Jini – an Infrastructure for Dynamic Service Networks

Peer Hasselmeyer
IT Transfer Office (ITO)
Darmstadt University of Technology

- Java Intelligent Network Infrastructure
- Jini Is Not Initials
Overview

- Jini, what’s that?
  - motivation
  - overview
- RMI/Serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - more details
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions
  - JavaSpaces
- Summary
Jini

- Infrastructure ("middleware") for dynamic, cooperative, spontaneously networked systems
  - facilitates writing / deploying distributed applications
Jini

- Infrastructure ("middleware") for dynamic, cooperative, spontaneously networked systems
  - facilitates writing / deploying distributed applications
- framework of APIs with useful functions / services
- helper services (discovery, lookup,...)
- suite of standard protocols and conventions
Jini

- Infrastructure ("middleware") for dynamic, cooperative, spontaneously networked systems
  - facilitates writing / deploying distributed applications

- services, devices, ... find each other automatically ("plug and play")
- added, removed components
- changing communication relationships
- mobility
Jini

- Based on Java and implemented in Java
  - may use RMI (Remote Method Invocation)
  - typed (object-oriented) communication structure
  - requires JVM / bytecode everywhere
  - code shipping
- Strictly service-oriented
Service Paradigm

• Everything is a service (hardware / software / user)
  – like object-orientation: “everything” is an object
  – e.g. persistent storage, software filter, help desk, ...

• Services are defined by interfaces and provide their functionality via their interfaces
  – services are further characterized by their attributes (e.g. “600 dpi”, “version 21.1”)
Service Paradigm

- Jini’s run-time infrastructure offers mechanisms for adding, removing, and finding services
- Services (and service users) “spontaneously” form a system (“federation”)
A Jini Federation

- Picture mailing service
- Picture storage service
- Network
- Camera 1 (client)
- Camera 2 (client)
- Print service
What Kind of Services?

• Devices:
  – printer, fax machine, ...
  – storage, persistency, configuration, ...
  – computation, ...

• Software:
  – spell checking, format conversion, ...
  – online banking, stock trading, ...
  – tourist guide, local maps, hotels, restaurants, ...
Spontaneous Networking

• Objects in an open, distributed, dynamic world find each other and form a transient community
  – cooperation, service usage, ...

• Problem: no a priori knowledge about existence, interface, functionality, and trustworthiness of services
  – “traditional” client/server model: server knows nothing about its clients, but client knows the server, its interface (API, protocol) and its functionality
Spontaneous Networking

- Objects must be able to find each other
- Must be fast, easy, and automatic

But: what for?
What Does Tomorrow Look Like?

• Increasing number of internet users
• Powerful PDAs and notebooks
• Increasing mobility
• New wireless information devices
• Numerous mobile networked devices
What Does Tomorrow Look Like?

• Numerous processors in embedded systems
  – e.g. software updates for your washing machine, internet-ready microwave, ...

• Trend towards ubiquitous networks and spontaneous networking / service access
  – high bandwidth, wireless, cheap
Middleware

- Approach from a different direction: “middleware”
  - components to help build and deploy distributed applications (development-time vs. run-time)
  - located between the application logic and the underlying physical network and operating system
Middleware

Jini: Motivation
Middleware

- Abstraction from tedious network programming wanted
- Abstraction from differing machine architectures wanted
  - problem: data encoding (e.g. big/little endian, integer size, array storage layout, ...)
- Components for recurring problems (e.g. naming service, security service, ...)
Problems with Current Middleware

- Systems hide network from programmer
  - programmers don’t have to deal with the network and its inherent problems (unreliability, latency, bandwidth, ...)
  - no exception handling
- Data is moved to the computation
  - “classical” client/server paradigm
  - not always most efficient solution
  - problem: different data formats (byte-order)
  - but: program code is usually not available everywhere (different system architectures, installation, ...)

Jini: Motivation
Middleware: Partial Failure

- Difference between local and remote call: total vs. partial failure
- Components might be unreachable: server crash, broken network connection, ...
- Problem: how to distinguish unreachable from slow server
Middleware: Partial Failure

• Components / middleware have to be able to deal with partial failure
• Unreachability of one component should not (or only minimally) affect other components
  – e.g. Internet: server crash does not affect other servers
• Jini offers mechanisms to deal with partial failure
Some Fallacies of Common Distributed Computing Systems

- The idealistic view...
  - the network is reliable
  - latency is zero
  - bandwidth is infinite
  - topology doesn’t change
  - there is one administrator

- ...isn’t true in reality
  - Jini addresses some of these issues
  - at least it does not hide or ignore them
Bird’s-Eye View on Jini

- Jini consists of a number of APIs
- Is an extension to the Java 2 platform dealing with distributed computing
- Is a layer of abstraction between application and underlying infrastructure (network, OS) → “middleware”
- Extension of Java in three dimensions
# Jini’s Extensions to Java

- Extensions regarding networked systems

<table>
<thead>
<tr>
<th>Infrastructure model</th>
<th>Programming model</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• JVM</td>
<td>• Swing</td>
<td>• Transaction Service</td>
</tr>
<tr>
<td>• RMI</td>
<td>• Beans</td>
<td>• Enterprise JavaBeans</td>
</tr>
<tr>
<td>• ...</td>
<td>• ...</td>
<td>• ...</td>
</tr>
<tr>
<td>Jini</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Discovery</td>
<td>• Transactions</td>
<td>• JavaSpaces</td>
</tr>
<tr>
<td>• Lookup</td>
<td>• Leasing</td>
<td>• Transaction Manager</td>
</tr>
<tr>
<td>• Join</td>
<td>• Distributed Events</td>
<td></td>
</tr>
</tbody>
</table>

- Distinction may seem blurred
Jini’s Use of Java

• Jini requires JVM (as bytecode interpreter) and (currently) RMI
  – homogeneity in a heterogeneous world
  – is this realistic?
• But: devices that are not “Jini-enabled” can be managed by a software proxy which resides at some place in the net
  – e.g. “Device Bay”
• Safety of Java applies to Jini as well
  – type safety, array bounds, sand box, ...
Jini vs. Mobile Agents

• Jini is **Java-based**
  – most mobile agent platforms are based on Java
  – Java bytecode often used as a universal machine language
  – applets are working instances of the mobile code paradigm

• Jini uses the **mobile code** paradigm
  – service proxies are sent to the client (via the lookup service)
Jini vs. Mobile Agents

- Mobile agents need an **infrastructure**
  - Jini is an infrastructure for highly dynamic distributed systems
  - Jini provides elementary services and functionality
    - e.g. agent coordination with JavaSpaces
- But: Jini is *not* a mobile agent platform!
  - further services required
  - however: Jini (and similar systems) are of general interest to computer scientists (→ ubiquitous computing, distributed systems)
Overview

• Jini, what’s that?
  – motivation
  – overview

• RMI/Serialization

• Jini infrastructure
  – lookup service
  – discovery & join protocols
  – programming example
  – more details

• Jini programming model
  – leasing
  – distributed events

• Jini services
  – transactions
  – JavaSpaces

• Summary
RMI: Introduction

• RMI: “Remote Method Invocation”
• Java’s native middleware
• “Regular” Java: method invocation within one Java VM
• By using RMI objects can call methods of objects running in a different Java VM
• Java VMs can be distributed among multiple machines in a network
Remote Invocation

Java VM
Client

Method invocation
Result
Network

Java VM
Server
What To Use RMI For?

• Server objects offer their functionality (“services”: data, computation) to clients
• Clients can access them via a network
• RMI offers mechanism to bring clients and servers together
• Goal: clients can use remote server objects in (almost) the same way as they use local objects
  – partial failure possible
  – larger latency
How Does RMI Work?

- Basic difference between local and remote objects: different Java VMs
- Idea: create a proxy for the remote object in the local Java VM
  - so-called “stub”
  - signatures of methods are identical to methods in remote server object
  - handles communication with remote object
- “Skeleton” in server is counterpart to stub
  - forwards parameter to “actual” server object
  - returns result to stub
How Does RMI Work?

RMI: Introduction

Java VM

Client

Stub

Network

Method invocation

Result

Java VM

Server

Skeleton
RMI: Introduction

More Details

Programmer's view

Client Program
  Stub
  Remote Reference Layer
  Transport Layer

Server Program
  Skeleton
  Remote Reference Layer
  Transport Layer

Network
RMI Details

- Client: invokes methods of an object
- Stub: implements “remote interface” of server object, passes on parameters
- Remote Reference Layer: maps object references to machines and objects
- Transport Layer: manages connections between machines and handles data transfer
- Skeleton: passes data to server object
- Server: implements functionality
Getting Access to Objects

- Clients must get stubs for remote objects
- RMI registry: naming service
  - mapping name (string) → object
  - location of registry (machine name and port number) must be known
  - name of server object must be known
  - e.g. \\
    "//server.tud.de:2222/HelloServer"
  - usually reference to first object only; grants access to further objects ("factory pattern")
    - e.g. root: database object, access to entries
- Jini
Example: Remote Interface

- Remote methods are defined by “remote interfaces”:

```java
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```

- Methods can always throw RemoteExceptions
  - server or network may fail
- Interfaces are marked as remote by extending tag interface java.rmi.Remote
Example: Server Object

- Server implements remote interface and extends `UnicastRemoteObject`

```java
public class HelloImpl
    extends java.rmi.server.UnicastRemoteObject
    implements Hello {
    public HelloImpl()
        throws java.rmi.RemoteException {
        super(); // UnicastRemoteObject
    }
    public String sayHello()
        throws java.rmi.RemoteException {
        return "Hello World!";
    }
}
```
Example: Server Object

public HelloImpl()
throws java.rmi.RemoteException {
    super(); // UnicastRemoteObject
}
public String sayHello()
throws java.rmi.RemoteException {
    return "Hello World!";
}

- **UnicastRemoteObject** handles RMI related work (make object known to RMI, relay calls)
- Server has to realize interface’s functionality
Example: Server Startup

public class HelloStart {
    public static void main(String args[]) {
        try {
            HelloImpl obj = new HelloImpl();
            java.rmi.Naming.rebind(  
                “//server.tud.de/HelloServer”, obj);
        } catch (Exception e) {}  
        [...wait and unregister object on exit...]
    }
}

- Create server object
- Register it with RMI registry
  - address/port of registry must be known
  - name for object must be supplied
Example: Client

```java
public class HelloClient {
    public static void main(String args[]) {
        System.setSecurityManager(
            new java.rmi.RMISecurityManager());
        try {
            Hello obj = (Hello) java.rmi.Naming.lookup(
                “//server.tud.de/HelloServer”);
            System.out.println(obj.sayHello());
        } catch (Exception e) {}
    }
}
```

- Security manager enables class download
- Get server’s stub from RMI registry
- Use the server
Example: Deployment

• Naming Service must be running: rmiregistry

• Stub has to be created: rmic HelloImpl
  – creates classes HelloImplStub and HelloImplSkel
  – no skeletons needed in Java 2 (option -v1.2)
Example: Deployment

• Security policy needed to access sockets:
  - `java -Djava.security.policy=policy HelloStart`
  - `java -Djava.security.policy=policy HelloClient`

• For non-local environments classes (here: `HelloImpl_Stub`) have to be put onto an HTTP server and server’s codebase has to be supplied
Serialization

- Parameters and return values (arbitrary objects) have to be transferred between client and server
- Copy of object has to be sent to remote JVM
- Solution: transform objects into stream of bytes ("serialization")
- Problem: objects may contain references to other objects
  - all referenced objects have to be recursively serialized ("transitive closure")
Serialization

Source JVM → Byte stream → Destination JVM

Network

RMI: Serialization
Serialization

- Transforming an object into a stream of bytes
- Object consists of its code and its (data) state (values of variables)
- Code is stored in class files
- State is dynamic and only available at runtime
- Take a “snapshot” of this state
How Does Serialization Work?

• Realized by Java’s introspection facilities
• All non-transient, non-static fields are recursively written to a byte-stream
• Simple types have predefined storage functions
• Objects’ types (full class names, e.g. java.util.List) and signatures are written
How Does Serialization Work?

- Objects are only written once, even if referenced multiple times
- “Serial version UID” to control versions
  - calculated from the signature of a class
  - can be overridden manually to mark compatible class evolution
- Objects have to implement tag interface `java.io.Serializable`
- Predefined serialization semantics can be overridden by implementing certain functions (`writeObject()`, `readObject()`)
What Can Be Serialized?

- All primitive types (int, boolean, etc.)
- “Remote objects” (by sending serialized stubs)
- Base types (String, Integer, ...) that implement `java.io.Serializable`
- Most container classes (Hashtable, Vector, ...) if and only if they only contain serializable objects
- Some AWT and Swing classes
- Your classes, if they implement `java.io.Serializable`
What Cannot Be Serialized?

• Base types that do not implement `Serializable`
  – “wrapper classes” possible (by inheritance)

• “Low level” classes:
  – input and output streams
  – threads
  – peer classes

• Objects that reference non-serializable objects
What To Use Serialization For?

- Write to file: make objects persistent
  - configurations
  - (Enterprise) Java Beans
  - to continue working on the same state later

- Write to socket:
  - to transfer objects to another JVM
  - parameter passing
  - mobile agents
Code Mobility

- Objects consists of two parts:
  - code (in Java class files)
  - state (values of attributes, execution pointers)
- RMI transfers only data state
- Problem: code is usually not locally available at recipient (i.e. not listed in its classpath)
- Solution: code can be downloaded at run-time (e.g. from an HTTP server)
Code Mobility

State
a=5
b=7

Data from
source JVM

Implementation
a=?
b=?

Reconstructed object
a=5
b=7

Code from
(web) server
Codebase

- Code can be downloaded, but: where from?
- Location of code is transferred together with its state ("codebase")
- Codebase is a list of URLs
- URL might point to
  - directory containing the class tree
  - JAR file containing the classes
- Codebase is set as a property when starting a JVM
Security

• Unknown objects are executed on local machine (like applets)

• Access restrictions desired; specified by security policies
  – fine-grained control of local resources (especially storage, network) possible
  – rights are granted based on
    • where code came from (network, local file)
    • who signed code

grant signedBy "sysadmin", codeBase "http://server.tud.de/-" {
    permission java.net.SocketPermission "*:1024-", "connect";
};
Overview

- Jini, what’s that?
  - motivation
  - overview
- RMI/Serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - more details
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions
  - JavaSpaces
- Summary
Jini Infrastructure

• Main components are:
  – **lookup service** as repository / naming service / trader
  – **protocols** based on TCP/UDP/IP
    • discovery, join, lookup
  – **proxy objects**
    • transferred from service to clients
    • represent the service locally at the client
Jini Infrastructure

• Goal: spontaneous networking and formation of federations without prior knowledge of local network environment
• Problem: How do service providers and clients get to know their local environment?
Lookup Service

- Similar to RMI registry and CORBA naming service
- Repository of service providers
- Tasks:
  - registration of services ("advertising")
  - distribution of services (clients find services)
- Main component of every Jini federation
- May be redundant and hierarchically organized (similar to DNS)
Jini Federation

- Lookup Service
- register
- lookup
- use

Client

Service
Lookup Service

• Reference implementation uses Java RMI for communication
  – required: objects must be able to migrate through the net

• Not only name and address of a service are stored (as in traditional naming services), but also set of attributes
  – e.g.: printer (color: true, dpi: 600, ...)
  – attributes may be complex classes
    • e.g.: user interface(s)
Discovery: Finding a LUS

- Goal: Finding a lookup service without knowing anything about the network
  - to advertise (register) a service
  - to find (look up) an existing service
- Discovery protocol:
  - multicast to well-known address/port
  - lookup service replies with a serialized object (interface `ServiceRegistrar`)
    - proxy object of lookup service gets loaded to discovering entity
    - communication with LUS via this proxy
Where is the lookup service?

Discovery

foreign network

Multicast Request

Lookup Service

Proxy

Reply

That’s me!!

Communication

Jini: Protocols
Join: Registering a Service

- Service provider already received a proxy of the lookup service
- Provider uses this proxy to register its service (`register()`)
- Provider sends the lookup service
  - its service proxy
  - attributes that further describe the service
- Provider can now be found and used in this Jini federation
Join

Service

Lookup Service
Proxy

Entry 1  Entry 2  Entry n

Registration

Lookup Service

Service database in LUS

Jini: Protocols
Lookup: Searching Services

- Client knows lookup service (e.g. via discovery protocol)
- Looking for certain (type of) service
- Creates query for lookup service
  - in form of a “service template”
  - matching by service type and/or attributes and/or registration number of service
  - wildcards possible
Lookup: Searching Services

- Lookup service returns one or more matches (or zero)
- Selection usually done by client
- Service use by calling functions of service proxy
- Any protocol between proxy and service provider possible
Lookup

Client

Lookup Service
Proxy

Lookup

Communication

Lookup Service database in LUS

Entry 1
Entry 2
Entry n

Service proxy

Entry 1
Entry 2
Entry n

Service proxy

Jini: Protocols
Jini Programming Example

- How is a service implemented?
- How does a client get access to a service?
- How is a service described?
The “Running Example”

Lookup Service

Printer proxy

Communication between application and printer via function calls of proxy

Office application

Printer

Any protocol
The “Running Example”

- Printer registers itself with the office’s lookup service
- Printer provides print interface as its service

```java
public interface Print extends java.rmi.Remote {
    public PrinterParams readPrinterParams()
        throws java.rmi.RemoteException;
    public SuccessCode print(Document doc)
        throws java.rmi.RemoteException;
    [...etc...]  
}
```
The “Running Example”

• Implementation of service consists of provider and proxy
• Proxy is stored in the lookup service and will be transferred to clients upon request
• Protocol between proxy and service provider depends on implementation and is not stipulated by Jini
• Example uses RMI
Implementing Service Provider

public class PrintImpl extends UnicastRemoteObject implements Print {
    
    public PrinterParams readPrinterParams() throws RemoteException {
        // something should be done here
    }

    public SuccessCode print(Document doc) throws RemoteException {
        // something else should be done here
    }

    [...]
}

There’s no Jini in here!
public class PrintRegistration {
    public static void main(String[] args) {
        System.setSecurityManager(
            new RMISecurityManager());
    
    } // main
}

Print service = new PrintImpl();
Entry[] attribute = new Entry[1];
attribute[0] = new ServiceInfo(
        "Print Service",
        "HyperClear", "Shiny Inc.",
        "2000", "", "08/15");

JoinManager jmgr = new JoinManager(service, attribute,
        (ServiceIDListener) new Listener(),
        new LeaseRenewalManager());

}
public class MCExample implements DiscoveryListener {
    public static void main(String[] args) {
        /* Security Manager etc... */
        LookupDiscovery ld =
            new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        ld.addDiscoveryListener(new MCExample());
        ...
    }

    public void discovered(DiscoveryEvent ev) {
        ServiceRegistrar[] regs = ev.getRegistrars();
        ServiceRegistrar reg = regs[0];
        Class[] cl = new Class[] { Print.class }; ServiceTemplate tmpl =
            new ServiceTemplate(null, cl, null);
        Print proxy = (Print) reg.lookup(tmpl);
    }

    public void discarded(DiscoveryEvent e) {}
Overview

- Jini, what’s that?
  - motivation
  - overview
- RMI/Serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - more details
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions
  - JavaSpaces
- Summary
Discovery

- Protocols to find lookup services
- Multicast request protocol
  - client asks for local lookup services
  - no prior knowledge of local network necessary
- Unicast request protocol
  - used to contact known lookup services
  - works across subnet boundaries and over the Internet
Discovery

• Multicast announcement protocol
  – protocol for lookup services to announce their presence

• Example: printer registers with the office via multicast, but gets service for software updates from a dedicated lookup server via unicast discovery
Multicast Request Protocol

- No information about the host network needed
- Active search for lookup services
- Discovery request is multicast packet
  - IP-centric
  - multicast address for discovery is 224.0.1.85
  - default port number of lookup services is 4160
  - hexadecimal subtraction: $\text{CAFE}_{16} - \text{BABE}_{16} = 4160_{10}$
Multicast Request Protocol

- Discovery request is multicast packet
  - recommended time-to-live is 15
  - usually does not cross subnet boundaries
- Discovery reply is establishment of a TCP connection
  - port for reply is included in multicast request packet
Multicast Request Protocol

JavaVM client/service

Multicast request client

Multicast response server

Multicast request server

Multicast response client

TCP

UDP

Service registrar

Multicast request packet
Multicast Request Packet

- Protocol version (int)
- Reply port (int)
- Number of known LUSes (int)
  - LUS 1 (ServiceID)
  - ... 
  - LUS n (ServiceID)
- Number of groups (int)
  - Group 1 (String)
  - ... 
  - Group m (String)

max. 512 bytes
Unicast Discovery Protocol

- Used to contact lookup services with known locations
- Uses TCP (unicast) connections to port 4160
- Simple request-response protocol

![Diagram showing the process of establishing a TCP connection, making a unicast request, and receiving a unicast response between a JavaVM client/service and a JavaVM lookup service.]

Jini: More Details
Unicast Packets

- Unicast Request (client → lookup service)
  - Protocol version (int)
- Unicast Response (lookup service → client)
  - LUS proxy (MarshalledObject)
  - Number of groups (int)
  - Group 1 (String)
  - ... 
  - Group m (String)
Multicast Announcement Protocol

- Used by lookup services
- Announces the availability of lookup services
- Based on multicast UDP
- Announcements are sent periodically
  - recommended: every 120 seconds
- Receivers listen for announcements on well-known port
- If not yet known, receiver starts unicast discovery of this service
Multicast Announcement Packet

- Protocol version (int)
- Host for unicast discovery (String)
- Port for unicast discovery (int)
- Service id (ServiceID)
- Number of groups (int)
- Group 1 (String)
- ...
- Group m (String)

max. 512 bytes
Lookup Service Details

![Diagram of Lookup Service](image)

- Service
- Client
- Service usage
- Registration
- Lookup

Lookup Service
Groups

- There may be lots of lookup services in a large Jini system
- Idea: split lookup services into groups
  - group name: simple text identifier
  - clients/services always announce interest in certain group(s)
  - unwanted groups are ignored
  - services do not explicitly register for certain groups, rather with LUS that carries group
- e.g. one lookup service per department: accounting, production, research, ...
LUS: Proxy Interface

```java
public interface ServiceRegistrar {
    ServiceRegistration register(ServiceItem item,
                                   long leaseDuration)
                                   throws RemoteException;
    java.lang.Object lookup(ServiceTemplate tmpl)
                                   throws RemoteException;
    ServiceMatches lookup(ServiceTemplate tmpl,
                           int maxMatches)
                           throws RemoteException;
    [...]
}
```

Used by service providers

Used by clients

Jini: More Details
Join: More Features

• To join, a service supplies:

  ServiceItem(Object service,
              ServiceID id,
              Entry[] attributes)

  – its proxy
  – its ServiceID (if previously assigned; “universally unique identifier”)
  – set of attributes
  – set of groups (or ALL_GROUPS)
  – (possibly empty) set of specific lookup services to join
Join: Service IDs

To join, a service supplies:

```java
ServiceItem(Object service,
            ServiceID id,
            Entry[] attributes)
```

Service-ID is a 128 bit “universally unique identifier”
- generated by the lookup service when registering for the first time
- service has to reuse it for all later registrations
- service has to make it persistent
Join: More Features

- Service waits a random amount of time after start-up
  - prevents packet storm after restarting a network segment
- Registration with a lookup service is bound to a lease
  - service has to renew its lease periodically
- Discovery and join can be handled by objects of class `JoinManager`
Lookup: More Features

• Client looks for service(s) registered with a lookup service
• Any combination of search criteria possible:
  – ServiceID
  – service type
  – certain attributes
Lookup: More Features

• **Client creates** `ServiceTemplate`  

```java
ServiceTemplate(ServiceID serviceID, java.lang.Class[] serviceTypes, Entry[] attrSetTemplates)
```

• Template filled with interfaces, entries and/or ServiceID

• Wildcards possible, represented by **null**

• Attributes: only exact matching possible (no “larger-than”, ..)

• No query language
Entries

• Difference to “traditional” naming services
• Not only a name for a service
• Properties:
  – set of attributes
    • e.g.: PrinterParams (dpi: 600, type: color, …)
  – every serializable data type is possible
  – data and methods
  – complex classes possible
    • e.g. different user interfaces (AWT, Swing, speech, …)
public class Name extends AbstractEntry {
    public String name;
    public Name() {}
    public Name(String name) {
        this.name = name;
    }
}

gin class PrinterEntry extends AbstractEntry {
    public PrinterType type;
    public Integer pagesPerSecond;
    public PrinterEntry() {}
    public PrinterEntry(PT type) {
        this.type = type;
    }
}

gin class AWTGUIEntry extends AbstractEntry {
    public Panel panel;
    public GUIEntry() {}
    public GUIEntry(Panel panel) {
        this.panel = panel;
    }
}
Entries (Examples)

```java
public class Name extends AbstractEntry {
    public String name;
    public Name() {}
    public Name(String name) {
        this.name = name;
    }
}

public class PrinterEntry extends AbstractEntry {
    public PrinterType type;
    public Integer pagesPerSecond;
    public PrinterEntry() {}  
    public PrinterEntry(PT type) {
        this.type = type;
    }
}

public class AWTGUIEntry extends AbstractEntry {
    public Panel panel;
    public GUIEntry() {}  
    public GUIEntry(Panel panel) {
        this.panel = panel;
    }
}

Complex data structures
```
# Lookup Service vs. Naming Service

<table>
<thead>
<tr>
<th>Naming service</th>
<th>Lookup service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description by name (text) only</td>
<td>Description by ServiceItems</td>
</tr>
<tr>
<td>• e.g. /devices/printers/ → printers</td>
<td>• type &amp; attributes</td>
</tr>
<tr>
<td>• search by name only</td>
<td>• arbitrary search dimensions</td>
</tr>
<tr>
<td>Name implies type</td>
<td>Type is explicit</td>
</tr>
<tr>
<td>Standardized naming conventions</td>
<td>Standardized interfaces</td>
</tr>
<tr>
<td>Usually no expiration of entries (heartbeat, keep-alive)</td>
<td>Services have to renew their entries in the lookup service periodically (leasing)</td>
</tr>
<tr>
<td>Identified by (static) address; clients need to know address</td>
<td>Discovery; no network information necessary</td>
</tr>
<tr>
<td>groups can be modeled by addresses</td>
<td>group concept</td>
</tr>
</tbody>
</table>
Proxy: Features

• All communication between clients and services is handled by service’s proxy
• Proxy object is stored in the lookup service upon registration
• Serialized object
• Implements one or more service interfaces
• Service type is defined by the type of the interface(s)
Proxy: Features

- Upon request, stored object is sent to the client as a local proxy of the service.
- If needed, client retrieves necessary classes.
  - Class location is stored in codebase (URL).
- Client communicates with service provider via service proxy: client invokes methods of the proxy object.
- Client only needs to know interface.
- Proxy implementation hidden from client.
Proxy: Implementation

- Implementation of service functionality is independent from service interface
- Partition of service functionality depends on service implementer’s choice
- Parts of or whole functionality may be executed by the client (within the proxy)
Proxy: Implementation

• When dealing with large volumes of data, it usually makes sense to preprocess parts of or all the data
  – e.g.: compressing video data before transfer
Jini 1.1

- More classes belong to standard Jini (i.e. `net.jini.*`), e.g. JoinManager

- Helper classes
  - JoinManager, LeaseRenewalManager
  - Discovery utilities, protocol utilities
  - ServiceDiscoveryManager & LookupCache

- Helper services
  - LookupDiscoveryService
  - EventMailbox
  - LeaseRenewalService
ServiceDiscoveryManager

• Eases handling of multiple lookup services
• Supports different access patterns: on-demand, cached, reactive
• LookupCache
  – states of LUSes are mirrored: increased communication and memory use
  – immediate access to available services
  – usefulness depends on usage scenario, especially useful for frequent and low-latency queries
public class LCExample {
    public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        ServiceDiscoveryManager sdm = new ServiceDiscoveryManager(null, null);
        Class[] cl = new Class[] { Print.class };
        ServiceTemplate tmpl = new ServiceTemplate(null, cl, null);
        LookupCache cache = sdm.createLookupCache(tmpl, null, null);
        Print proxy = (Print) cache.lookup(null);
    }
}
Overview

- Jini, what’s that?
  - motivation
  - overview
- RMI/Serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - more details
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions
  - JavaSpaces
- Summary
Leases

• Lease is contract between two parties
• Leases introduce notion of time
  – resource usage is restricted to a certain time frame
  – interaction is modeled by repeatedly expressing interest in some resource:
    • I’m still interested in X
      – renew lease periodically
      – lease renewal can be denied
    • I don’t need X anymore
      – cancel lease or let it expire
      – lease grantor can use X for something else
Leases

- Lease is a contract between two parties.
- Lease grantor is willing to let lease taker (client) access a certain resource for a certain amount of time.
- Lease can be extended (renewed) if resource access is still desired.
  - Extension can be denied by grantor.
- Resource is released upon lease expiration or cancellation (by client).
What To Use Leases For?

• For allocating hardware and software resources intelligently
  – no administration necessary (for removal of stale resource reservations)
  – e.g. persistent storage, ...

• Distributed “garbage collection”
  – lease expired → “garbage”

• Inside Jini
  – registrations with lookup services
  – resources: transactions, event registrations
Distributed Events

• Objects in a JVM can register interest in certain events of another object in a different (remote) JVM
  – network failure
  – crossing of event notifications
  – late and lost messages

• “publisher/subscriber” model

• Architecture:

1. Registration
2. Event occurs
3. Send notification
Distributed Events

- Delivery separated from event semantics
- Different delivery behaviors desired by applications, e.g. guaranteed, in-order,…
- Interfaces do not reflect particular delivery behavior
- Event mechanism can be extended to accommodate different application semantics
Distributed Events (Example)

- Management application wants to know about printers running out of paper
- Gets printer’s proxy object from lookup service and registers for notifications
Distributed Events (Example)

- Printer could implement the interface:

```java
public interface PrinterEvents extends Remote {
    public EventRegistration register(
        long eventID,
        MarshalledObject handback,
        RemoteEventListener toInform,
        long leaseLength)
    throws RemoteException;
}
```

- “handback” is returned to the notified object in every notification. Used to attach arbitrary information to events.
- Registrations are leased.
- Reference to the object to be notified.

[Type of event to be notified about (e.g. out of paper).]
Distributed Events (Example)

- Management application (or dedicated listener object) implements interface `RemoteEventListener`

```java
public interface RemoteEventListener extends Remote, java.util.EventListener {
    void notify(RemoteEvent theEvent) throws UnknownEventException, RemoteException;
}
```

- Method `notify()` will be called for every event
Distributed Events (Example)

- Printer informs clients about events by sending `RemoteEvents`

```java
toInform.notify(new RemoteEvent(this, eventID, seqNum, handback));
```

- Event is unambiguously identified by tuple `<source, id, sequenceNumber>`

- Notification is synchronous
  - allows event source to know that event was delivered
Overview

- Jini, what’s that?
  - motivation
  - overview
- RMI/Serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - more details
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions
  - JavaSpaces
- Summary
Transactions

• Transactions encapsulate a number of operations
  – “all or nothing” semantics
  – operations appear to happen simultaneously (from outside transaction)
  – method to ensure consistency

• Two-phase commit protocol
  – method to perform transactions
  – Two phases:
    • vote on transaction validity
    • commit/abort operations
Transactions

- Central: a manager
  - ensures consistency: each transaction participant will ultimately “commit” or “abort”
  - performs 2PC protocol
- Example: transfer money from one bank account to another
Transactions: Properties

- Design goal: maximum flexibility, minimum number of interfaces
- Objects must implement certain interface to participate in transactions
- Traditionally transaction manager enforces certain semantics (ACID properties)
- Jini separates protocol from semantics:
  - 2PC performed by manager
  - semantics implemented by participants
Distributed Transactions in Jini...

- ...are no transactions in a traditional (database) sense
- “Lightweight” transaction
- ACID properties
  - atomicity / consistency / isolation / durability
  - each participant implements these properties as he sees fit
    - e.g.: transient objects do not need persistency
  - main property is atomicity, other properties are “sometimes” optional
- Transactions are leased from manager
Transactions: Participants

- Transaction manager
  - coordinates transaction
  - Jini service
  - implements interface TransactionManager

- Clients
  - initiate transactions

- Transaction participants
  - objects that perform transactional operations
  - implement interface TransactionParticipant
  - participation is initiated by join operation
Transactions: Semantic Objects

- Transactions have no fixed semantics
- Participants need to know the type of a transaction → “semantic objects”
  - Java type defines the kind of transaction
  - class `Transaction` tells objects to use their standard transaction semantics
- Participants accept only certain transaction types
- e.g. DBTransaction
  - requires database semantics
  - transient objects not allowed (durability)
2PC Protocol: Details

• Client starts a transaction
  – get transaction manager from lookup service (lookup TransactionManager)
  – call to create() starts transaction
    • usually indirectly via semantic object factory

• Participation in a transaction
  – participants find out about transaction upon function invocation
    • receive Transaction object
    • call transaction manager’s join() method
  – implement TransactionParticipant
2PC Protocol: Details

- Client starts a transaction
  - get transaction manager from lookup service (lookupTransactionManager)
  - call to create() starts transaction
    - usually indirectly via semantic object factory

- Participation in a transaction
  - participants find out about transaction upon function invocation
  - receive Transaction object
  - call transaction manager's join() method
    - implement TransactionParticipant

```java
public interface TransactionManager {
    TransactionManager.Created create(...);
    void commit(...)
    void abort(...) 
    void join(...)
    int getState(...)
}
```

```java
public interface TransactionParticipant {
    void commit(TransactionManager mgr, long id)
    void abort(TransactionManager mgr, long id)
    int prepare(TransactionManager mgr, long id)
    int prepareAndCommit(TransactionManager mgr, long id)
}
```

- Implement TransactionParticipant
JavaSpaces

• Tool for developing distributed applications
• Platform for exchanging objects between distributed applications ("shared blackboard")
• "Bag full of objects"
• Realizes distributed persistency
• Jini service
  – implemented completely in Java (and RMI)
  – interaction with service via local proxy
What To Use JavaSpaces For?

• Models “object flow”
  – e.g. producer / consumer applications
  – load-balancing

• Job-oriented view
  – jobs / events are put into the space and picked up “eventually”, e.g. workflow

• Build-in “good” properties
  – concurrent access possible
  – write / read are atomic operations
  – access within transaction possible
JavaSpaces: Objects

- JavaSpace = “a bag full of objects”
- Entries in a JavaSpace service:
  - `net.jini.core.entry.Entry` – known as `Entry` interface in Jini
    - strongly typed by Java type system
    - two entries of different classes are not equal, even if they encapsulate the same data types
      - Entry A `{Integer, Char}` ≠ Entry B `{Integer, Char}`
- Classes: include data/state and methods \Rightarrow behavior
Basic Operations: Write

- **write** puts an entry into a JavaSpace
- Uses copy of the object, never the object itself
- **write** returns a lease
  - entry in a JavaSpace has a limited duration
  - “garbage collection” in a JavaSpace service

![Diagram of write operation in JavaSpaces]

**Producer** → **JavaSpace**

**write(Entry)** → **Lease** → **JavaSpace**
Basic Operations: Read

- uses template (**Entry**)
  - exact value match
  - matches subclasses as well
  - **null** matches everything
  - if more than one match, select and return one entry at random
Basic Operations: Read

- **readIfExists** returns `null` if no match found
  - `read` waits until matching entry was found
- **take/takeIfExists** = `read` + removal of entry
Basic Operations: Notify

- Notify listeners at arrival of certain entries
- Uses Jini distributed events
- Clients can register with a JavaSpace
  - registration requires a “matching template”
  - clients implement `RemoteEventListener`
  - registration is leased
- Order of notification unspecified
  - Note: “first come, first serve” for registered listeners on take
Peculiarities

• Operations can be part of a transaction
• Entries can be “lost”:
  – write/take may cause a RemoteException
  – meaning: “may or may not have been successful”
    • write: often unproblematic (may be repeated)
    • take: entries may be lost
    • transactions necessary
  – example: moving entries between JavaSpaces (e.g. for load balancing)
    • transactions required
JavaSpaces and Linda

• Design based on Linda tuple spaces
  – David Gelernter (Yale University)

• Differences:
  – strong type checking and objects (methods/behavior)
    • matching on tuple structure, not just data
    • templates match subclasses as well
    • all (serializable) data types can be used as data
  – multiple spaces possible
  – leasing
  – no “eval” – a JavaSpace is just a repository
Overview

- Jini, what’s that?
  - motivation
  - overview
- RMI/Serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - more details
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions
  - JavaSpaces
- Summary
Summary

• Vision:
  – everything will be networked
  – everything will (be able to) communicate
  – ubiquitous network access
  – mobility/spontaneity as important paradigms

• Problems:
  – getting to know the network neighborhood
  – infrastructure should adapt to devices, not the other way round
  – distribution: communication, partial failure,…

• Possible solution: Jini
Conclusion

• A number of good ideas
  – simplicity, “less is more” → flexibility
  – discovery, join, lookup
  – extension of name services by attributes
  – leases, events, transactions → recurring design patterns

• Right direction
  – ubiquitous networks
  – mobility, spontaneity

• Individual concepts are not new, but together they offer new possibilities
But...

- Resource usage
  - each service usually requires a JVM
- Small devices
  - JVM and RMI required on device
  - adaptation to resource-restricted environment necessary (how?)
  - proxy objects are moved to client (memory)
- Standardized (base) interfaces
- What about the competitors (SLP, UPnP, e-speak, ...)?
Problem Areas

- **Security**
  - important in dynamic environments
  - user requires
    - confidentiality: encrypted communication
    - trust: authentication
  - services might use other services on behalf of the user
  - what about charging for services?
  - Java RMI security extension does not seem to be the solution

- **Scalability**
  - does Jini scale to a global level?
Suggested Reading

• Jini Homepage: http://www.sun.com/jini

• Jini Community: http://www.jini.org

• W. Keith Edwards: Core Jini, Prentice Hall, 1999
  – good motivation, very detailed
  – don’t be frightened by more than 700 pages (everything is said at least twice...)
Contact Information

Peer Hasselmeyer
IT Transfer Office (ITO)
Darmstadt University of Technology

Email: peer@ito.tu-darmstadt.de
WWW: http://www.ito.tu-darmstadt.de

Slides: http://www.ito.tu-darmstadt.de/publs/presentations/asama00.pdf