user the ability to access

resources located anywhere on the network. Both user and resource locations can change without affecting the application. Users, devices, and resources can join and leave the network without manual reconfiguration.

- *Scalability*—Jini addresses the scalability problem through federation. The groups of services are called communities or federations, which can also be linked together into larger groups.
- *Reliability*—Communities of Jini services are largely self-healing. This is a key property built into Jini from the ground up: Jini doesn't make the assumption that networks are perfect, or that software never fails. Given enough time, damage to the system will be repaired automatically.
- *Security*—Jini runs on top of machines with Java Virtual Machine (JVM). It is based entirely on Java to function. The JVM protects the client machine from viruses that could come with downloaded code. Downloaded code is restricted to operations that the virtual machine's security allows.

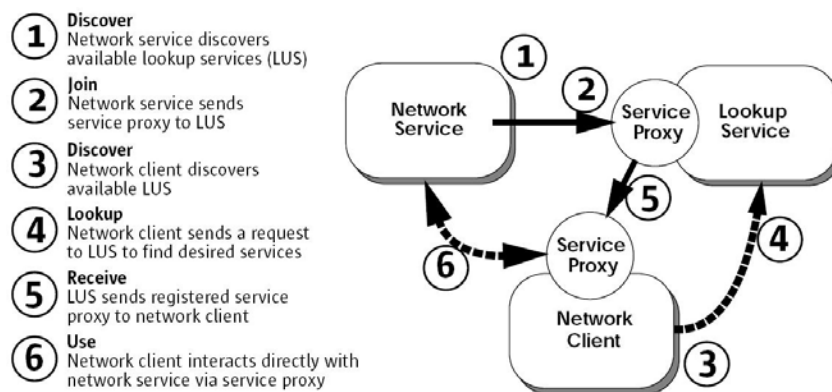


Figure 1-1 The way Jini technology works

Jini network technology is a middleware that provides a set of application programming interfaces as well as network protocols. As illustrated in Figure 1-1, the *Lookup Service* (LUS) is where services advertise their availability so that you can find

them. When a service is booted on the network, it uses *Discovery protocol* to find the local lookup services, and then registers its *proxy* object with each lookup service. While a client requests a service on network, it will look for local lookup services and automatically download the code for proxy objects if necessary. Each device publishes its own interfaces, which can be used by other devices to communicate with it and thereby access its particular service.

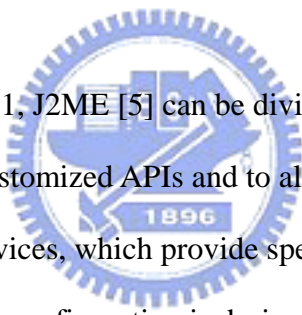
Jini leases a resource to a client for fixed amount of time to handle network failures. The *lease* automatically expires for all authorized users when a service goes down, the client must renew the lease to continue accessing that service after this period expires. Jini also supports redundancy in the infrastructure and resilience against failure. Servers register their proxy service objects with all the lookup services they can discover, and clients may obtain the reference of the desired service from any of those lookup services. This keeps services available, even if key machine crashes. In addition, *Jini Surrogate Architecture* — it allows non-Jini-capable devices to join the service federation— is a prominent method that lets limited-capability devices deliver their codes to an entity called a *surrogate host*. Generally speaking, such technology ensures compatible and standardized access among all devices.

The architecture proposed in this thesis is composed of three elements: *Jini Mobile Edition (JiniME)*, *Kerberos network authentication protocol* and *Java 2 Micro Edition, Connected Device Configuration (J2ME CDC)*. The goal is to build a secure ubiquitous architecture such that JiniME-capable devices can safely communicate with each other over network. In the rest of this thesis, we will describe related works in Chapter two, and then proposed architecture and implementation in Chapters three and four, respectively. Finally, some concluding remarks and future works are given in Chapter five.

Chapter 2 Related Works

2.1 J2ME

Java 2 Micro Edition (J2ME), one of three distinct Java 2 editions, is designed to address a wide range of consumer-oriented small devices. These devices can range from very limited mobile phones or pagers to high-end PDA or cable set-top boxes with resources approaching those of desktop computers. In this thesis, the JVM we want to adopt belongs to J2ME. The JVM is the cornerstone of the Java and Java 2 platforms and the component of Java technology responsible for its hardware- and operating system-independence, the small size of its compiled code, as well as its ability to protect users from malicious programs.



As illustrated in Figure 2-1, J2ME [5] can be divided into *configurations* and *profiles*, to prevent a proliferation of customized APIs and to allow Java to be flexible enough to run on a variety of platforms and devices, which provide specific information about APIs and different families of devices. A configuration is designed for a specific kind of device based on memory constraints and processor power. It usually specifies a JVM that can be easily ported to devices supporting the configuration. It also specifies some subset of the Java 2 Platform Standard Edition (J2SE) APIs that will be used on the platform, as well as additional APIs that may be necessary. Profiles are more specific than configurations. A profile is based on a configuration with addition of APIs for user interface, persistent storage, and whatever else is necessary for developing running applications.

MIDP (Mobile Information Device Profile)	PDAP (Personal Digital Assistant Profile)	Personal Profile	RMI Profile
CLDC (Connected, Limited Device Configuration)		Foundation Profile	
CLDC (Connected, Limited Device Configuration)		CDC (Connected Device Configuration)	
J2ME (Java 2, Micro Edition)			

Figure 2-1 The J2ME universe

2.1.1 CLDC and KVM

Connected, Limited Device Configuration (CLDC) is the configuration that it encompasses mobile phones, pagers, PDAs and other devices of similar size. The goal of CLDC is to standardize a highly portable, minimum-footprint Java application development platform for resource-constrained, connected devices. More specifically, the CLDC Specification defines a Java application development platform with the following characteristics and goals: (1) keep footprint small, (2) focus on application programming rather than systems programming, (3) enable dynamic downloading of applications and encourage third-party application development.

The *Kilobytes Virtual Machine* (KVM) is a specific implementation of the class of Virtual Machine (VM) defined by the CLDC specification. Although KVM is a complete virtual machine, in order to keep the size of the KVM small, certain features of the Java language specification and the JVM specification were omitted (as defined in the CLDC specification). This decision was made due to size and security concerns, and because CLDC does not include the J2SE security model. KVM offers a complete implementation of the Java language specification and limited error-handling capabilities, except for the following: (1) finalization of class instance, (2) support for floating-point data, (3) reflection support, (4) Java Native Interface (JNI) support, (5) User-defined Java-level class

loaders, (6) support for thread groups or daemon threads, and (7) support for weak references.

2.1.1.1 MIDP

The *Mobile Information Device Profile* (MIDP) is designed to work on top of the CLDC and thus supplements the CLDC API. The MIDP API provides Java's extension for programming mobile devices. The MIDP specification is tailor-made for high-end phones and other mobile devices, and it has a unique user interface API. An MIDP application is called a *MIDlet*. It's usually hosted inside an application management environment of a mobile device. In that respect, it is more similar to a Java applet, which works within an application environment provide by the Web browser, than to a J2SE application, which interacts directly with the native operating system via a virtual machine. The *Application Management Software* (AMS) maintains the state of the MIDlet and triggers state changes inside the MIDlet by invoking various methods that each MIDlet implements. Figure 2-2 shows the state diagram for a MIDlet.

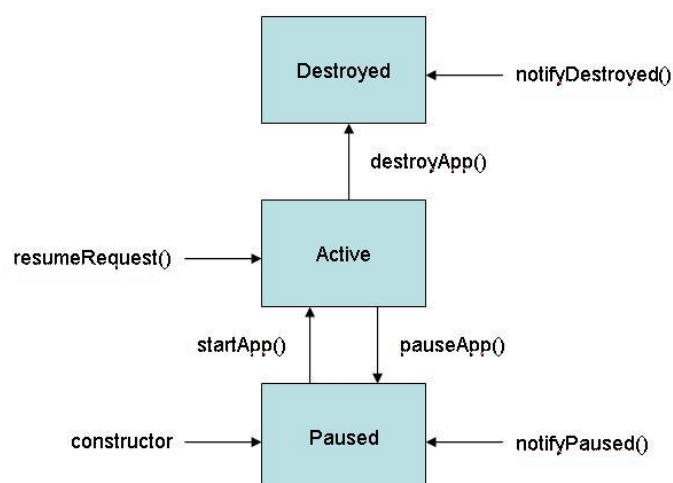


Figure 2-2 The MIDlet lifecycle

2.1.2 CDC and CVM

The *Connected Device Configuration* (CDC) is a configuration for using Java technology to build and deliver applications that can be shared across a range of network-connected consumer and embedded devices, including smart communicators, high-end personal digital assistants, and set-top boxes. It was designed around the two goals of J2SE compatibility and support for resource-constrained devices. J2SE compatibility allows developers to leverage their investments in J2SE technology, including libraries, tools and skills. Support for resource constrained devices allows device vendors to offer a feature-rich Java application environment that can support mobile enterprise applications with security.

The *C Virtual Machine* (CVM) is a specific implementation of the class of VM defined by the CDC specification. It was designed to be portable, space-efficient, and real time operation system aware. It also provides support for direct mapping of Java threads to native threads and for running Java classes from ROM. The CVM has the following additional features:

1. Extensible, well-defined interfaces that allow the CVM to be highly portable, modular, and easily customized.
2. The capability to clearly shut down and start up in a single address space without help from a process model.
3. Support for the full Java 2, version 1.3 virtual machine specification and libraries, which include the following:
 - i. Java 2 security APIs
 - ii. Weak references

- iii. Reflection
- iv. Serialization
- v. Java Native Interface (JNI)
- vi. Remote Method Invocation (RMI)
- vii. Java Virtual Machine Debugging Interface (JVMDI)
- viii. Java Debugger Architecture

2.1.2.1 Foundation Profile

The *Foundation Profile* is a J2ME CDC profile, which is intended to be used by devices requiring a complete implementation of the JVM up to and including the entire J2SE API. It may lack the specialized classes present in the MIDP, such as a graphic user interface (GUI) and persistence, but it does add many of the standard API features present in J2SE. Together with CDC, it presents a complete environment for producing applications that are network-aware but do not have a user interface. This flexibility could be used to create Foundation Profile applications that are embedded in network devices. In addition to the underlying requirements for the CDC, the Foundation Profile requires that the underlying device support the following:

1. Greater than 1M ROM and greater than 512K RAM for the Java environment and applications
2. Network connectivity, although the protocol is not specified
3. No GUI, unless the Foundation Profile is supplemented with another profile with a graphics library

2.1.2.2 RMI Optional Profile

The *Remote Method Invocation (RMI) Optional Profile* is also built on top of the Foundation Profile. It is different from the other CDC-based profiles because it is a supplemental profile. As the name implies, this profile is designed for smallish devices that will support RMI, and by extension, Jini. These capabilities enable a program to work with remote objects completely abstracted from the underlying networking layer. The remote objects look and work normally, with local objects, greatly simplifying the process of distributing and working with remote applications. This functionality is particularly useful when designing distributed applications across many host computers. Devices running the RMI should include the following specifications: minimum 2.5MB ROM, minimum 1MB RAM, connection to a TCP/IP-based network and a complete CDC and Foundation Profile environment



The API for the RMI Optional Package is a subset of the J2SE 1.3 RMI API that may be used with J2SE. This package conforms to the J2SE 1.3 RMI specification. Classes from the J2SE 1.3 RMI API are completely supported, or modified in ways allowed by the J2ME platform specification. Implementations of the RMI Optional Package must support the following “client-oriented” interfaces and functionality specified by the J2SE 1.3 RMI API:

1. Full RMI call semantics
2. Marshalled object support
3. RMI wire protocol
4. Export of remote objects through the `java.rmi.server.UnicastRemoteObject` API
5. Client and server side distributed garbage collection and garbage collector interfaces

6. The activator interface and the client side activation protocol
7. Registry interfaces and export of a Registry remote object
8. All system properties defined by the J2SE 1.3 RMI specification except those noted in the section below

The following interfaces and functionality defined by the J2SE 1.3 RMI specification and API are not part of the specification for this package and cannot be added to a conforming implementation:

1. RMI through firewalls via HTTP proxies
2. RMI's multiplexing protocol
3. Implementation model for an "Activatable" remote object
 - i. Interfaces:
`java.rmi.activation.ActivationInstantiator`,
`java.rmi.activation.ActivationMonitor`,
`java.rmi.activation.ActivationSystem`
 - ii. Classes:
`java.rmi.activation.Activatable`,
`java.rmi.activation.ActivationDesc`,
`java.rmi.activation.ActivationGroup`,
`java.rmi.activation.ActivationGroupDesc`,
`java.rmi.activation.ActivationGroupID`,
`java.rmi.activation.UnknownGroupException`
4. Deprecated methods, classes, and interfaces
 - i. Interfaces: `java.rmi.server.ServerRef`
 - ii. Classes:

java.rmi.registry.RegistryHandler,
java.rmi.RMISecurityException ,
java.rmi.server.LoaderHandler,
java.rmi.server.LogStream,
java.rmi.server.Skeleton,
java.rmi.server.SkeletonMismatchException,
java.rmi.server.SkeletonNotFoundException,
java.rmi.ServerRuntimeException

iii. **Methods:**

java.rmi.dgc.VMID.isUnique(),
java.rmi.server.Operation.getOperation(),
java.rmi.server.RemoteCall.getInputStream(),
java.rmi.server.RemoteCall.getOuputStream,
java.rmi.server.RemoteCall.releaseOuputStream(),
java.rmi.server.RemoteCall.releaseInputStream(),
java.rmi.server.RemoteCall.getResultStream (boolean),
java.rmi.server.RemoteCall.executeCall(),
java.rmi.server.RemoteCall.done(),
java.rmi.server.RemoteRef.done (RemoteCall),
java.rmi.server.RemoteRef.invoke (RemoteCall),
java.rmi.server.RemoteRef.newCall(RemoteObject, Operation[], int, long),
java.rmi.server.RemoteStub.setRef(RemoteStub, RemoteRef),
java.rmi.server.RMIClassLoader.getSecurityContext(ClassLoader),
java.rmi.server.RMIClassLoader.loadClass(String),
java.rmi.server.SocketSecurityException

5. Stub compiler, skeleton compiler and JDK 1.1 stub/skeleton protocol

2.2 JiniME

2.2.1 JiniME Architecture

Jini Mobile Edition (JiniME) [4] is a version of Jini Connection Technology designed to run on wireless mobile computing devices. It is targeted for devices having the J2ME CLDC MIDP. A JiniME federation consists of a number of JiniME-capable mobile devices and a Jini Bridge (optionally) as illustrated in Figure 2-3. It is desired to allow JiniME-capable mobile devices to use services in a standard Jini federation. Jini Bridge connected the two federations. On the one side, it connected to one Jini federation with wire network. On the other side, it connected to another JiniME federation with wireless mobile network. In later sections of this chapter, we will describe more detail about the architecture of the JiniME-capable mobile devices, the Jini Bridge, the Sessile-Client-to-Mobile-Service Bridge, and the Mobile-Client-to-Sessile-Service Bridge.

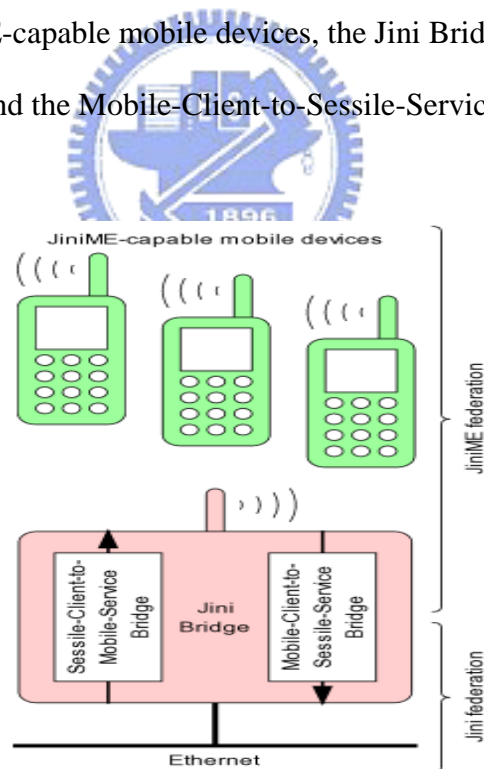


Figure 2-3 JiniME federation architecture

2.2.1.1 JiniME-Capable Mobile Devices

Each *JiniME-capable mobile device* as illustrated in Figure 2-4 includes the following elements:

1. Wireless network connection
2. Operating environment: each mobile device has a J2ME operating environment with Java class libraries
 - i. Connected, Limited Device Configuration: K Virtual Machine, Generic Connection Framework
 - ii. Mobile Information Device Profile: Interface `HttpConnection`
 - iii. JiniME Profile (proposed in [4]):
Interface `HttpServerConnection`, Interface `HttpServerConnectionNotifier`, Interface `SocketConnection`, Interface `SocketConnectionNotifier`, Class `Classpath`, Class `LookupDiscovery`
3. HTTP server
4. Lookup Service
5. Services implemented locally
6. Services implemented remotely
7. Client applications

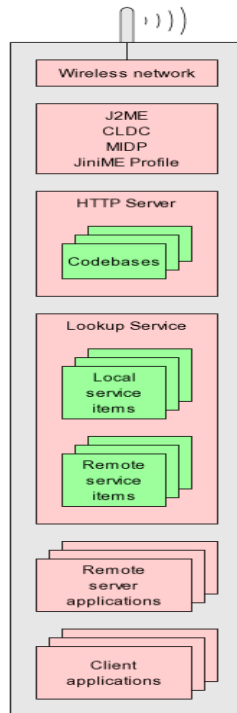


Figure 2-4 JiniME-capable mobile device architecture

2.2.1.2 Jini Bridge



The *Jini Bridge* as illustrated in Figure 2-5 includes the following elements:

1. Wireless network connection
2. Wired network connection
3. Operating environment
 - i. Full J2SE operating environment,
 - ii. J2ME implementation of the CLDC Generic Connection Framework: HTTP and socket connections as defined in the MIDP and the JiniME Profile
4. HTTP Server
5. Sessile-Client-to-Mobile-Service Bridge
6. Mobile-Client-to-Sessile-Service Bridge

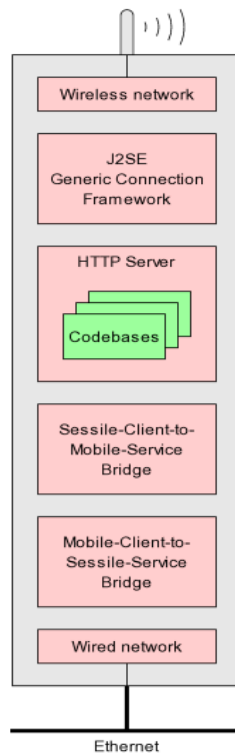


Figure 2-5 Jini Bridge architecture

2.2.1.3 Sessile-Client-to-Mobile-Service Bridge

The *Sessile-Client-to-Mobile-Service Bridge* as illustrated in Figures 2-6 and Figure 2-7 includes the following elements:

- **Mobile Service Bridge Builder application.** It does the following:
 1. The builder application periodically discovers JiniME Lookup Services (devices) on the wireless side.
 2. From each JiniME Lookup Service, the builder application obtains all the JiniME service items. Each of these is a *Mobile Service* (M.S.).
 3. For each Mobile Service, the builder application uses reflection to construct a standard Jini service, called the *Mobile Service Bridge* (M.S.B.), that has the

same interfaces as the Mobile Service and that is implemented to delegate all method invocations to the Mobile Service's proxy object.

4. The builder application runs each newly-constructed Mobile Service Bridge.
5. When the builder application detects that a wireless device has disappeared, the builder application shuts down the corresponding Mobile Service Bridges.

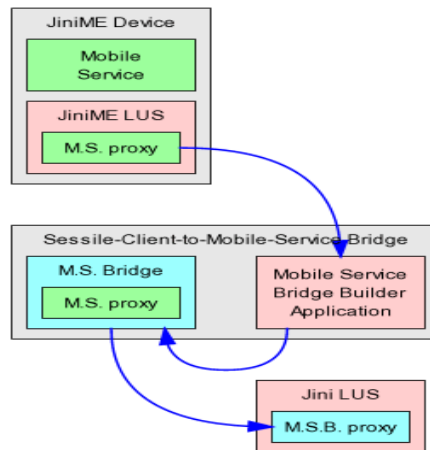


Figure 2-6 Constructing a Bridge for a Mobile Service and registering it with the Jini federation

- **Mobile Service Bridges.** Each Mobile Service Bridge which the bridge builder constructs does the following:

1. Registers itself with the Jini Lookup Services in the Jini federation. The Jini service item consists of:
 - i. The same service ID as the Mobile Service.
 - ii. The same service attributes as the Mobile Service.
 - iii. A service proxy object which is an RMI stub for the Mobile Service Bridge.
2. Exports its service proxy object's codebase via the Jini Bridge's HTTP server.
3. Maintains the leases on its service registrations with the Jini Lookup Services.

4. When an RMI call arrives from the Mobile Service Bridge proxy object, delegates the call to the Mobile Service proxy object.
5. When shut down, cancels its service registration leases with the Jini Lookup Services.

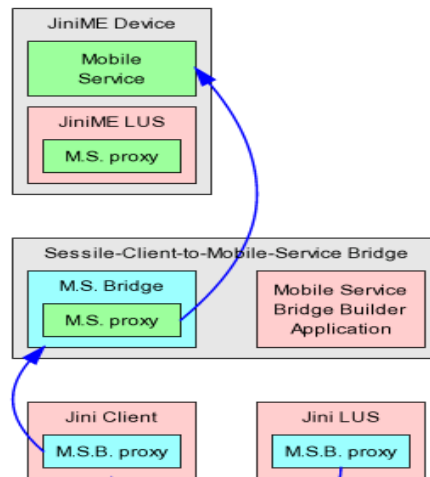


Figure 2-7 A Jini clients discovering a Mobile Service and invoking it via the Mobile Service Bridge

2.2.1.4 Mobile-Client-to-Sessile-Service Bridge

As illustrated in Figures 2-8 and 2-9, the Mobile-Client-to-Sessile-Service Bridge, which includes following, is exactly the reverse of the Sessile-Client-to-Mobile-Service Bridge.

- **Sessile Service Bridge Builder application.** It does the following:
 1. The builder application discovers all the Jini Lookup Services on the wired side using the Jini discovery protocols and registers to receive notifications of added, changed, and removed services.
 2. From each Jini Lookup Service, the builder application obtains all the Jini service items. Each of these is a *Sessile Service (S.S.)*.

3. For each Sessile Service, the builder application uses reflection to construct a JiniME service, called the *Sessile Service Bridge* (S.S.B.), that has the same interfaces as the Sessile Service and that is implemented to delegate all method invocations to the Sessile Service's proxy object.
4. The builder application runs each newly-constructed Sessile Service Bridge.
5. When the builder application detects that a Jini service has disappeared, the builder application shuts down the corresponding Sessile Service Bridge.

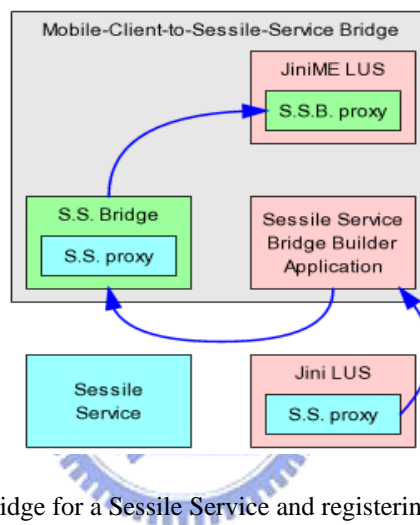


Figure 2-8 Constructing a Bridge for a Sessile Service and registering it with the JiniME federation

- **Sessile Service Bridges.** Each Sessile Service Bridge which the bridge builder constructs does the following:
 1. Adds its JiniME service item to the Jini Bridge's own JiniME Lookup Service in the JiniME federation. The JiniME service item consists of:
 - i. The same service ID as the Sessile Service.
 - ii. The same service attributes as the Sessile Service.
 - iii. A service proxy object which uses the previously-described techniques to let JiniME client applications invoke the Sessile Service Bridge.

2. Exports its service proxy object's codebase via the Jini Bridge's HTTP server.
3. When a message arrives from the Sessile Service Bridge proxy object, translates the message into a method call on the Sessile Service proxy object.
4. When shut down, removes its service item from the Jini Bridge's JiniME Lookup Service.

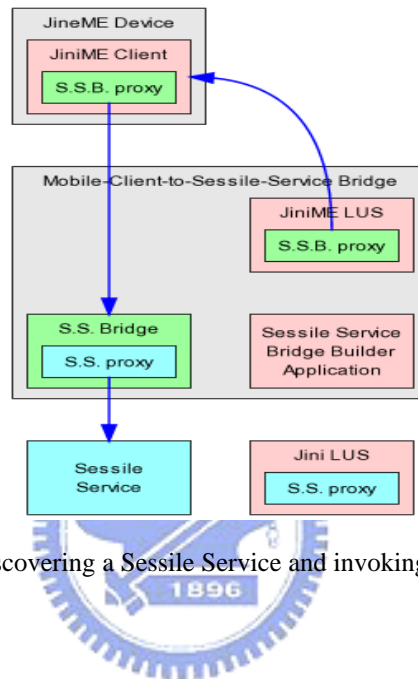


Figure 2-9 A JiniME clients discovering a Sessile Service and invoking it via the Sessile Service Bridge

2.3 Security

2.3.1 Jini Security

The security design model of Jini Connection Technology is built on two relative concepts: *Principal* and *Access Control List*. Principal is on behalf of some entity accessing the services of Jini federation. Server sometimes needs to access the services of other server, it depends on the security notion to fulfill in such situation; Therefore, the security functionality of the access control list must adhere to its rules without any violation behavior when client would like to access resource.

The original Java security technology and Jini security model cannot satisfy the

requirements of embedded devices in distributed environment. In attempt to support more secure protection mechanism, this thesis would adopt some mechanisms: secret key cryptosystems, public key cryptosystems, authorization and authentication.

2.3.2 Kerberos Authentication Protocol

Kerberos Version 5 (Kerberos V5) [6], a trusted third party network authentication protocol developed at MIT, is designed to provide strong authentication for client/server applications by using secret key cryptography, and does the authenticating and encrypting transparently. In addition, it is a *single-sign-on* system, which means that you have to type your password only once per session. As illustrated in Figure 2-10, there are four roles in Kerberos V5 authentication protocol: *Client*, *Server*, *Authentication Server (AS)*, and *Ticket Granting Server (TGS)*.

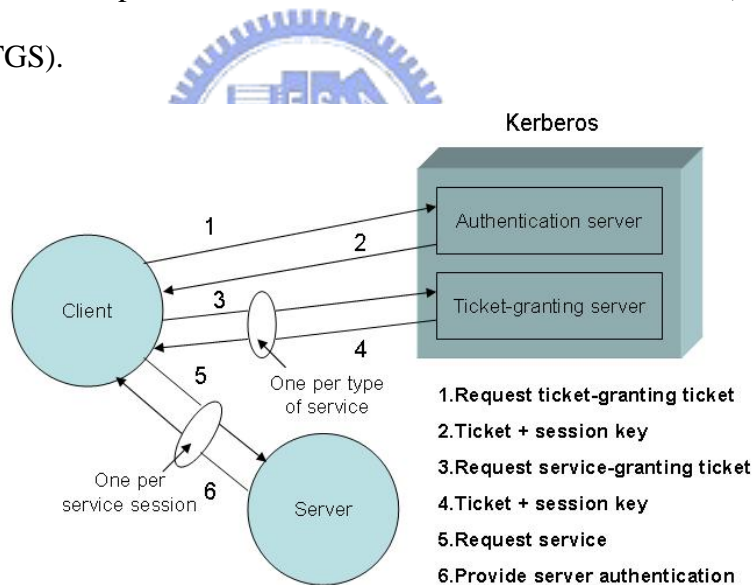


Figure 2-10 Kerberos authentication protocol

Under Kerberos, a client (either a user or a service) sends a request for a ticket to the *Key Distribution Center (KDC)* including AS and TGS. The KDC creates a TGT for the client, encrypts it using the client's password as the key, and sends the encrypted TGT back to the client. The client then decrypts the TGT using its password. If successfully decrypts

the TGT, it keeps the decrypted TGT, which indicates proof of the client's identity. The TGT, which will expire at a specified time, permits the client to obtain additional tickets, for permission to specific services. Before the TGT expires, client only communicates with TGS without via AS.

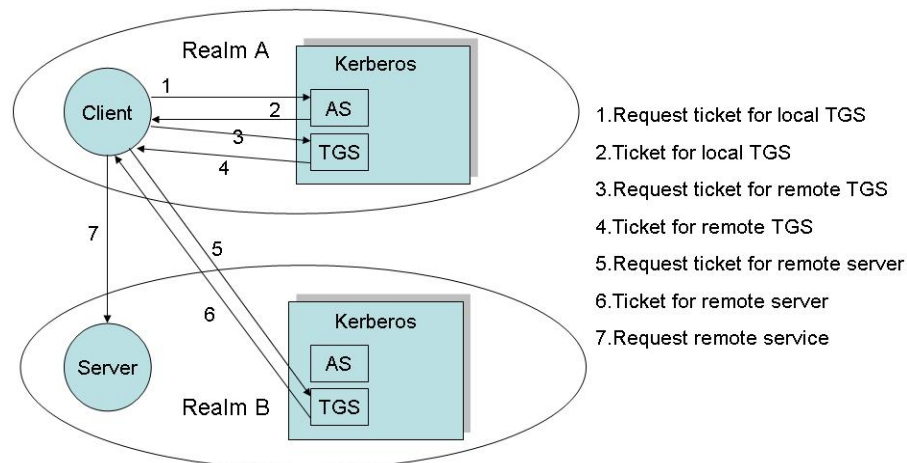


Figure 2-11 Interaction of different Kerberos administrative domain

On the other hand, Kerberos also supports cross domain authentication as illustrated in Figure 2-11. Each administrative domain will have its own Kerberos database, which contains information about the users and services for that particular site or administrative domain. This administrative domain is the *Kerberos realm*. Each Kerberos realm will have at least one Kerberos server, where the master Kerberos database for that site or administrative domain is stored. A Kerberos realm may also have one or more *slave servers*, which have read-only copies of the Kerberos database that are periodically propagated from the master server. Under such authentication mechanism, the requesting and granting of these additional tickets is user-transparent. Since Kerberos negotiates authenticated, and encrypted optionally, communications between two points anywhere on the internet, it provides a layer of security that is not dependent on which side of a firewall either client is on.

An application server is a host that provides one or more services over the network which can be “secure” or “insecure.” A “secure” host is set up to require authentication from every client connecting to it. If not, an “insecure” host will still provide Kerberos authentication, but will also allow unauthenticated clients to connect. We recommend that you make your hosts secure to take advantage of the security that Kerberos authentication affords. If your applications want to adopt Kerberos V5, either the operation system or applications must support it. In the former, your applications do not need to be rewritten. For example, UNIX series, Microsoft Windows series, Mac OS, and so on support Kerberos V5. In addition, Kerberos V5 has provided Kerberos-enhanced versions of server for UNIX network daemons such as ftpd, klogind, kshd, and telnetd. In the latter, you should rewrite all applications to support it, which may not be a good idea.

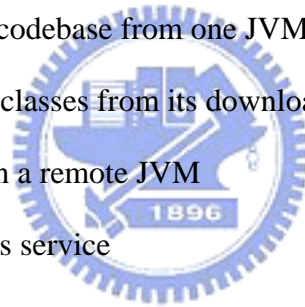


Chapter 3 Architecture

3.1 JiniME Methodology

The J2ME CLDC MIDP was designed to run on small mobile computing devices. The J2ME CLDC MIDP requires fewer computing resources than the J2SE or the J2ME CDC. It omits the classloaders, security, reflection, objects serialization, and marshaled objects on which Jini technology relies, thus a J2ME CLDC device is simply not adequate to support Jini Connection Technology. There are several technical challenges as the following and methods to fulfill the principles of JiniME discussing in its whitepaper:

1. Moving an object from one JVM to another
2. Moving an object's codebase from one JVM to another
3. Loading an object's classes from its downloaded codebase
4. Invoking a service in a remote JVM
5. Publishing a device's service
6. Discovering a device



3.2 Secure JiniME Methodology

Until now, we have already discussing about JiniME architecture, J2ME CLDC, J2ME CDC, MIDP Profile, Foundation Profile, RMI Optional Profile and Kerberos authentication protocol in related works. And, we will describe what JiniME implementation problem are and how to solve it in Section 3.2.1. Afterward, we will begin to describe the Secure JiniME Architecture in Section 3.2.2.

3.2.1 Why modify JiniME Methodology

In the first place, JiniME originally proposed a method, which is moveable interface, to solve moving an object depending on remote method invocation because J2ME CLDC MIDP lacks serialization and reflection abilities. That is a main issue with serialization: it depends on reflection. The dependence on reflection is the hardest of these to eliminate. Both serializing and de-serializing require the serialization mechanism to discover information about the instance it is serializing.

In attempt to solve the issue very hard to implement JiniME, the Secure JiniME architecture adopts J2ME CDC Foundation Profile and RMI Optional Profile (1.9MB) instead of J2ME CLDC MIDP (about 800KB). The advantages are (1) To fulfill easily due to the former supported serialization, (2) Support security packages more than CLDC, (3) Support a rich-networked J2ME environment, but memory sizes has increased which is under the tolerance embedded devices can endure. Secondary, we only retains 5th solution of which the JiniME has proposed. The first item we have explained before. Moreover, the other items are the same reason as first item as we adopt J2ME CDC Foundation Profile and RMI Optional Profile.

Lastly, JiniME is a version of Jini Mobile Edition, it is another reason we shift from CLDC into CDC, which exposed itself under unsafe circumstances without any security protection mechanism. JiniME don't have any standardized concepts for security as they apply to network communication. If I am a JiniME client, and I talk to you as JiniME service, there are standard notions of network security that should come into play. Due to the reason mentioned before, we incorporated security mechanism such as authentication protocol to JiniME.

3.2.2 Secure JiniME Architecture

A Secure JiniME Connection Technology federation consists of a number of Secure JiniME-capable mobile devices and a Jini Bridge (optionally) as illustrated in Figure 3-1. (You could compare Section 3.2.2 with Section 2.2) Each member of Secure JiniME federation ascribes to one Kerberos realm which we have discussed in Section 2.3.2, and all Jini federation could ascribe to one Kerberos realm.

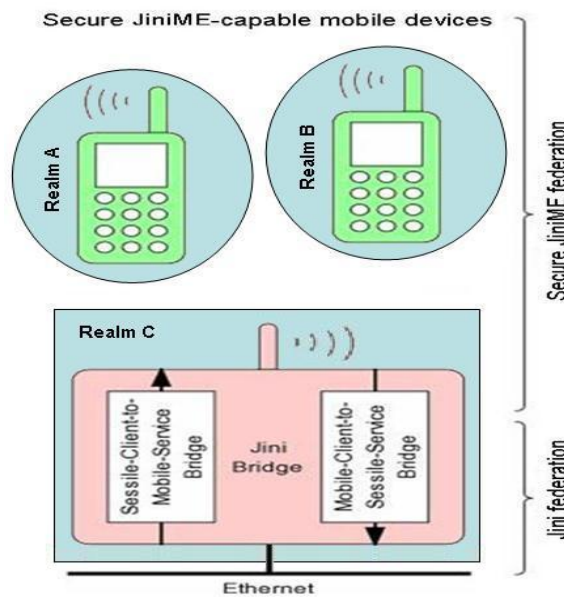


Figure 3-1 Secure JiniME federation architecture

3.2.2.1 Secure JiniME-Capable Mobile Devices

Each *Secure JiniME-capable mobile device* as illustrated in Figure 3-2 includes the following elements:

1. Wireless network connection
2. Operating environment: each mobile device has a J2ME operating environment with Java class libraries

- i. Connected, Device Configuration: C Virtual Machine
 - ii. Foundation Profile
 - iii. RMI Optional Profile
 - iv. Secure JiniME Package: JAAS and Jini packages
3. Kerberos authentication systems
 4. HTTPS/HTTPMD Server
 5. Lookup Service
 6. Services implemented locally and remotely, Client applications

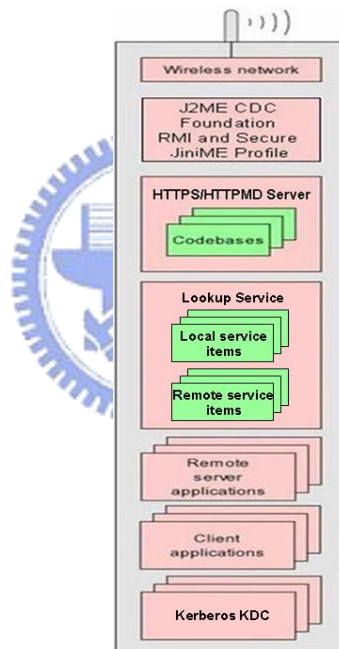


Figure 3-2 Secure JiniME-capable mobile device architecture

3.2.2.2 Jini Bridge

The *Jini Bridge* as illustrated in Figure 3-3 includes the following elements:

1. Wireless network connection
2. Wired network connection

3. Operating environment:
 - i. Full J2SE operating environment
4. Kerberos authentication systems (or on other host)
5. HTTPS/HTTPMD Server
6. Sessile-Client-to-Mobile-Service Bridge
7. Mobile-Client-to-Sessile-Service Bridge

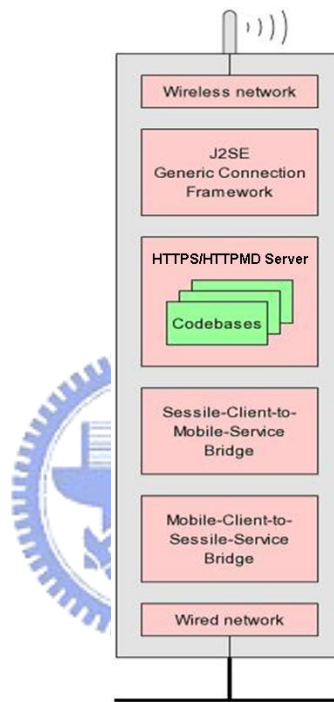


Figure 3-3 Jini Bridge architecture

3.2.2.3 Sessile-Client-to-Mobile-Service Bridge

The *Sessile-Client-to-Mobile-Service Bridge* as illustrated in Figures 3-4 and Figure 3-5 includes the following elements:

- **Mobile Service Bridge Builder application.** It does the following:
 1. The builder application logs in the KDC belongs to Realm B and periodically discovers Secure JiniME Lookup Services (devices) on the wireless side.

2. From each Secure JiniME Lookup Service, the builder application obtains all the Secure JiniME service items. Each of these is a *Mobile Service* (M.S.).
3. For each Mobile Service, the builder application uses reflection to construct a Secure JiniME service, called the *Mobile Service Bridge* (M.S.B.), that has the same interfaces as the Mobile Service and that is implemented to delegate all method invocations to the Mobile Service's proxy object.
4. The builder application runs each newly-constructed Mobile Service Bridge.
5. When the builder application detects that a wireless device has disappeared, the builder application shuts down the corresponding Mobile Service Bridges.

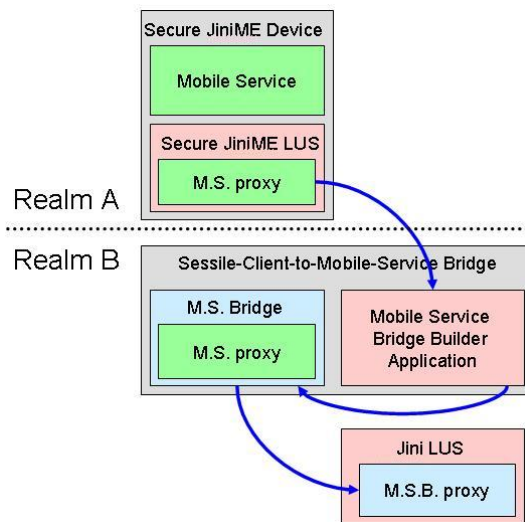


Figure 3-4 Constructing a Bridge for a Mobile Service and registering it with the Jini federation

- **Mobile Service Bridges.** Each Mobile Service Bridge which the bridge builder constructs does the following:
 1. Login the KDC belongs to Realm B and registers itself with the Jini Lookup Services in the Jini federation. The Secure JiniME service item consists of:
 - i. The same service ID as the Mobile Service.
 - ii. The same service attributes as the Mobile Service.

- iii. A service proxy object which is an RMI stub for the Mobile Service Bridge.
2. Exports its service proxy object's codebase via the Jini Bridge's HTTPS or HTTPMD server.
 3. Maintains the leases on its service registrations with the Jini Lookup Services.
 4. When an RMI call arrives from the Mobile Service Bridge proxy object, delegates the call to the Mobile Service proxy object.
 5. When shut down, cancels its service registration leases with the Jini Lookup Services.

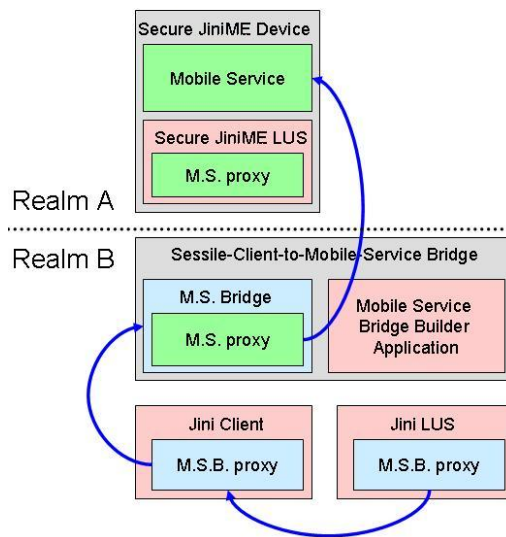


Figure 3-5 A Jini clients discovering a Mobile Service and invoking it via the Mobile Service Bridge

3.2.2.4 Mobile-Client-to-Sessile-Service Bridge

As illustrated in Figures 3-6 and 3-7, Mobile-Client-to-Sessile-Service Bridge, which includes following, is exactly the reverse of the Sessile-Client-to-Mobile-Service Bridge.

- **Sessile Service Bridge Builder application.** It does the following:
 1. The builder application logs in the KDC belongs to Realm B and discovers all the Jini LUS on the wired side using the Jini discovery protocols, then registers to receive notifications of added, changed, and removed services.
 2. From each Jini Lookup Service, the builder application obtains all the Jini service items. Each of these is a *Sessile Service* (S.S.).
 3. For each Sessile Service, the builder application uses reflection to construct a standard Jini service, called the *Sessile Service Bridge* (S.S.B.), that has the same interfaces as the Sessile Service and that is implemented to delegate all method invocations to the Sessile Service's proxy object.
 4. The builder application runs each newly-constructed Sessile Service Bridge.
 5. When the builder application detects that a Jini service has disappeared, the builder application shuts down the corresponding Sessile Service Bridge.

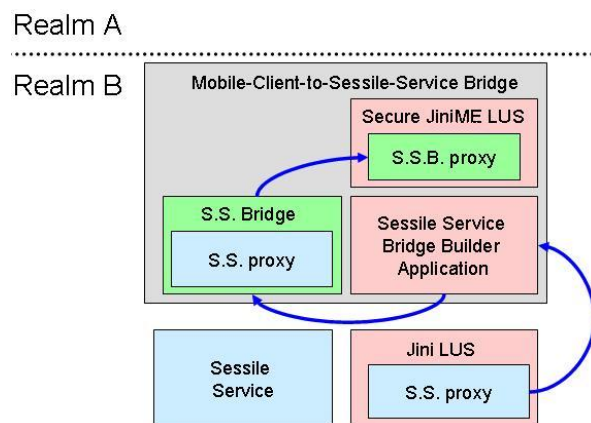


Figure 3-6 Constructing a Bridge for a Sessile Service and registering it with the JiniME federation

- **Sessile Service Bridges.** Each Sessile Service Bridge which the bridge builder constructs does the following:

1. Adds its Jini service item to the Jini Bridge's own Secure JiniME Lookup Service in the Secure JiniME federation. The Jini service item consists of:
 - i. The same service ID as the Sessile Service.
 - ii. The same service attributes as the Sessile Service.
 - iii. A service proxy object which uses the previously-described techniques to let Secure JiniME client applications invoke the Sessile Service Bridge.
2. Exports its service proxy object's codebase via the Jini Bridge's HTTPS or HTTPMD server.
3. When a message arrives from the Sessile Service Bridge proxy object, translates the message into a method call on the Sessile Service proxy object.
4. When shut down, removes its service item from the Jini Bridge's Secure JiniME Lookup Service.

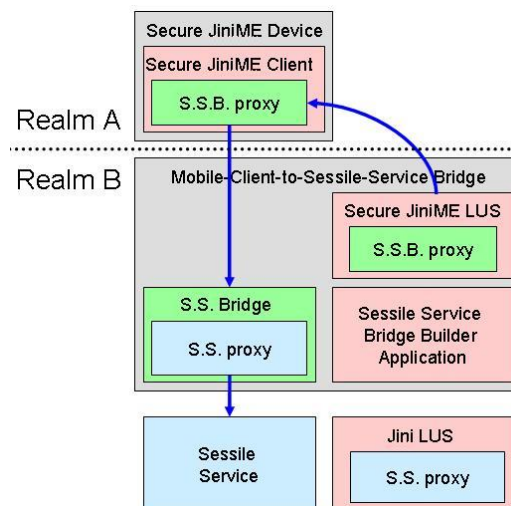


Figure 3-7 A JiniME clients discovering a Sessile Service and invoking it via the Sessile Service Bridge

Chapter 4 Implementation

In this chapter, the way an overview will be provided for the implementation of the proposed modified version of [2, 3]: (1) coding snips of Secure JiniME, (2) the way system is set up, and (3) the execution of program. For the first topic, it includes some code snips such as Client, Service, HTTPMD Server, and Jini Bridge that we will be described in Sections 4.1 - 4.4; for the second topic, we will describe how to set security properties and system properties and how to set JAAS configuration as well as the security policy of access control in Section 4.5; for the last topic, which will describe how to prepare to running program in Section 4.6.

4.1 Client Code Snip

As described in Section 3.2.2, Secure JiniME Architecture consists of two devices. Specifically, the role of Secure JiniME-capable device could be a client to request services, a server to offer services, or both. In this Section, we will discuss some implementation details about client code. The Secure JiniME Client uses Kerberos to authenticate and login with KDC as follows:

```
final Configuration config = ConfigurationProvider.getInstance(args);  
  
LoginContext loginContext = (LoginContext) config.getEntry(  
    "com.sun.securejinime.hello.Client", "loginContext", LoginContext.class, null);  
if (loginContext == null) {  
    System.out.println("No JAAS login is performed!");  
}  
else {  
    loginContext.login();  
    Subject.doAsPrivileged(loginContext.getSubject(),  
        new PrivilegedExceptionAction() {  
            public Object run() throws Exception {
```

```

        mainAsSubject(config);
        return null; }
    }, null);
}

```

In order to login KDC, client needs to instantiate *LoginContext* and *Configuration* objects. *LoginContext* uses *Configuration* to configure itself. The purpose of a login context is to allow retrieval of an authenticated user, which is represented as a subject object. The *Subject* class is used to represent an authenticated user. In fact, each user is represented as an array of *Principal* objects stored by this class. There is an array of objects because each user is likely to have several identifying characteristics.

After obtain a user login object, the program issues a method call (the *doAS()* or *doAsPrivileged()* method), passing in the user login object and the code that should be executed on behalf of that user. As in following code, “ServiceTemplate” describes service attributes and Client uses “serviceDiscovery” to lookup it. If succeeded, it will return the service’s proxy. Otherwise, it will throw *Exception*.

```

static void mainAsSubject(Configuration config) throws Exception {

    ServiceDiscoveryManager serviceDiscovery;

    try {
        serviceDiscovery = (ServiceDiscoveryManager) config.getEntry(
            "com.sun.securejinime.hello.Client",
            "serviceDiscovery",ServiceDiscoveryManager.class);
    } catch (NoSuchEntryException e) {
        /* Default to search in the public group */
        serviceDiscovery = new ServiceDiscoveryManager(
            new LookupDiscovery(new String[] { "" }, config),
            null, config);
    }
}

```

```

ServiceItem serviceItem = serviceDiscovery.lookup(

    new ServiceTemplate(null, new Class[] { Hello.class }, null),null,
    Long.MAX_VALUE);

Hello server = (Hello) serviceItem.service;

ProxyPreparer preparer = (ProxyPreparer) config.getEntry(
    "com.sun.securejinime.hello.Client",
    "preparer", ProxyPreparer.class, new BasicProxyPreparer());
server = (Hello) preparer.prepareProxy(server);

System.out.println("Server says: " + server.sayHello());
System.exit(0);
}

```

4.2 Server Code Snip

In this Section, we will discuss some implementation details about server code in this Section because the role of Secure JiniME-capable device could be a server to offer services. The Secure JiniME server uses Kerberos to authenticate and login with KDC as follows:

```

protected void init() throws Exception {
    LoginContext loginContext = (LoginContext) config.getEntry(
        "com.sun.securejinime.hello.Server", "loginContext",
        LoginContext.class, null);
    if (loginContext == null) {
        System.out.println("No JAAS login is performed!");
    } else {
        loginContext.login();
        Subject.doAsPrivileged(
            loginContext.getSubject(),
            new PrivilegedExceptionAction() {
                public Object run() throws Exception {
                    initAsSubject();
                }
            }
        );
    }
}

```

```

        return null;
    }
    }, null);
}
}

```

In the code above, it first instantiates a LoginContext object then uses config object to configure itself and login with KDC. It behaves in the same way as a client as discussed in Section 4-1, the only different is the behavior of the method call (the doAsPrivileged() or doAS() method). After login, it instantiates proxy, discovery manager, and join manager objects then makes use of DiscoveryManagement object to discover lookup services. If succeeded, server joins local lookup service; otherwise, it prints error message.

```

protected void initAsSubject() throws Exception {
    /* Export the server */
    Exporter exporter = getExporter();
    serverProxy = (Hello) exporter.export(this);

    /* Create the smart proxy */
    Proxy smartProxy = Proxy.create(serverProxy);

    /* Get the discovery manager, for discovering lookup services */
    DiscoveryManagement discoveryManager;
    try {
        discoveryManager = (DiscoveryManagement) config.getEntry(
            "com.sun.securejinime.hello.Server", "discoveryManager",
            DiscoveryManagement.class);

        /* Get the join manager, for joining lookup services */
        JoinManager joinManager =
            new JoinManager(smartProxy, null /* attrSets */, getServiceID(),
                discoveryManager, null /* leaseMgr */, config);
    } catch (NoSuchEntryException e) {

```

```

        /* Print error message */
        System.out.println("Can't find local lookup service!");
    }
}

```

4.3 HTTPMD Code Snip

A preferred class provider computes HTTP Message Digest (HTTPMD) URLs to use as the class annotation for classes in the application and bootstrap classpath.

```

public class MdClassAnnotationProvider
    extends PreferredClassProvider
{
    private final String codebase =
        HttpmdUtil.computeDigestCodebase(
            System.getProperty("export.codebase.source"),
            System.getProperty("export.codebase"));

    public MdClassAnnotationProvider()
        throws IOException, MalformedURLException
    {
        super(false);
    }

    protected String getClassAnnotation(ClassLoader loader) {
        return codebase;
    }
}

```

The following system properties control the HTTPMD URLs created:

- `Dexport.codebase`: Space-separated lists of the HTTPMD URLs for use as codebase. The digest values specified in the URLs will be ignored. The path portion of the URLs, without the message digest parameters, will be used to


specify the source files, relative to the source directory, to use for computing message digests.

- `Dexport.codebase.source`: The name of the directory containing the source files corresponding to the URLs in the codebase

4.4 Jini Bridge Code Snip

The Secure JiniME Sessile-Client-to-Mobile-Service Bridge uses Kerberos to authenticate and login with KDC as follows:

```
final Configuration config = ConfigurationProvider.getInstance(args);
LoginContext loginContext = (LoginContext) config.getEntry(
    "com.sun.securejinime.hello.SCMSBridge", "loginContext",
    LoginContext.class, null);
if(loginContext == null){
    System.out.println("No JAAS login performed!");
}
else{
    loginContext.login();
    Subject.doAsPrivileged(
        loginContext.getSubject(),
        new PrivilegedExceptionAction() {
            public Object run() throws Exception {
                builderAsSubject(config);
                return null;
            }
        }, null);
}
```



We can see the code above, it first instantiates `LoginContext` object and uses `config` object to configure itself and login with KDC. It behaves the same as client discussed in Section 4-1. After login, it instantiates discovery manager object then uses service

discovery manager object to discovery services and stores all information in lookup cache. Afterward, it constructs each M.S.Proxy for each service to export service proxy and issues run method to login Jini federation KDC, discovers Jini federation Lookup service, registers itself and exports M.S.B proxy.

```
protected void builderAsSubject(Configuration config) throws Exception {  
  
final int maxMatches = 100;  
ServiceItem[] serviceitem;  
ServiceDiscoveryManger serviceDiscovery;  
MSBridge msBridge[];  
  
try{  
    serviceitem = new int[maxMatches];  
    serviceDiscovery = (ServiceDiscoveryManager) config.getEntry(  
        "com.sun.securejinime.hello.SCMSBridge", "serviceDiscovery",  
        ServiceDiscoveryManager.class, null);  
  
    /* Use ServiceDiscoveryManager to discovery services and  
        store all information in Lookup Cache */  
  
    ServiceCache serviceCache =  
        serviceDiscovery.createLookupCache(null, null, serviceChange);  
    serviceitem = serviceCache.lookup(null, maxMatches);  
    msBridge = new MSBridge[maxMatches];  
  
    for(int i=0; i<maxMatches; i++){  
  
        /* Each M.S.Bridge setMSProxy will prepare for M.S.Proxy,  
            The serviceCache implements LookupCache which uses  
            serviceChange that implement ServiceDiscoveryListener,  
            if any serives change occur */  
  
        msBridge[i].setMSProxy(serviceitem[i]);  
        msBridge[i].run();  
  
    }  
  
}
```

```

        /* run( ) method will do the same behavior as client: login
        Jini federation KDC, discover Jini federation Lookup
        service, register itself and export M.S.B proxy */
    }
} catch (NoSuchEntryException e){
    /* Print error message */
    System.out.println("No such entry!");
}
}
}

```

The Mobile-Client-to-Sessile-Service Bridge code is exactly similar with the Sessile-Client-to-Mobile-Service Bridge code, so that we omitted it.

4.5 Properties Setting

As mentioned, we have discussed some implementation details of Secure JiniME coding. In subsequent contents, we will describe properties setting of Secure JiniME. We will discuss Kerberos requirements in Section 4.5.1, setting security properties in Section 4.5.2, login configuration file in Section 4.5.3, and policy file in Section 4.5.6.

4.5.1 Kerberos Requirements

The Kerberos LoginModule required for the JAAS authentication may not be available in all vendors' JREs. We will be using the LoginModule for Kerberos provided in the JRE. In order to run programs, you will need to access a Kerberos setup installation as described in Appendix B. You may also need a krb5.conf Kerberos configuration file and an indication as to where that file is located.

Typically, the default realm and the KDC for that realm are indicated in the Kerberos krb5.conf configuration file. However, you can instead specifying these values by setting the following properties to indicate the realm and KDC, respectively:

java.security.krb5.realm and *java.security.krb5.kdc*. If you set one of these properties you must set all of them both. Also note that if you set these properties, then no cross realm authentication is possible unless a *krb5.conf* file is also provided from which the additional information required for cross-realm authentication may be obtained.

If you set values for these properties, then they override the default realm and KDC values specified in “*krb5.conf*”. The *krb5.conf* file is still consulted if values for items other than the default realm and KDC are needed. If no *krb5.conf* file is found, then the default values used for these items are implementation-specific. The algorithm to locate the *krb5.conf* file is the following:

- If the system property *java.security.krb5.conf* is set, its value is assumed to specify the path and file name.
- If that system property value is not set, then the configuration file is looked for in the directory
 - `<java-home>\lib\security` [Windows]
 - `<java-home>/lib/security` [Linux]

Here `<java-home>` refers to the directory where the JRE was installed.

- If the file is still not found, then an attempt is made to locate it as follows:
 - `c:\winnt\krb5.ini` [Windows]
 - `/etc/krb5.conf` [Linux]
- If the file is still not found, and the configuration information being searched for is *not* the default realm and KDC, then implementation-specific defaults are used. If, on the other hand, the configuration information being searched for is the default

realm and KDC because they weren't specified in system properties, and the krb5.conf file is not found either, then an exception is thrown.

4.5.2 Setting Security Properties

Some aspects of Java security [7] may be customized by setting security properties. A security property may be set *statically* or *dynamically*. To set a security property statically, you just add a line to the security properties file. You can modify or add to the contents of the file in order to change security property values. To specify a security property value in the security properties file, adding a line of the following form: *propertyName = propertyValue*. The value of the security property named “policy.provider” specifies the canonical name of the Policy implementation class to be installed. For example:

```
policy.provider=net.jini.security.policy.DynamicPolicyProvider
```

To set a security property dynamically, you can call the “java.security.Security.setProperty” method, substituting the appropriate property name and value: *Security.setProperty* (“*propertyName*”, “*propertyValue*”); For example:

```
Security.setProperty(“policy.provider”, “ net.jini.security.policy  
.DynamicPolicyProvider”)
```

4.5.3 Login Configuration File

Java Authentication and Authorization Service (JAAS) [8] provides a set of classes that authenticate a user. This typically means that JAAS-enabled application requires a user to log into it, much like the user logs into his computer. JAAS also provides a set of classes that authorize users to perform certain operations; this authorization is very similar to the

permissions-based authorization that the default sandbox grants to codes loaded from particular locations or signed by particular entities.

A JAAS-enabled application works like this: (1). the program asks the user to log in, obtaining a user login object. (2). the program executes a method call (the `doAS()` or `doAsPrivileged()` method), passing in the user login object and the code that should be executed on behalf of that user. (3). within the context just created, the program executes code that requires a specific permission. This code results in a call to the security manager (and hence the access controller) to see if the appropriate permission is granted.

For developers, there are two steps to using JAAS: they must make a call to authenticate the user and execute particular methods on behalf of that user. For administrators, there are three steps: they must configure a set of login modules, configure a set of JAAS policy files, and set up the program's environment correctly. A login configuration file consists of one or more entry, the structure of each entry as follows:

```
<name used by application to refer to this entry> {  
    <LoginModule><Flag><LoginModule options>;  
    <optional additional LoginModules, flags and options>;  
};
```

Each LoginModule-specific entry has the following subparts:

- LoginModule: a class implementing the desired authentication technology.
- Flag: a value indicating whether success of the preceding LoginModule as the follows:

- **REQUIRED:** The login module is required to succeed. Regardless of whether it succeeds or fails, however, authentication still proceeds down the login module list.
- **REQUISITE:** The login module is required to succeed. If it succeeds, authentication continues down the login module list. If it fails, control immediately returns to the application; authentication does not proceed down the login module list.
- **SUFFICIENT:** The login module is not required to succeed. If it does succeed, control immediately returns to the application; authentication does not proceed down the login module list. If it fails, authentication continues down the login module list.
- **OPTIONAL:** The login module is not required to succeed. If it succeeds or fails, authentication still proceeds down the login module list.
- **LoginModule options:** value for any desired options in the specified LoginModule implementation.

For example:

```
com.sun.securejinime.hello.Server {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="config/servers.keytab"
    storeKey=true
    doNotPrompt=true
    principal="${serverPrincipal}";
};
```

"com.sun.securejname.hello.Server" which specifies that the LoginModule to be used to do the user authentication is the *Krb5LoginModule* in the "com.sun.security.auth.module" package and that this "Krb5LoginModule" is required to "succeed" in order for authentication to be considered successful. The Krb5LoginModule succeeds only if the name and password supplied by the user are successfully used to log the user into the Kerberos KDC.

If "useTicketCache=true" means that Krb5LoginModule will use the native ticket cache to get the TGT available in it; If "useKeyTab=true" means that the secret key from the keytab is used to authenticate the principal="{serverPrincipal}" and both the TGT obtained from the Kerberos KDC and the secret key are stored in the Subject's private credentials set; The "keyTab=config/krb-servers.keytab" means that server's keytab is stored in servers.keytab file. The "doNotPrompt=true" means there is no prompt to ask your user name and password that we stored them in a .keytab file. Once you've written a login configuration file, you must write at least two policy files, which we will discuss in next section, for the application: a JAAS policy file that grants users particular permissions based on how they were authenticated and a standard Java policy file.

4.5.4 Policy Files

Policy file [10] grant statements can optionally includes one or more Principal field. Inclusion of a Principal field indicates that the user or other entity represented by the specified Principal, executing the specified code, has the designated permissions. Thus, the basic format of a grant statement is as follows:

```

grant <signer(s) field>, <codeBase URL>
    <Principal field(s)> {
        permission perm_class_name "target_name", "action";
        ....
        permission perm_class_name "target_name", "action";
    };

```

Where each of the signer, codeBase and Principal field(s) are optional and the order between the fields doesn't matter. A Principal field looks like the following: *Principal* *Principal_class* "*principal_name*". That is, it is the word "Principal" (where case doesn't matter) followed by the (fully qualified) name of a Principal class which implements the "java.security.Principal" interface and a principal name. All Principal objects have an associated name that can be obtained by calling their getName method. The format used for the name is depending on each Principal implementation. The type of Principal placed in the Subject created by the Kerberos authentication mechanism used by this thesis is "javax.security.auth.kerberos.KerberosPrincipal", so that is what should be used as the *Principal_class* part of our grant statement's Principal designation. User names for KerberosPrincipals are of the form "name@realm". Thus, if the user name is "mjones" and the realm is "KRBNT-OPS.ABC.COM", the full *principal_name* designation to use in the grant statement is "mjones@KRBNT-OPS.ABC.COM".

The absence of the signer(s) field signifies "any signer". That is, whether the code is signed and by whom does not matter. Its value, when specified, is a comma-separated list of one or more aliases that are mapped, using the keystore, to certificates. When the *signedBy* value is a comma-separated string containing names of multiple signers, for example "Ada, John, Tom", the relationship is "AND", not "OR". A codeBase value

indicates the code source location (URL); you grant the permission(s) to code from that location. The absence of a codeBase entry signifies “any code”; that is, where the code originates from does not matter. A permission entry [9] must begin with the keyword *permission* and terminate with a semicolon. The *perm_class_name* specified after the word *permission* in the previous grammar would be a specific permission type, such as “java.io.FilePermission” or “java.lang.RuntimePermissio”. The “*target_name*” is required for all permission types. The “*action*” is optional for some permission types and required for others. For instance, the “java.io.FilePermission” requires the target to specify the file and the action that specifies the permitted type of file access (“read” or “write” or both). The following is the security policy for Kerberos server:

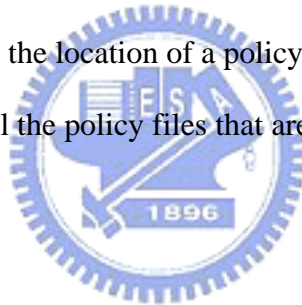
```
/* Grant the local JARS all permissions */
grant codeBase "file:lib${/*}*" {
    permission java.security.AllPermission;
};

/* Grant permissions to client principal */
grant principal
    javax.security.auth.kerberos.KerberosPrincipal "${clientPrincipal}"
{
    /* Call sayHello method */
    prmission com.sun.securejinime.hello.ServerPermission "sayHello";
};

/* Grant permissions to all principals */
grant {
    /* Call getProxyVerifier method */
    permission com.sun.securejinime.hello.ServerPermission "getProxyVerifier";
};
```

It is possible to include more than one Principal field in a grant statement. If multiple Principal fields are specified, then the permissions in that grant statement are granted only if the Subject associated with the current access control context contains *all* of those Principals. To grant the same set of permissions to different Principals, create multiple grant statements where each lists the permissions and contains a single Principal field designating one of the Principals.

It is also possible to specify an additional or a different policy file when invoking execution of an application. This can be done via the "-Djava.security.policy" command line argument, which sets the value of the *java.security.policy* property. For example, if you use "java -Djava.security.manager -Djava.security.policy=*someURL* SomeApp" where *someURL* is a URL specifying the location of a policy file, then the specified policy file will be loaded in addition to all the policy files that are specified in the security properties file.



4.6 Running the Program

In attempt to execute Secure JiniME program, you should prepare for running environment first of all: install J2ME CDC Foundation and RMI as described in Appendix A, then place jaas.jar into directory <java.home>/jre/lib, Jini packages that you want to use as described in Appendix C into your project directory. These packages you can download from "java.sun.com/products/jaas/index-10.html" and "www.jini.org", Kerberos V5 authentication systems installation in "MIT's Kerberos homepage". Lastly, to execute the application with the Login utility, do the following:

1. Writing all java files, config files, log files and policy files, then replace "your_user_name@your_realm" in policy file with your user name and realm.

2. Compile all *.java file
3. Create each java file's .jar file for codebase
4. Place the *.java and *.class files into a directory; *.config, *.login, and *.policy into subdirectory "config"; *.sh or *.bat into subdirectory "scripts"; *.jar into subdirectory "lib"; *.password, *.keystore, *.cert into subdirectory "prebuiltkeys".
We recommend it, but you can put all together in a directory and modify all setting depending on your situation.
5. Execute the class, specifying by an appropriate arguments:
 - i. by -Djava.security.manager that a security manager should be installed,
 - ii. by -Djava.security.policy=<your policy> that the policy file to be used is .policy.
 - iii. by -Djava.security.properties=*path-to-file*/security.properties defines a system property using when running your application
 - iv. by -Djava.rmi.server.RMIClassLoaderSpi=<your classloaderspi>, https or httpmd server.
 - v. by -Djava.security.auth.login.config=<your config> that the login configuration file to be used is .conf.
 - vi. by -Djava.security.krb5.realm=<your_realm> that your Kerberos realm is the one specified.
 - vii. by -Djava.security.krb5.kdc=<your_kdc> that your Kerberos KDC is the one specified.
 - viii. by -Djava.protocol.handler.pkgs=net.jini.url, to includes a protocol handler for HTTPMD URLs.
 - ix. by -Dexport.codebase=<your codebase directory>, a space-separated list of the HTTPMD URLs for use as codebase.

- x. by `-Dexport.codebase.source= httpmd://$host:8080/<codebase name>.jar;<crpto>=0`, the name of the directory containing the source files corresponding to the URLs in the codebase. `<crypto>=md5,sha` and so on.

The following is the example to execute server:

```
java -Djava.security.manager=  
-Djava.security.policy=config/krb-server.policy  
-Djava.security.auth.login.config=config/krb-server.login  
-Djava.protocol.handler.pkgs=net.jini.url  
-Djava.security.properties=config/dynamic-policy.security-properties  
-Djava.rmi.server.RMIClassLoaderSpi=  
com.sun.securejini.hello.MdClassAnnotationProvider  
-Dexport.codebase.source=lib  
-Dexport.codebase=httpmd://$host:8080/server-dl.jar\;md5=0 \  
-DclientPrincipal="$CLIENT"@"$REALM"  
-DserverPrincipal="$SERVER"@"$REALM"  
-DreggiePrincipal="$REGGIE"@"$REALM"  
-Djava.security.krb5.realm=$REALM  
-Djava.security.krb5.kdc=$KDC_HOST  
-jar lib/server.jar  
config/server.config
```

In addition to pass the name of your application as an argument to login, you would add any arguments required by your application. Type the full command on one line.

Multiple lines are used here for legibility. If the command is too long for your system, you

may need to place it in a .bat file (for Windows) or a .sh file (for UNIX) and then run that file to execute the command.



Chapter 5 Conclusion and Future Works

This thesis describes the design and implementation of Secure JiniME architecture. Since Secure JiniME architecture is running on JVM rather than running directly on the native operating system, programs of Secure JiniME architecture are interpreted by the JVM for the native operating system. This means that any computer system with the JVM installed can run a Secure JiniME program regardless of the computer system on which the application was developed originally. In attempt to implement the architecture proposed in this thesis, one should master Jini coding skill, network security, and computer network.

In this thesis, we have discussed JiniME architecture issues, security mechanism, J2ME middleware and Secure JiniME architecture. With the supported security mechanism, JiniME has become a strong architecture under distributed environment, but it is not vigorous enough. Therefore, future works include the support of key management which would not burden the architecture with memory size since J2ME CDC has supported security packages. It is also desirable to profile benchmarks on the secure architecture, so as to highlight performance bottlenecks, and study undiscovered security holes where further improvements can be focused on.

Appendix A: Building CVM

CDC cannot be built with an optional package unless a profile, such as Foundation Profile or Personal Basis Profile, is also included in build. Optional packages, such as RMI OP, must be downloaded separately. See <http://jcp.org> for more information on the optional packages available. For more porting information, please refer to “Porting Guide”, “Connected Device Configuration and Foundation Profile, Version 1.0.1 Java 2 Platform, Micro Edition”.

Setting The Path To GNU Tools

Before performing your first build, you must set the path to your GNU tools using the build option `CVM_GNU_TOOLS_PATH`. It is located in the file `build/share/defs.mk`. `CVM_GNU_TOOLS_PATH` must be set to the top level path containing your GNU tools. When a make is started, this path is used to invoke the necessary tools, including the compiler, assembler, and archiver. It is usually necessary to set this variable only once. If your compilation cannot find needed tools, it may be because of the setting to `CVM_GNU_TOOLS_PATH`. The following example shows how to use `CVM_GNU_TOOLS_PATH` in `build/share/defs.mk`:

```
ifeq ($(uname), Linux)
    CVM_GNU_TOOLS_PATH=/usr/bin
endif
ifeq ($(uname), SunOS)
    CVM_GNU_TOOLS_PATH=/net/tools/bin
endif
```

Optional Package Makefile Variables

There are two variables that must be included within each `defs_<pkgname>_pkg.mk` file for each optional package. They are:

`OPT_PKGS_SRCPATH` - the list of directories containing Java source files

`OPT_PKGS_CLASSES` - the list of classes that make up the optional package

For example, if the following are defined in `defs_<pkgname>_pkg.mk`:

```
MY_PACKAGE_SRCDIR = $(MY_ROOT)/src/mypackage/classes
```

```
MY_PACKAGE_CLASSES = my.class.Class1 my.class.Class2 etc.
```

Then the same makefile should contain:



```
OPT_PKGS_SRCPATH += $(MY_PACKAGE_SRCDIR)
```

```
OPT_PKGS_CLASSES += $(MY_PACKAGE_CLASSES)
```

Notice that the symbols plus “+” and equals ”=“ are extremely important if more than one optional package will be used. Failure to set `OPT_PKGS_SRCPATH` and `OPT_PKGS_CLASSES` correctly may cause compilation errors and/or the failure of your classes to be part of the build.

Optional Packages Command-line Syntax

The `OPT_PKGS` build option indicates that an optional package will be compiled along with the current build. The syntax of this flag is as follows:

OPT_PKGS=all | *<pkgname>*[,*<pkgname>*]

where *<pkgname>* is the name of the optional package and a comma is used, without spaces, to separate multiple package names. For example, the following command includes the RMI OP package in the Foundation Profile build:

```
make J2ME_CLASSLIB=foundation OPT_PKGS=rmi CVM_JAVABIN=javapath/bin
```

This produces several items, including the following two items in your corresponding build/linux-i686 directory, bin/cvm and lib/ foundation_rmi.jar, and sets the default boot class path accordingly. When OPT_PKGS is set to the value all, all available optional packages are part of the current build. The make process recognizes these available packages by the existence of their makefiles. For example, if build/share contains, among others, the files defs_rmi_pkg.mk, rules_rmi_pkg.mk, defs_io_pkg.mk, rules_io_pkg.mk, defs_graphics_pkg.mk, rules_graphics_pkg.mk the OPT_PKGS=all flag will recognize the RMI OP, I/O, and graphics packages and include them in the build in alphabetical order.

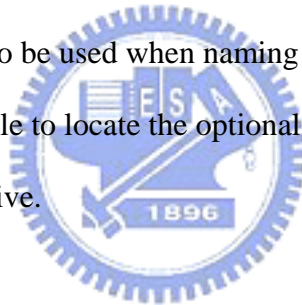
Naming Convention For Optional Package Makefiles

As with profile makefiles in build/share, each optional package must also have a definitions file and a rules file in build/share. In build/share, you will notice the files defs_cdc.mk, rules_cdc.mk, defs_foundation.mk, and rules_foundation.mk. The file name format for optional package makefiles is similar. The scheme is as follows for definitions: defs_*<pkgname>*_pkg.mk For example, defs_rmi_pkg.mk The scheme is as follows for rules: rules_*<pkgname>*_pkg.mk For example, rules_rmi_pkg.mk.

The additional `_pkg` portion of the makefile name alerts the build that the file is an optional package makefile. When `OPT_PKGS` is specified on the command line, the make process automatically looks for these makefiles and includes them into the build. It does this by taking each name defined with `OPT_PKGS` and using the filenaming scheme to identify the optional package makefiles. For example, with the command:

```
make J2ME_CLASSLIB=foundation OPT_PKGS=rmi CVM_JAVABIN=javapath/bin
```

The make process will automatically search for and include the files `defs_rmi_pkg.mk` and `rules_rmi_pkg.mk` in the build. Therefore, it is necessary that the optional package name set with `OPT_PKGS` also be used when naming the makefiles. Failure to do so will result in the build not being able to locate the optional package makefiles. Optional package names are case sensitive.



Build Quick Test Procedure

To test for rudimentary operation of the system, complete the following steps:

1. Change directory into your platform's bin directory `build/linux-i686/bin` using: `cd build/linux-i686/bin`.
2. Test for rudimentary operation of the system using:
`./cvm -Djava.class.path=../testclasses.zip HelloWorld`

Above step should print "Hello world" to your screen, indicating a successful build.

Appendix B: Kerberos Setup Instructions

Introduction

Beginning in version 1.4.1, the Java™ 2 SDK, Standard Edition provides three client-side Kerberos tools: kinit, klist, and ktab in its Windows and Linux distributions. In this document, we explain how to use these tools to obtain Kerberos Ticket Granting Tickets (TGTs) and keytabs on their provided platforms.

Create keytab files

Create a keytab file, if it does not exist already, and add entries using:

```
ktab -a <principal_name> <password> -k <keytab_name>
```

For example, to create a keytab file named as krb-servers.keytab in the config directory, which contains three entries whose principal names are server, phoenix, and reggie, and passwords are serverpw, phoenixpw, and reggiepw respectively, you would use the following:

```
ktab -a server serverpw -k config/krb-servers.keytab
```

```
ktab -a phoenix phoenixpw -k config/krb-servers.keytab
```

```
ktab -a reggie reggiepw -k config/krb-servers.keytab
```

To check what entries are in a keytab file, use: `ktab -l -k <keytab_name>`

Create TGT cache

kinit can be used to obtain and cache Kerberos TGTs. Here are links to the documentation of this command on various platforms:

Linux: <http://java.sun.com/j2se/1.4.2/docs/tooldocs/linux/kinit.html>

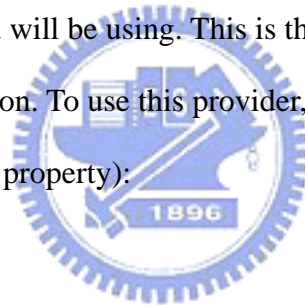
Windows: <http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/kinit.html>

Appendix C: Jini JAR Files

C.1 Platform JAR files

In this release we have extended our platform requirements to be more than just the Java 2 Platform, Standard Edition.

jsk-policy.jar: This JAR file contains a security policy provider, `DynamicPolicyProvider`, which supports dynamic granting of permissions at run time. Although use of this policy provider is not required, its use is highly recommended when deploying secure applications and services. To permit effective use of this policy provider, it must first be installed as an extension in the Java 2 SDK (or JRE) that you will be using. This is the *only* JAR file that we recommend installing as an extension. To use this provider, you need to define a *security* property (*not* a system property):



```
policy.provider=net.jini.security.policy.DynamicPolicyProvider.
```

In some file (for example, named `security.properties`) and then define a *system* property using when running your application:

```
-Djava.security.properties=path-to-file/security.properties.
```

If you are using a Java 2 SDK (or JRE) from a vendor other than Sun, you may also need to set a security property:

```
net.jini.security.policy.PolicyFileProvider.basePolicyClass=provider-class
```

in the same `security.properties` file, where *provider-class* is the vendor's default policy provider class, which typically can be found as the value of the `policy.provider` security property in the `jre/lib/security/java.security` file of the Java 2 SDK installation.

jsk-platform.jar: This JAR file contains classes and interfaces that we have chosen to include in all of our applications, and have also chosen to assume are available in all other applications that receive objects from our applications. This JAR file primarily contains classes and interfaces that are typically referenced in:

- *service provider* resources to control the configuration of `RMIClassLoader`, `TrustVerifier`, `IntegrityVerifier`, `ServerContext`, `DiscoveryFormatProvider` providers.
- `ConfigurationFile` source files, but which are unlikely to be referenced directly by the applications and services being configured (in particular, classes for creating `Exporter` and `ProxyPreparer` instances and their components).
- dynamically downloaded code, but which we believe are not themselves reasonable to download (in particular, because they are needed to bootstrap proxy trust verification, or because their implementations require extraordinary permissions)

plus all of the classes and interfaces that their implementations directly or indirectly depend on. This JAR file contains all of the classes and interfaces in the following namespaces (including all subpackages): `net.jini.activation`, `net.jini.config`, `net.jini.constraint`, `net.jini.core`, `net.jini.export`, `net.jini.id`, `net.jini.iiop`, `net.jini.io`,

net.jini.jeri, net.jini.jrmp, net.jini.loader, net.jini.security, com.sun.jini.discovery, net.jini.url, and plus the following classes: KeyStores, ConfigUtil, LogManager, ConstrainableLookupLocatorTrustVerifier, any other classes or interfaces found in this JAR file should be considered implementation details. This JAR file includes a protocol handler for HTTPMD URLs. To enable this handler, you need to specify the system property:

`-Djava.protocol.handler.pkgs=net.jini.url`

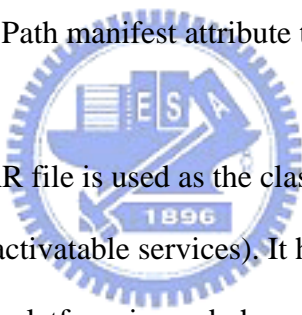
in any application that creates or receives such URLs. In particular, any service that uses an HTTPMD URL in its codebase, and any client that wants to download code from such a service, should set this system property. It is important to understand that the contents of this JAR file do *not* constitute a "standard" platform; this JAR file is simply our choice of *a* platform likely to be useful for deployments of this starter kit. Depending on your deployment requirements, you may want to alter the contents of this JAR file (but if you do so, you also might have to make changes to other JAR files).

jsk-resources.jar: This JAR file is referenced in the Class-Path manifest attribute of jsk-platform.jar, and configures the following specific service providers to be used: RMIClassLoader provider(PreferredClassProvider), TrustVerifier providers(ConstraintTrustVerifier, BasicJeriTrustVerifier, SslTrustVerifier, KerberosTrustVerifier, ProxyTrustVerifier, IntegrityVerifier providers (HttpmdIntegrityVerifier, HttpsIntegrity Verifier, FileIntegrity Verifier), ConstrainableLookupLocatorTrustVerifier, DiscoveryConstraintTrustVerifier), ServerContext provider (JrmpServerContext), DiscoveryFormatProviders which

discovery formats are: net.jini.discovery.ssl, net.jini.discovery.plaintext, net.jini.discovery.x500.SHA1withRSA, net.jini.discovery.kerberos, net.jini.discovery.x500.SHA1withDSA. Depending on your deployment requirements, you may want to alter the contents of this JAR file to use different sets of providers.

C.2 Service Starter JAR files

start.jar: This executable JAR file is the primary entry point for the Service Starter. It acts as both the class path for the container virtual machine (VM) for the Java platform that executes non-activatable services, and as the setup VM for activatable services. It has a Class-Path manifest attribute that references jsk-platform.jar.



sharedvm.jar: This JAR file is used as the class path for the activation group VM (the container VM for activatable services). It has a Class-Path manifest attribute that references both jsk-platform.jar and phoenix-init.jar.

destroy.jar: This executable JAR file can be used to destroy an existing activation group and all of the activatable services registered in that group.

group.jar: This JAR file contains the implementation of an activatable service used to destroy an existing activation group. It is typically used as the class path in a SharedActivatableServiceDescriptor in a configuration file passed to destroy.jar.

group-dl.jar

The codebase JAR file for group.jar, used in the same service descriptor as above

for group.jar.

C.3 Service JAR files

There are two primary JAR files for each service, a service JAR file with a name of the form *service.jar*, and a codebase JAR file with a name of the form *service-dl.jar*. The service JAR file contains the service implementation itself, and can be thought of as the *class path* for the service. The service JAR file generally contains three versions of the service: a transient (non-activatable, non-persistent) version; a non-activatable, persistent version; and an activatable, persistent version. In this release, Mahalo and Mercury do not yet provide transient versions. The service JAR file is designed to be run under the Service Starter, and as such is not directly executable. The codebase JAR file is used as the *codebase annotation* for the service; it contains classes and interfaces that are used by the service's proxies and trust verifiers, and that clients need to dynamically download. The codebase JAR file also contains a preferred list for use by clients that have the PreferredClassProvider enabled. Neither the service JAR file nor the codebase JAR file include any of the classes or interfaces found in *jsk-platform.jar*: the Service Starter container (either *start.jar* or *sharedvm.jar*) provides these classes for the service implementation, and clients are expected to have *jsk-platform.jar* in their class path.

fiddler.jar: The service JAR file for the Fiddler implementation of the lookup discovery service.

fiddler-dl.jar: The codebase JAR file for Fiddler.

mahalo.jar: The service JAR file for the Mahalo implementation of the transaction manager service.

mahalo-dl.jar: The codebase JAR file for Mahalo.

mercury.jar : The service JAR file for the Mercury implementation of the event mailbox service.

mercury-dl.jar: The codebase JAR file for Mercury.

norm.jar : The service JAR file for the Norm implementation of the lease renewal service.



norm-dl.jar: The codebase JAR file for Norm.

outrigger.jar : The service JAR file for the Outrigger implementation of the JavaSpaces(TM) service.

outrigger-dl.jar: The codebase JAR file for Outrigger.

reggie.jar : The service JAR file for the Reggie implementation of the lookup service.

reggie-dl.jar: The codebase JAR file for Reggie.

C.4 Activation JAR files

This release contains a configurable RMI activation system daemon implementation named Phoenix that we recommend using instead of rmid when deploying activatable versions of services.

phoenix.jar: This executable JAR file is used to run Phoenix.

phoenix-dl.jar: The codebase JAR file for Phoenix. Unlike rmid, Phoenix *requires* clients to dynamically download code.

phoenix-init.jar: This JAR file (or its contents) must be included in the class path of any activation group VM that is created by Phoenix to run activatable objects.

phoenix-group.jar: This JAR file contains the default ActivationGroup implementation for Phoenix. Normally it is not referenced explicitly, but is instead loaded automatically from the same directory as phoenix.jar.

C.5 Tools JAR files

tools.jar: This JAR file contains tools for: checking configuration files; checking for missing serialVersionUID fields; computing class dependencies; providing HTTP service; generating message digests; generating HTTPMD URLs; and generating wrapper JAR files. When used as an executable JAR file, it runs the ClassServer.

browser.jar: This executable JAR file is used to run the example ServiceBrowser.

browser-dl.jar: The codebase JAR file for the Service Browser.

C.6 Other JAR files

jini-core.jar: This JAR file contains all of the classes and interfaces in the `net.jini.core` namespace. No direct use of this JAR file is made in this release; it is simply provided as a possible convenience to developers.

jini-ext.jar: This JAR file contains all of the classes and interfaces in the `net.jini` namespace, plus all of the classes and interfaces that their implementations directly or indirectly depend on, including service providers in the `com.sun` namespace that might want to be configured in resources. The classes and interfaces in the `net.jini.core` namespace are not included directly; instead this JAR file has a Class-Path manifest attribute that refers to `jini-core.jar`. Note that the Class-Path manifest attribute does *not* refer to `jsk-resources.jar`, so no service providers are configured by default when using this JAR file at runtime. Any classes or interfaces in the `com.sun` namespace that appear in this JAR file should be considered implementation details. No direct use of this JAR file is made in this release; it is simply provided as a possible convenience to developers.

outrigger-logstore.jar: A secondary JAR file referenced by the Class-Path manifest attribute of `outrigger.jar`, containing classes for the default storage implementation used by the persistent versions of Outrigger. In practice you should not need to refer directly to this JAR file.

prebuilt-outrigger-logstore.jar: A secondary JAR file containing specially

postprocessed classes in `outrigger-logstore.jar`. This JAR file is never used a runtime, it is only used if you attempt to rebuild the starter kit from sources and do not have the necessary ObjectStore PSE Pro for Java postprocessor tool. In practice you should not need to refer directly to this JAR file.

pro.zip: A secondary JAR file referenced by the `Class-Path` manifest attribute of `outrigger-logstore.jar`, containing the ObjectStore PSE Pro for Java classes used by the default storage implementation of the persistent versions of Outrigger. In practice you should not need to refer directly to this JAR file.

sdm-dl.jar: This JAR file contains proxy and trust verifier classes that generally need to be included in the codebase of any application that uses the implementation of the `ServiceDiscoveryManager` provided in this release.

sun-util.jar: This JAR file contains various classes and interfaces in the `com.sun.jini` namespace that developers might want to reference. Of particular interest are the non-standard administrative interfaces implemented by our services, as well as the interfaces needed by services that want to run under the Service Starter. We make no claims about the API stability of these classes and interfaces. No direct use of this JAR file is made in this release; it is simply provided as a possible convenience to developers.

References

- [1]. Ken Arnold, *The Jini™ Specifications*, Second Edition, Addison Wesley, 2001.
- [2]. W. Keith Edwards, *Core JINI*, Prentice Hall PTR, 1999.
- [3]. Sun Microsystems, “Jini.org, The Community Resource for Jini Technology,”
<http://www.jini.org/>
- [4]. Alan Kaminsky, “JiniME: Jini™ Connection Technology for Mobile Devices,”
03-Aug-2000, <http://www.cs.rit.edu/~anhinga/whitepapers/JiniMEWhitePaper/>
- [5]. Sun Microsystems, “Java 2 Platform, Micro Edition (J2ME),”
<http://java.sun.com/j2me/index.jsp>
- [6]. Massachusetts Institute of Technology, “Kerberos: The network authentication protocol,” <http://web.mit.edu/kerberos/www/index.html>
- [7]. Li Gong, Gary Ellison, Mary Dageforde, *Inside Java™ 2 Platform Security*, Second Edition, Addison Wesley, May 2003.
- [8]. Sun Microsystems, “Java Authentication and Authorization Service (JAAS) Overview,” <http://java.sun.com/products/jaas/overview.html>
- [9]. Sun Microsystems, “Permission in Java™ 2 SDK,”
<http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html>
- [10]. Sun Microsystems, “Default Policy Implementation and Policy File Syntax,”
<http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html>