Jini

Peer Hasselmeyer
Darmstadt University of Technology

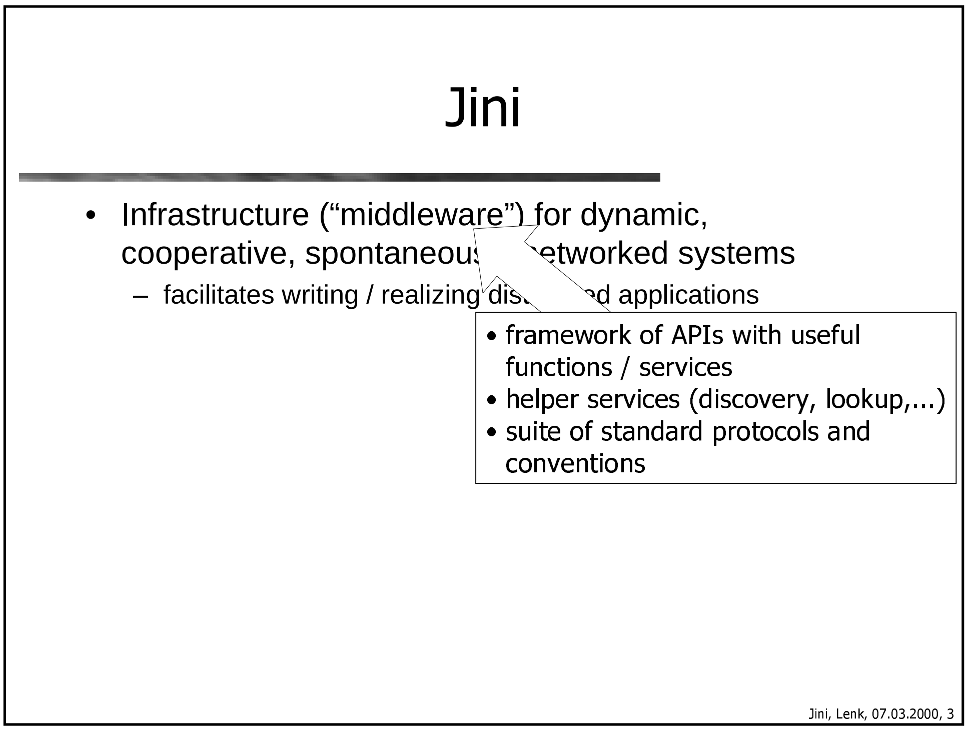
Friedemann Mattern
ETH Zürich

Andreas Zeidler
Darmstadt University of Technology

- **J**ava **I**ntelligent **N**etwork **I**nfrastructure
- **J**ini **I**s **N**ot **I**nitials

Jini

- Infrastructure (“middleware”) for dynamic, cooperative, spontaneously networked systems
  - facilitates writing / realizing distributed applications

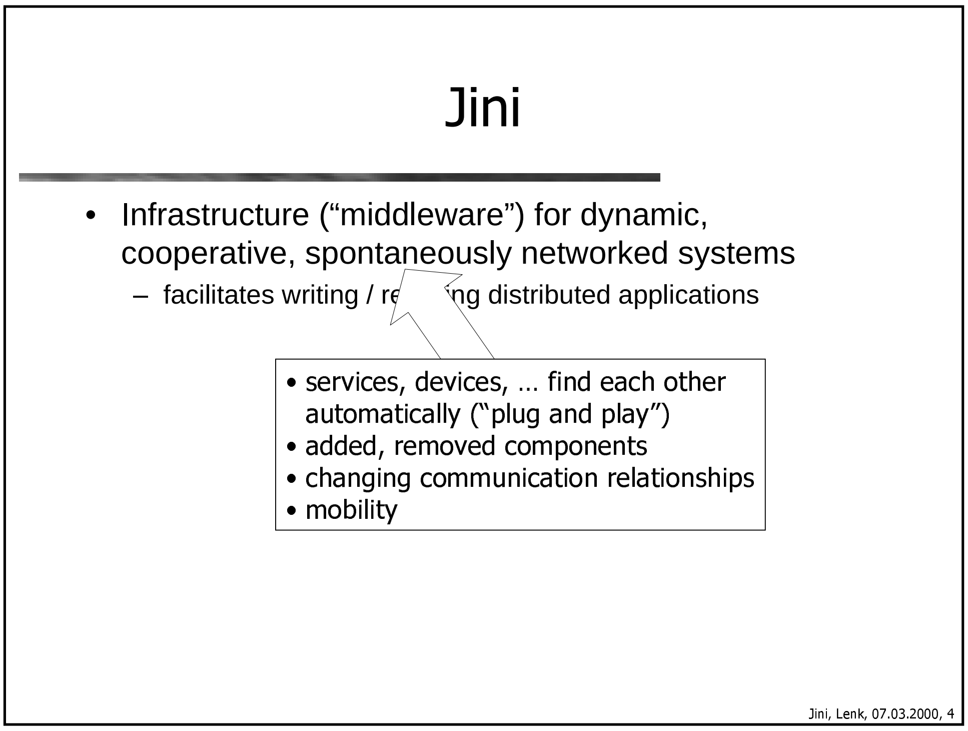


Jini

- Infrastructure (“middleware”) for dynamic, cooperative, spontaneous etworked systems
  - facilitates writing / realizing dis ed applications

• framework of APIs with useful functions / services
• helper services (discovery, lookup,...)
• suite of standard protocols and conventions

Jini, Lenk, 07.03.2000, 3



Jini

- Infrastructure (“middleware”) for dynamic, cooperative, spontaneously networked systems
  - facilitates writing / re ng distributed applications

• services, devices, … find each other automatically (“plug and play”)
• added, removed components
• changing communication relationships
• mobility

Jini, Lenk, 07.03.2000, 4

Jini

- Infrastructure (“middleware”) for dynamic, cooperative, spontaneously networked systems
  - facilitates writing / realizing distributed applications
- Based on Java and implemented in Java
  - typed (object-oriented) communication structure
  - may use RMI (Remote Method Invocation)
  - requires JVM / bytecode everywhere
  - code shipping
- Strictly service-oriented
  - everything is a service (hardware / software / user)
  - Jini system is a federation of services

Jini <===> Mobile Agents ?

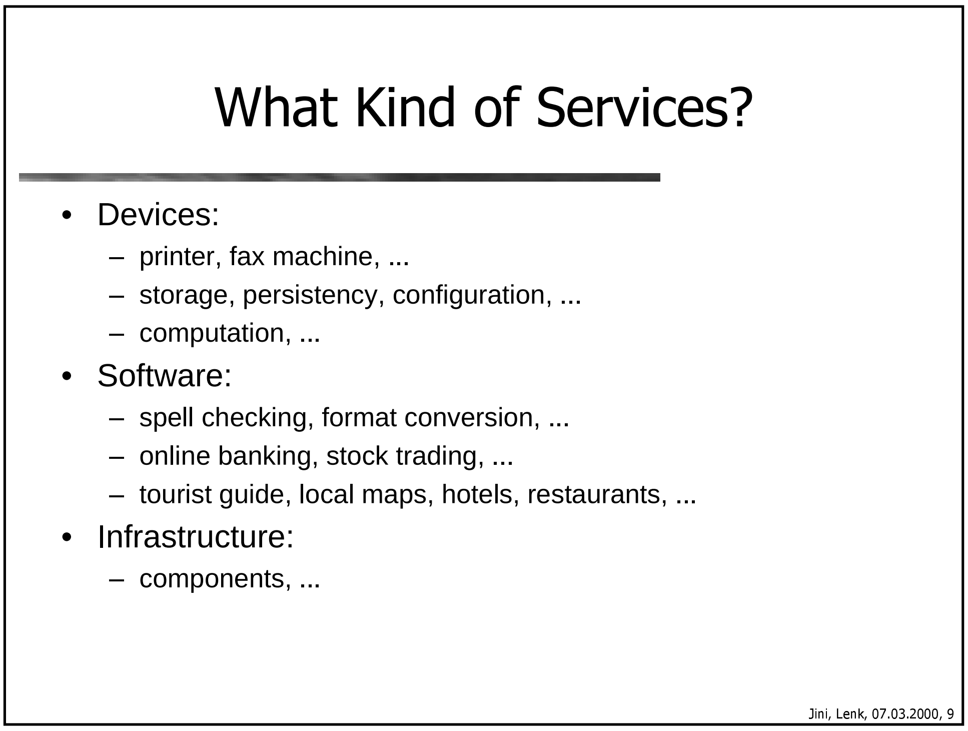
- Jini is **Java-based**
  - most mobile agent platforms are based on Java
  - Java bytecode often used as a universal machine language
  - applets are working instances of the mobile code paradigm
- Jini uses the **mobile code** paradigm
  - service proxies might be sent to the client (by the lookup server)
- Mobile agents need an **infrastructure**
  - Jini is an infrastructure for highly dynamic distributed systems
  - Jini provides elementary services and functionality
- But: Jini is *not* a mobile agent platform!
  - However: Jini (and similar systems) should be of general interest to computer science students (--> ubiquitous computing)

Overview

- Jini, what's that?
  - motivation
  - overview
- RMI
  - introduction
  - example
  - serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions and the transaction manager
  - JavaSpaces
- Summary

Service Paradigm

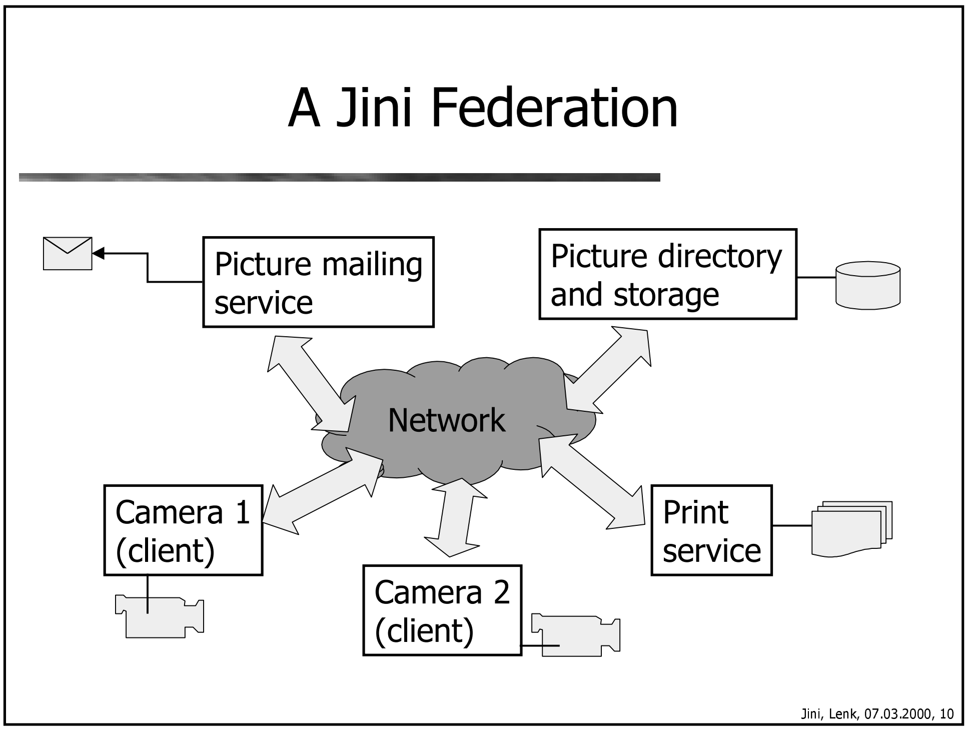
- Everything is a service (hardware / software / user)
  - like object-orientation: "everything" is an object
  - e.g. persistent storage, software filter, help desk, …
- Jini's run-time infrastructure offers mechanisms for adding, removing, finding, and using services
- Services are defined by interfaces and provide their functionality via their interfaces
  - services are characterized by their type and their attributes (e.g. "600 dpi", "version 21.1")
- Services (and service users) "spontaneously" form a system ("federation")



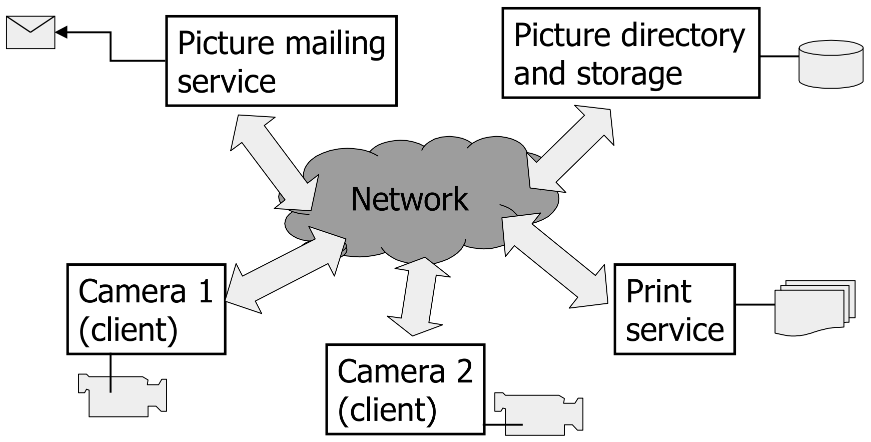
What Kind of Services?

- Devices:
  - printer, fax machine, ...
  - storage, persistency, configuration, ...
  - computation, ...
- Software:
  - spell checking, format conversion, ...
  - online banking, stock trading, ...
  - tourist guide, local maps, hotels, restaurants, ...
- Infrastructure:
  - components, ...

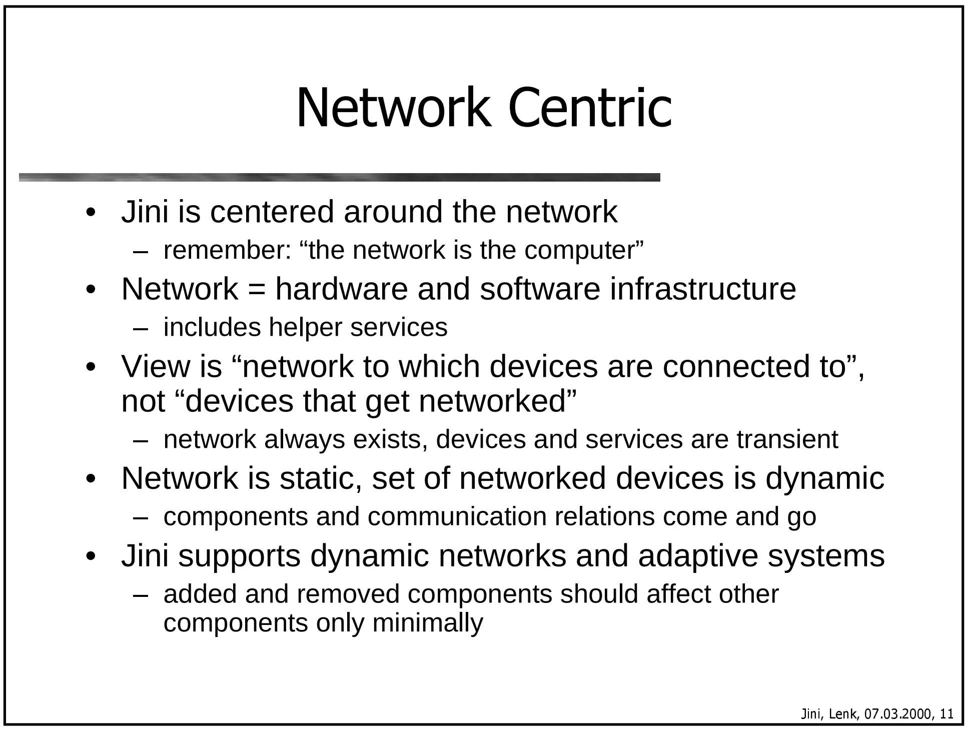
Jini, Lenk, 07.03.2000, 9



A Jini Federation

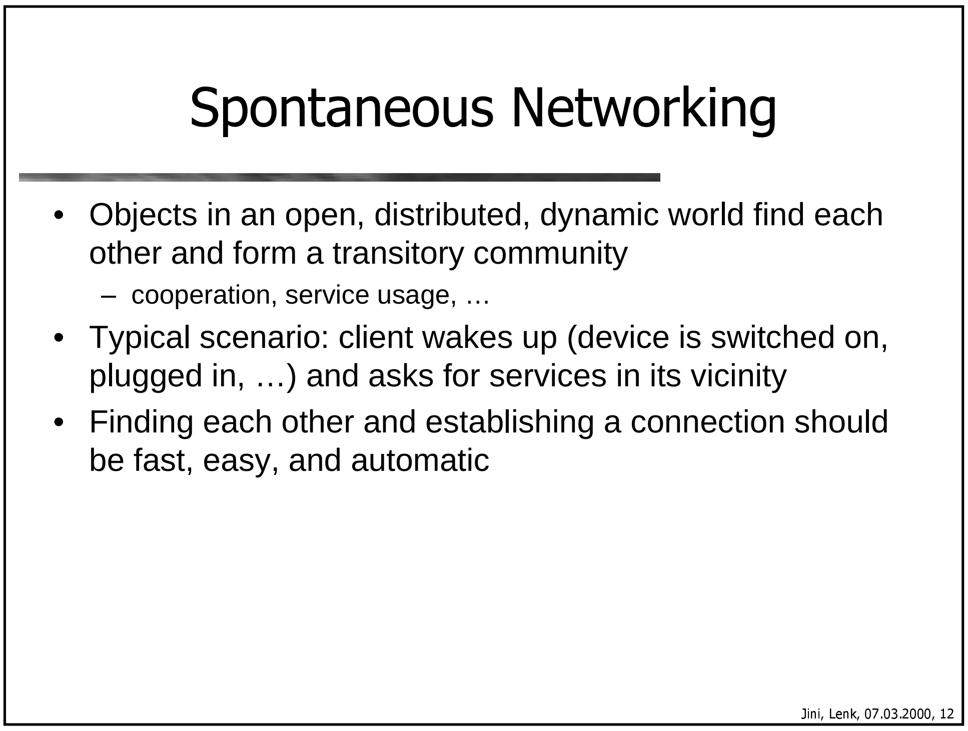


Jini, Lenk, 07.03.2000, 10



Network Centric

- Jini is centered around the network
  - remember: “the network is the computer”
- Network = hardware and software infrastructure
  - includes helper services
- View is “network to which devices are connected to”, not “devices that get networked”
  - network always exists, devices and services are transient
- Network is static, set of networked devices is dynamic
  - components and communication relations come and go
- Jini supports dynamic networks and adaptive systems
  - added and removed components should affect other components only minimally



Spontaneous Networking

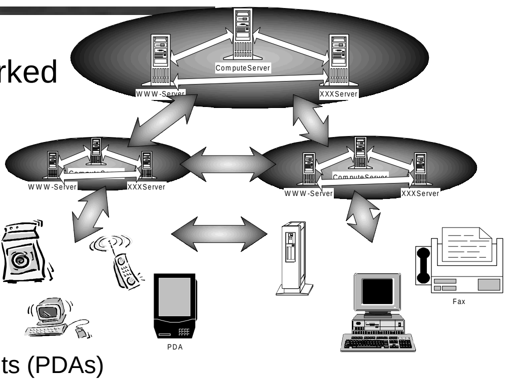
- Objects in an open, distributed, dynamic world find each other and form a transitory community
  - cooperation, service usage, …
- Typical scenario: client wakes up (device is switched on, plugged in, …) and asks for services in its vicinity
- Finding each other and establishing a connection should be fast, easy, and automatic

Why Jini?

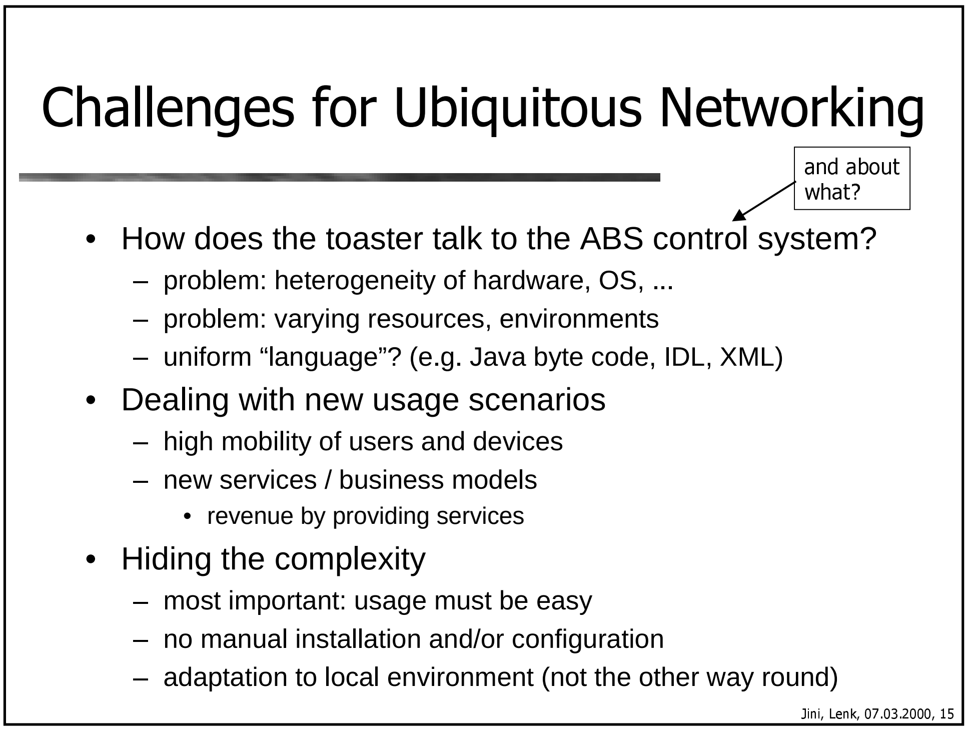
- Infrastructure for tomorrow's vision of ubiquitous computing
  - increasing number of internet users
  - powerful PDAs and notebooks
  - increasing mobility
  - new wireless information devices:
  - numerous processors in embedded systems
    - e.g. software updates for your washing machine, internet-ready microwave, ...
- Numerous mobile networked devices
- Trend towards ubiquitous networks and spontaneous networking / service access
  - high bandwidth, wireless, cheap

The Jini Domain

- Everything will be networked
  - server
    - web server
    - compute server
    - accounting server
  - desktop computer
  - mobile devices
    - notebooks
    - Personal Digital Assistants (PDAs)
    - SmartPhones
  - at home: washing machine, toys, ...
  - embedded systems
  - everyday things ("smart x")



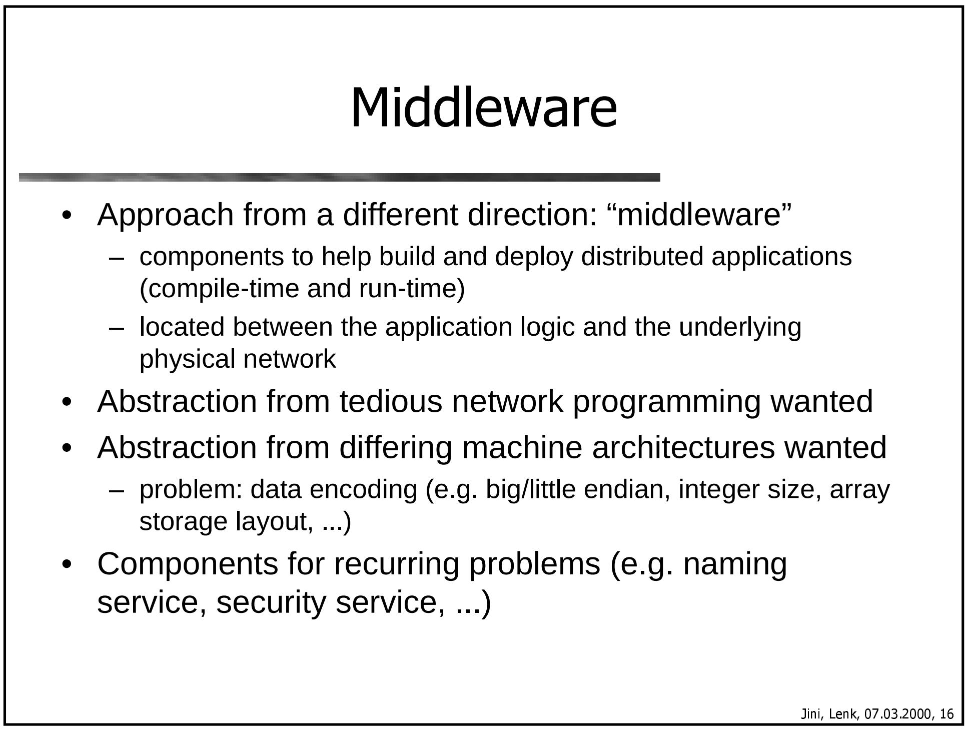
...and everything wants to communicate!



Challenges for Ubiquitous Networking

and about what?

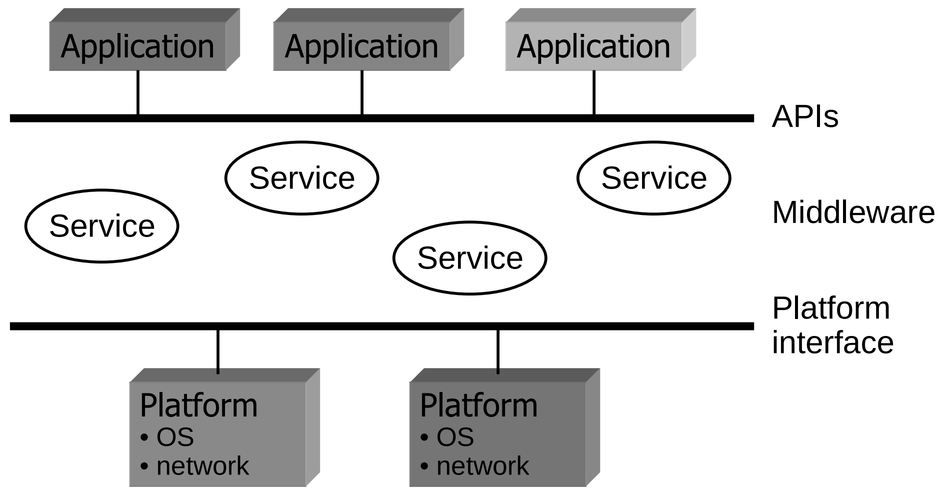
- How does the toaster talk to the ABS control system?
  - problem: heterogeneity of hardware, OS, ...
  - problem: varying resources, environments
  - uniform “language”? (e.g. Java byte code, IDL, XML)
- Dealing with new usage scenarios
  - high mobility of users and devices
  - new services / business models
    - revenue by providing services
- Hiding the complexity
  - most important: usage must be easy
  - no manual installation and/or configuration
  - adaptation to local environment (not the other way round)



Middleware

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

Middleware



Evolution of Middleware

- Low-level protocols
  - exchange of plain messages, e.g. HTTP, X.25, ASN.1, ...
- RPC
  - communication often follows a typical pattern
  - automatic generation of communication code
  - procedural paradigm
- CORBA
  - object-oriented paradigm
  - stubs: automatically generated local proxies that handle communication
- RMI
  - Java’s native middleware
  - code shipping

Problems with Current Middleware

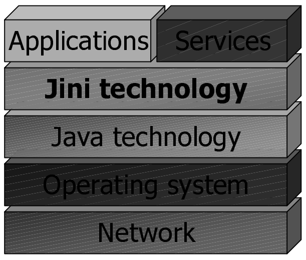
- Systems hide the network from the programmer
  - programmers don't have to deal with the network and its inherent problems (unreliability, latency, bandwidth, ...)
  - no exception handling
- Data is moved to the computation
  - "classical" client/server paradigm
  - not always most efficient solution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

Some Fallacies of Common Distributed Computing Systems

- The idealistic view…
  - the network is reliable
  - latency is zero
  - bandwidth is infinite
  - the network is secure
  - topology doesn't change
  - there is one administrator
- …isn't true in reality
  - Jini addresses some of 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)
  - Jini is a kind of "middleware"
- Extension of Java in three dimensions:
  - infrastructure
  - programming model
  - services



Jini's Java Extensions

- Extensions regarding networked systems

	Programming model	Infrastructure	Services
Java	- Swing - Beans - ...	- JVM - RMI - ...	- Transaction Service - Enterprise JavaBeans - ...
Jini	- Transactions - Distributed events - Leasing	- Discovery - Lookup	- JavaSpaces - ...

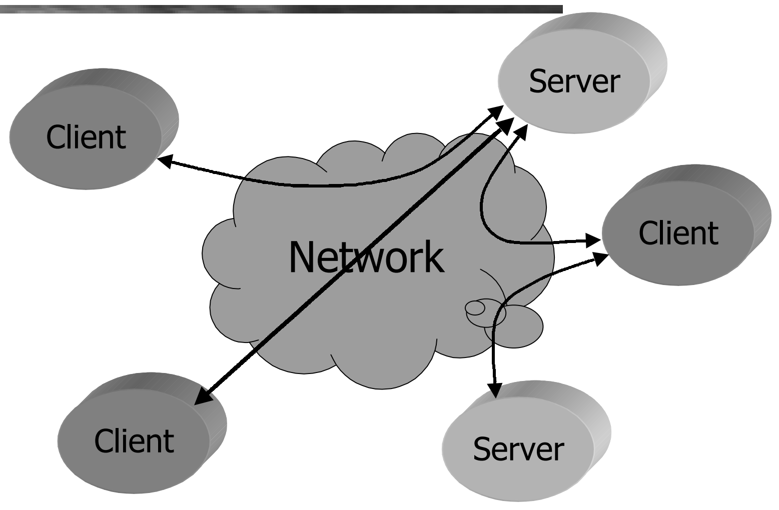
will be explained soon

Jini's Use of Java

- Jini requires JVM (as bytecode interpreter) and RMI
  - homogeneity in a heterogeneous world
  - is this realistic?
- But: devices that are not "Jini-enabled" or that do not have a JVM, 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, checks for array bounds, sand box, ...

Able to perform protocol for discovery and join; have a JVM; ...

Remote Method Invocation (RMI)



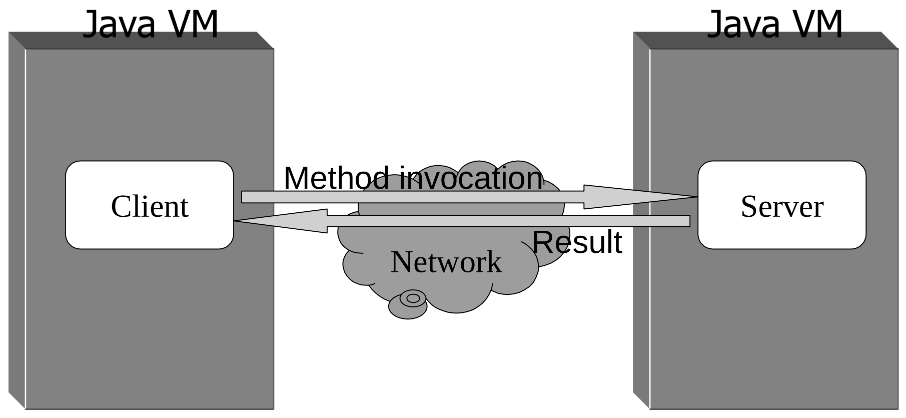
Overview

- Jini, what's that?
  - motivation
  - overview
- RMI
  - introduction
  - example
  - serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions and the transaction manager
  - 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 machines in a network

Remote Invocation



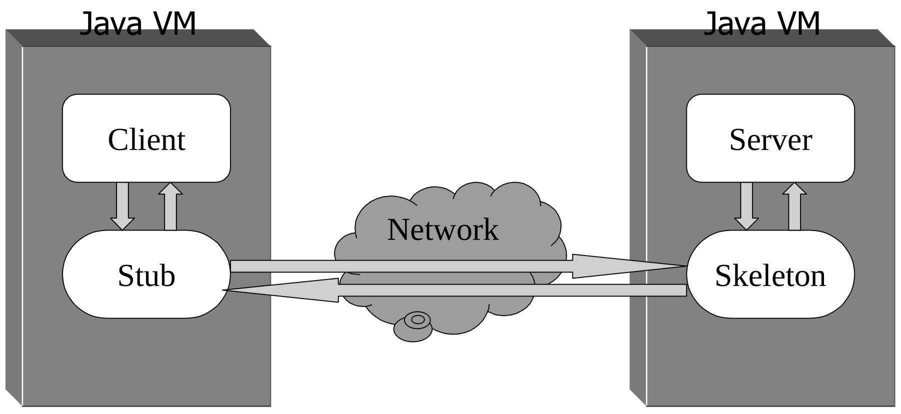
What Can You Do With RMI?

- 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

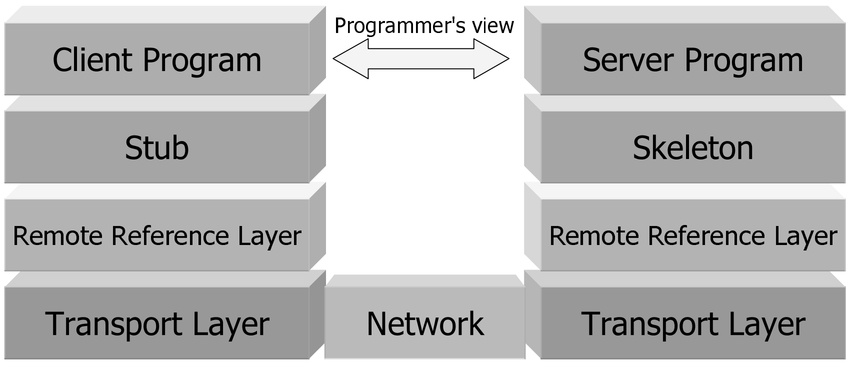
How Does RMI Work?

- 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”
  - signature of methods is identical to methods in remote server object
  - handles communication with remote object
  - “skeleton” in remote JVM is counterpart to stub
    - forwards parameter to “real” server object
    - returns result to stub

How Does RMI Work?



More Details



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

How to Get 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.tu-darmstadt.de:2222/HelloServer`"
  - usually supplies reference to first object only; all further objects handled by this one ("factory pattern")
  - e.g. root: database object, gives access to entries
- Jini

RMI Example: Remote Interface

- Remote methods are defined by "remote interfaces":

```
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`

RMI Example: Server Object

- Server implements remote interface and usually extends `UnicastRemoteObject`

```
public class HelloImpl extends java.rmi.server.UnicastRemoteObject
implements Hello {
  public HelloImpl() throws java.rmi.RemoteException {
    super();
  }
  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

RMI Example: Server Startup

```
public class HelloStart {
  public static void main(String args[]) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new java.rmi.RMISecurityManager());
    }
    try {
      HelloImpl obj = new HelloImpl();
      java.rmi.Naming.rebind(“//server.tud.de/HelloServer”, obj);
    } catch (Exception e) {}
  }
}
```

- Appropriate security manager needs to be set to allow remote access
- Server object is created and registered with RMI registry

RMI Example: Client

```
public class HelloClient {
  public static void main(String args[]) {
    if (System.getSecurityManager() == null) {
      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

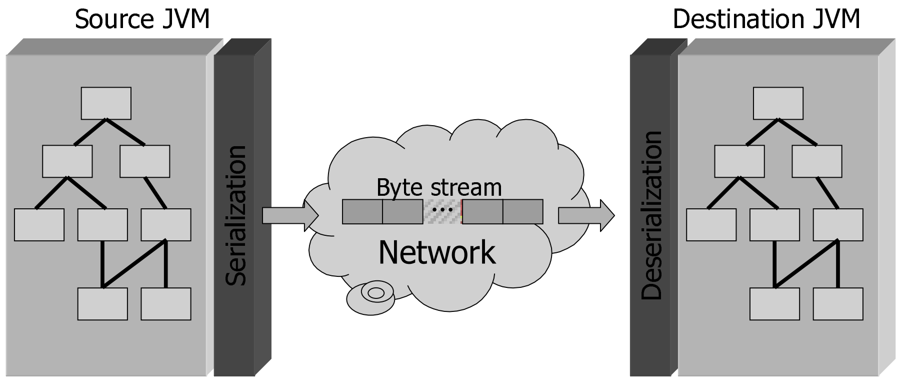
RMI Example: Deployment

- Naming Service must be running: `rmiregistry`
- Stub has to be created: `rmic HelloImpl`
  - creates classes `HelloImpl_Stub` and `HelloImpl_Skel`
  - no skeletons needed in Java 2 (option `-v1.2`)
- 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 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
- Transform it into stream of bytes (“serialization”)
- Problem: objects may contain references to other objects
  - not only simple data types like integer, boolean, ...
  - all referenced objects have to be recursively serialized (“transitive closure” of objects)

Serialization



Serialization

- Transforming an object into a stream of bytes
  - object consists of its code and its state (values of variables)
  - code is stored in class files
  - state is dynamic and only available at runtime
  - take a “snapshot” of this state

Serialization - How Does It 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
- Objects are only written once, even if referenced multiple times
- Serial version UID to control versions
  - calculated from the signature of a class
  - override manually to mark compatible class evolution

Serialization - How Does It Work?

- Objects have to implement interface `java.io.Serializable`
  - “tag” interface: does not contain any functions
  - not implementing this results in runtime error `java.io.NotSerializableException`
- Predefined serialization semantics is defined in `java.io.ObjectOutputStream.defaultWriteObject()`, `java.io.ObjectInputStream.defaultReadObject()`
- 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
- “Low level” classes:
  - input and output streams
  - threads
  - peer classes

Serialization - What To Use It For?

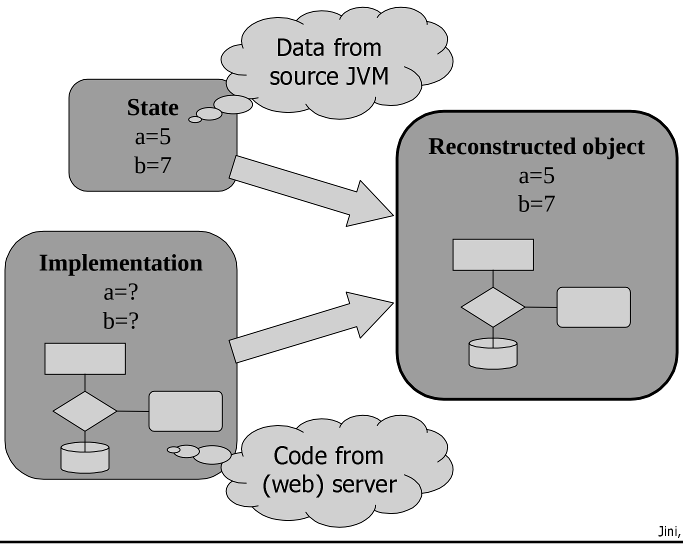
- Write to file: make objects persistent
  - Configurations
  - (Enterprise) Java Beans
  - to continue working on the same state later
- Write to socket:
  - to transfer objects to another JVM
  - parameter passing
  - mobile agents

Code Mobility

- Objects consists of two parts:
  - code (in Java class files)
  - state (values of attributes, execution pointers)
- RMI transfers only 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)

Jini, Lenk, 07.03.2000, 47

Code Mobility



Jini, Lenk, 07.03.2000, 48

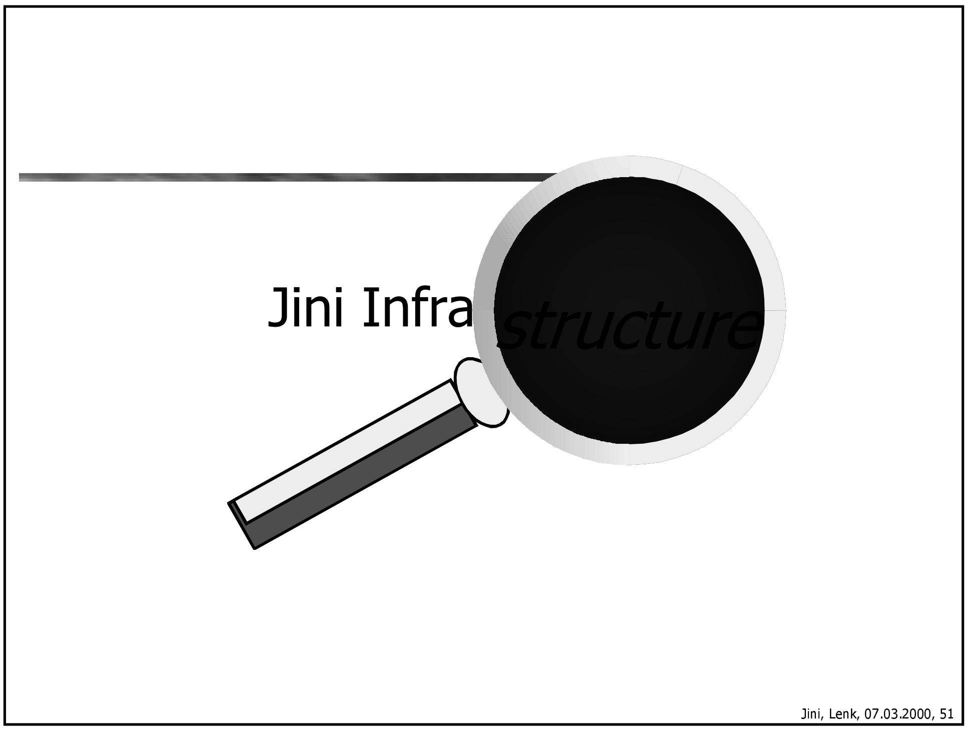
Codebase

- Code can be downloaded, but: where from?
- Location of code is transferred together with its state (“codebase”)
- Codebase is a list of URLs
- Codebase is set as a property when starting a JVM (“`-Djava.rmi.server.codebase=<URL>`”)
- URL might be
  - directory containing the class tree
  - JAR file containing the classes

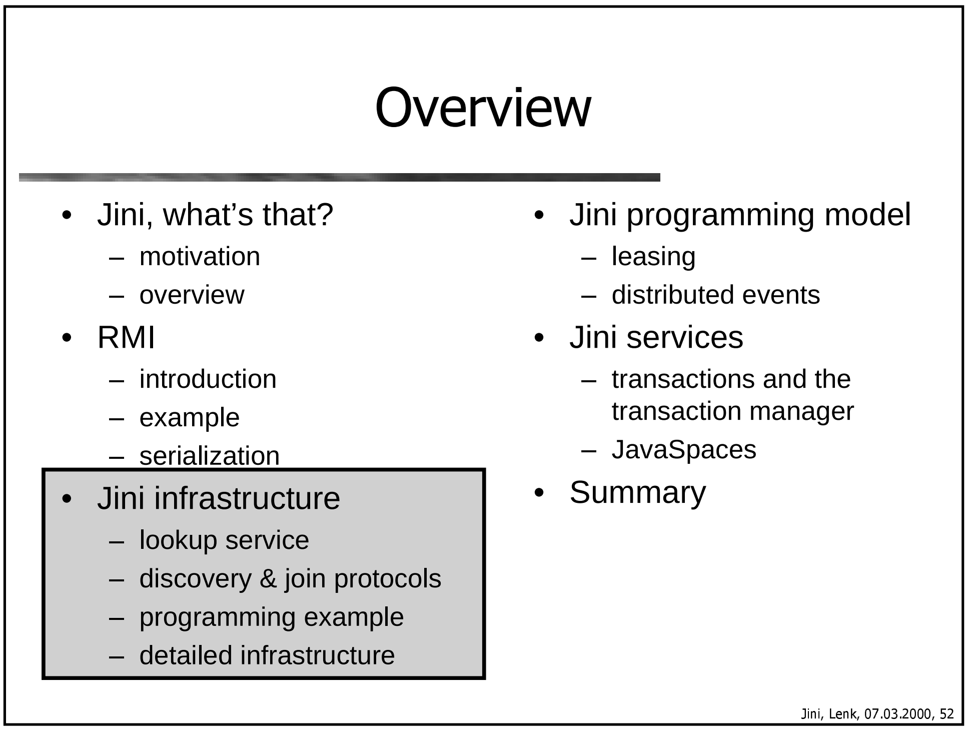
Security

- Unknown objects are executed on local machine (like applets)
- Access restrictions desired
- Access restrictions 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 system)
    - who signed code

```
grant signedBy “sysadmin”, codeBase “http://server.tud.de/-” {
  permission java.net.SocketPermission “*:1024-”, “connect,accept”;
};
```



Jini Infrastructure



Overview

- Jini, what's that?
  - motivation
  - overview
- RMI
  - introduction
  - example
  - serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions and the transaction manager
  - JavaSpaces
- Summary

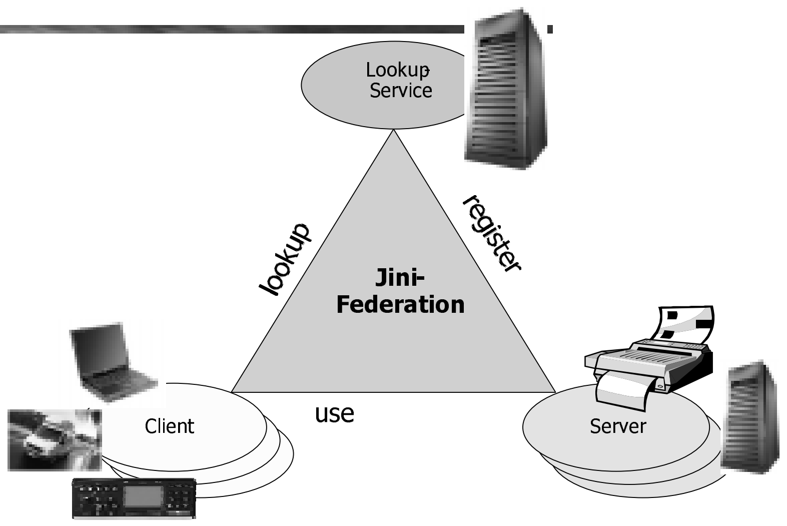
Jini Infrastructure

- Main components are:
  - lookup service as repository / naming service / trader
  - protocols based on TCP/UDP/IP
    - discovery & join, lookup of services
  - proxy objects
    - transferred from service to clients
    - represent the service locally at the client
- Goal: spontaneous networking and formation of federations without prior knowledge of local network environment
- Problem: How do service providers and users get to know their local environments?

Lookup Service (LUS)

- Similar to RMI registry and CORBA naming service
- Main component of every Jini federation
- Repository of service providers
- May be redundant and hierarchically organized (similar to “Domain Name Service”)
- Tasks:
  - “help-desk” for services and clients
    - registration of services (services advertise themselves)
    - distribution of services (clients find services)
  - offers mechanisms to bring together services and clients

Lookup Service



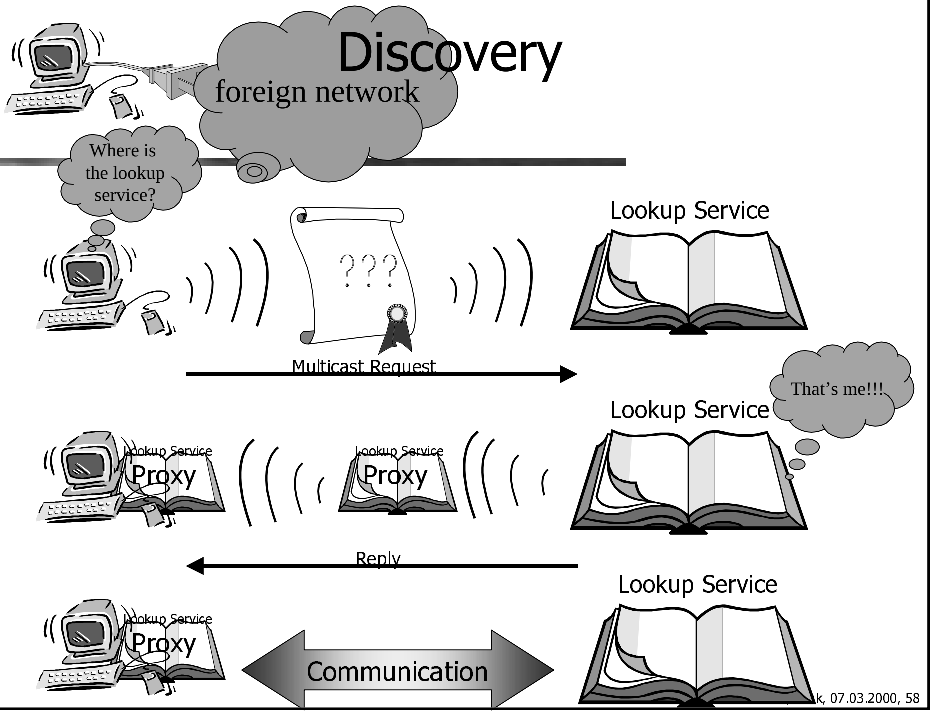
Lookup Service

- Uses Java RMI for communication
  - objects can migrate through the net
- Not only name/address of a service are stored (as in traditional naming services), but also
  - set of attributes
    - e.g.: printer (color: true, dpi: 600, ...)
  - proxies and attributes may be complex classes
    - e.g.: user interface(s)

Discovery: Finding a LUS

- Goal: Find a lookup service without knowing anything about the network to
  - advertise (register) a service
  - 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 (may implement any (proprietary) protocol)

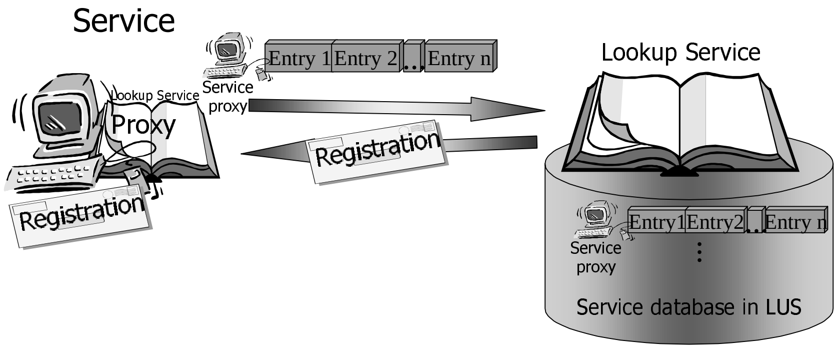
Discovery



Join: Registering a Service

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

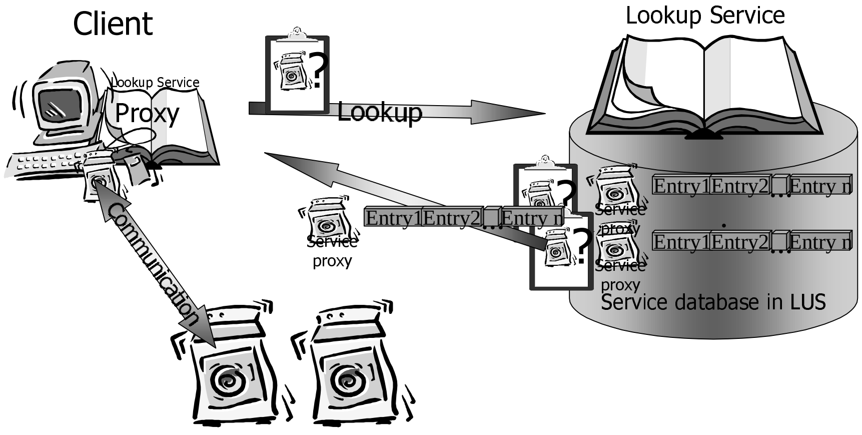
Join



Lookup: Searching Services

- Client knows lookup service (e.g. via discovery protocol)
- Looking for certain service
- Creates query for lookup service
  - in form of a “service template”
  - matching by registration number of service and/or service type and/or attributes
  - wildcards possible
- Lookup service returns one or more matches
- Selection usually done by client
- Service use by calling functions of service proxy
- Any protocol between proxy and service provider possible

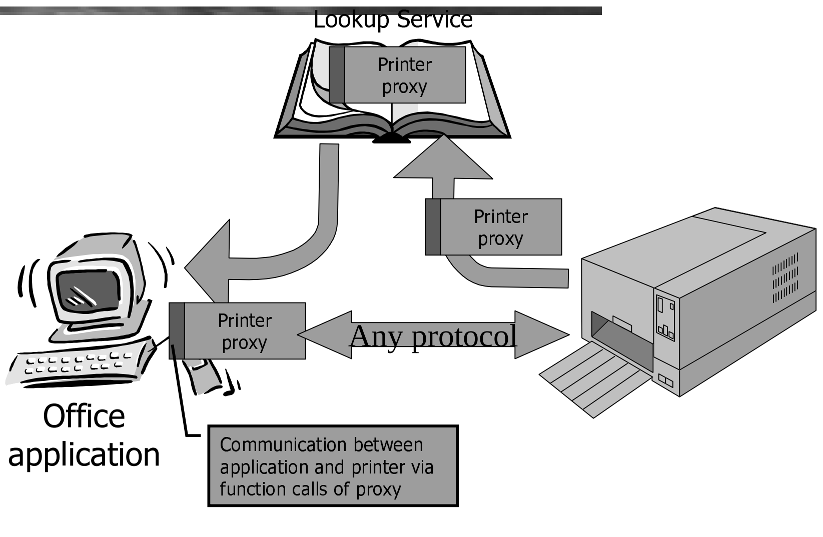
Lookup



Jini Programming Example

- How is a service realized?
- How does a client get access to a service?
- How is a service described?

The "Running-Example"



The “Running-Example”

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

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

  - 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

Implementing a Service Provider

```
public class PrintImpl extends UnicastRemoteObject
    implements Print {
[...]
  public PrinterParams readPrinterParams(
        throws RemoteException {
        // something should be done her
  }
  public SuccessCode print(Document doc)
        throws RemoteException {
        // something else should be done here
  }
[...]
}
```

There’s no
Jini in here!

Registering a Service

```
public class PrintRegistration {
 public static void main(String[] args) {
  if (System.getSecurityManager() == null)
      System.setSecurityManager(new RMISecurityManager());
[...]
  Print service = new PrintImpl();
  Entry[] attribute = new Entry[2];
  attribute[0] = new Name(“PrintService”);
  attribute[1] = new ServiceInfo(“Shiny Print Service”,
                     “HyperClear”, “Shiny Inc.”,
                     “2000”, “”, “08/15”);
  JoinManager jmgr = new JoinManager(service, attribute,
                     (ServiceIDListener) new SvcIDListener(),
                     new LeaseRenewalManager());
[...]
 }
}
```

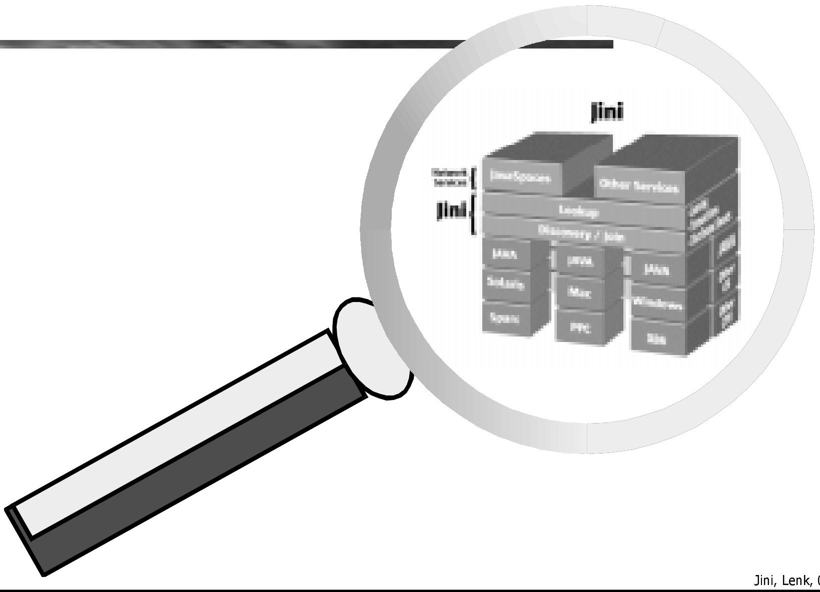
Implementing the Client

```
public class PrintClient {
 public static void main(String[] args) {
  if (System.getSecurityManager() == null)
      System.setSecurityManager(new RMISecurityManager());
[...]
  LookupLocator lus = new LookupLocator(“jini://tud.de/”);
  ServiceRegistrar registrar = lus.getRegistrar();
   // The service we would like to find:
  Class[] cl = new Class[] { Print.class };
  ServiceTemplate template =
                  new ServiceTemplate(null, cl, null);
  Print proxy = (Print) registrar.lookup(template);
  // Use the service
  proxy.print(doc);
[...]
 }
}
```

Multicast Discovery Client

```
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) {}
}
```

Round 2: More Details



Overview

- Jini, what's that?
  - motivation
  - overview
- RMI
  - introduction
  - example
  - serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions and the transaction manager
  - JavaSpaces
- Summary

Overview 2

- Detailed look at
  - discovery
  - join
  - lookup
  - entries
  - lookup service
    - proxies
    - groups
  - leases
  - distributed events
  - transactions
  - transaction manager
  - JavaSpaces

discovery, join, lookup, entries, lookup service (proxies, groups)	Jini infrastructure
leases, distributed events	Jini programming model
transactions, transaction manager, JavaSpaces	Jini services

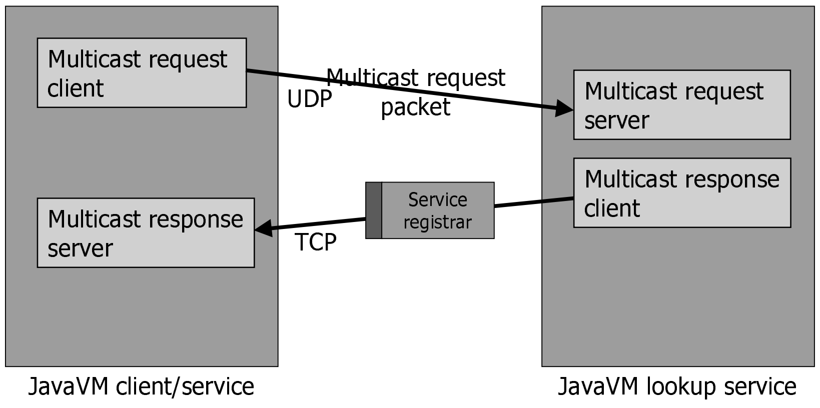
Discovery

- 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
- Multicast announcement protocol
  - protocol for lookup services to announce their presence
- Example: printer registers with the office via multicast, but gets software updates from a dedicated server via unicast discovery

Multicast Request Protocol

- No information about the host network needed
- Active search for lookup services
- Discovery request uses multicast UDP packets
  - IP-centric
  - multicast address for discovery is 224.0.1.85
  - default port number of lookup services is 4160
    - hexadecimal subtraction: $CAFE_{16}$-$BABE_{16}$=4160_{10}
  - 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



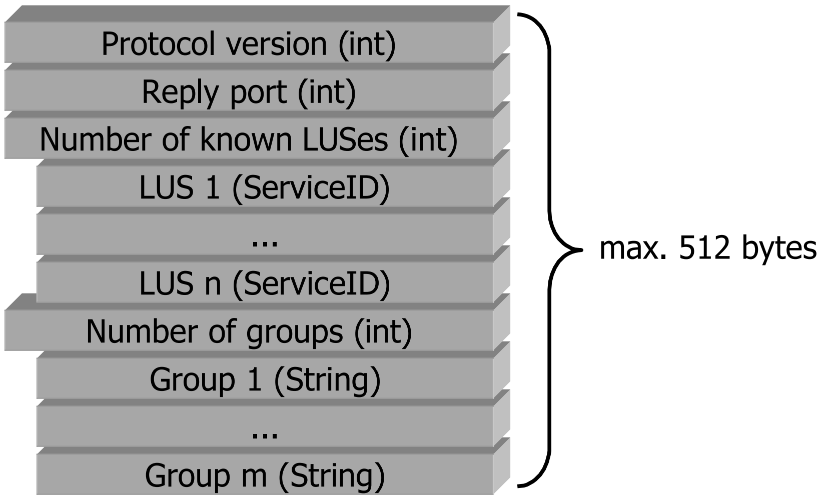
Multicast Request Protocol (Client)

- Initialize TCP server socket (“multicast response server”)
  - wait for incoming TCP connections from responding lookup services
- Initialize a UDP socket (“multicast request client”)
- Periodically send multicast request packets to well-known address/port
  - contains protocol version, contact information, desired groups, and ServiceIDs of known lookup services
  - unlisted lookup services answer by opening a TCP connection
  - multicasting is stopped after some time

Multicast Request Protocol (Server)

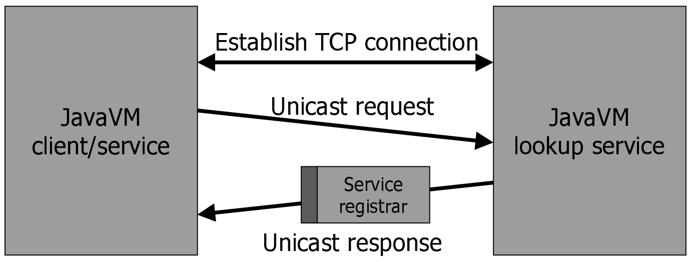
- Initialize a datagram socket on well-known address/port
- Wait for incoming multicast requests
- “Multicast Request Server” answers, if groups match and ServiceID is not listed in list of known lookup services
- Open TCP connection to client and continue with unicast discovery protocol

Multicast Request Packet



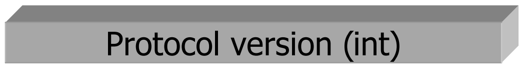
Unicast Discovery Protocol

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

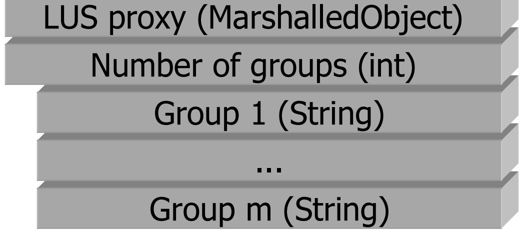


Unicast Packets

- Unicast Request (client → lookup service)



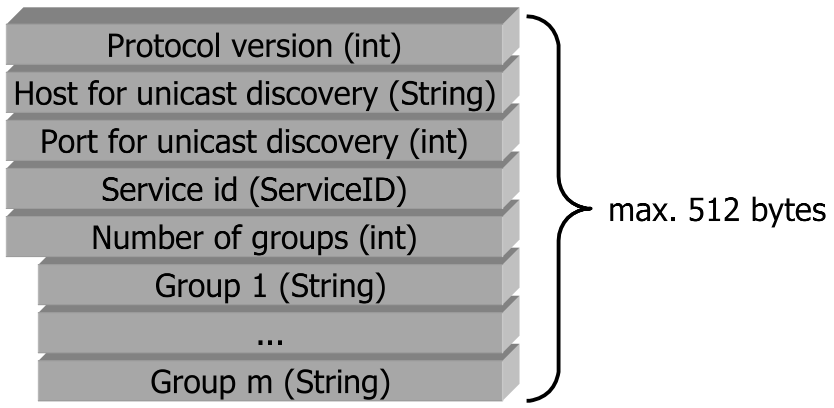
- Unicast Response (lookup service → client)



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 of announcements have to create a “multicast announcement server”
  - listens for announcements on well-known address/port
  - announcements contain protocol version, contact information, groups, and ServiceID of lookup service
  - if not yet known, start unicast discovery of this service

Multicast Announcement Packet



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
  - (possibly empty) set of specific lookup services to join
- 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

- Client looks for service(s) registered with a lookup service
  - any combination of search criteria possible:
    - ServiceID
    - service type
    - certain attributes
  - client creates a `net.jini.core.lookup.ServiceTemplate`

```
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.: printer (dpi: 600, type: color, …)
  - every serializable data type is possible
  - data and methods
  - complex classes possible
    - different user interfaces (AWT, Swing, speech, …)
    - references to further (complex) objects

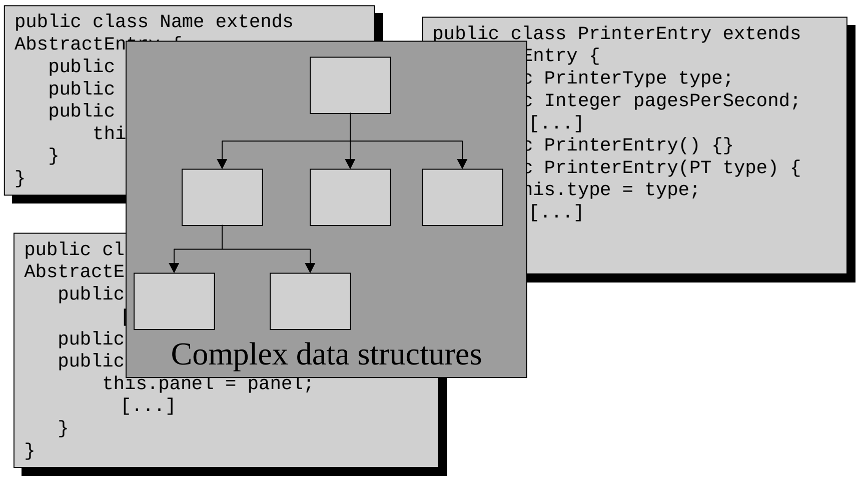
Entries (Examples)

```
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;
        [...]
   }
}
```

Entries (Examples)



```
public class Name extends
AbstractEn
   public
   public
   public
       thi
   }
}
```

```
public class PrinterEntry extends
          Entry {
        c PrinterType type;
        c Integer pagesPerSecond;
          [...]
        c PrinterEntry() {}
        c PrinterEntry(PT type) {
          is.type = type;
          [...]
```

```
public cl
AbstractE
   public
   public
   public
       this.panel = panel;
         [...]
   }
}
```

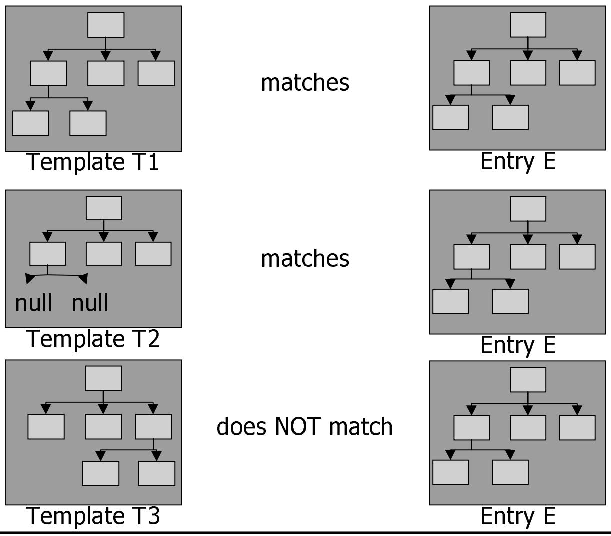
Entries

- Base is interface `net.jini.core.Entry.Entry`
- More useful: class `net.jini.entry.AbstractEntry`
  - implements `Entry`
  - realizes `equals()`, `hashCode()`, and `toString()`
- Entries usually extend class `AbstractEntry`

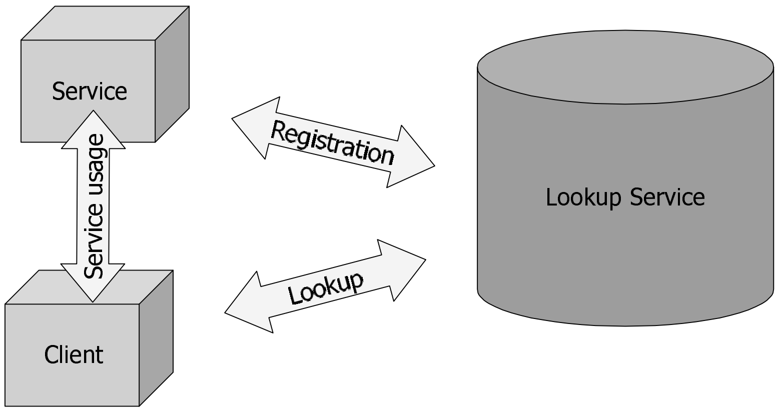
Template Matching (Examples)

- `ServiceTemplate(null, Print.class, null)`
  - matches all services that implement interface Print
  - attributes are ignored (wildcard `null` matches everything)
- `ServiceTemplate(serviceID, null, null)`
  - matches at most one service
- A `ServiceTemplate` filled with entries matches exact data structure and values of entries
  - entry E matches template T if field values are the same
  - wildcards in T match any value in the respective field in E
  - fields in E must have the same type or a subtype of field in T
  - lookup service compares serialized forms of entries and templates

Template Matching (Examples)



Lookup Service Details



Lookup Service

- Jini systems are grouped around one or more lookup service(s)
- Service providers register their services and capabilities with the lookup service (registration)
- Clients find services via the lookup service (lookup)
- Further possibilities:
  - increase robustness by running redundant lookup services
  - responsibility can be distributed to a number of (logically separated) lookup services

Groups

- There may be lots of lookup services in a large Jini system
- Idea: split services into groups and assign responsibility for each of them to a different lookup service
  - so-called “lookup groups”
  - clients/services always announce interest in certain group(s)
  - unwanted groups are ignored
  - simple text identifier
- Example: a company has different lookup services for all departments, e.g. accounting, production, research, ...

Lookup Service Versus “Traditional” Naming Service

Naming service	**Lookup service**
Description by text only • /devices/printers/ → all printers • /devices/printers/color → color printers • /software/wordprocessing/	Description by `ServiceItems` • interface `Printer` • interface `ColorPrinter` • additional information by typed attributes (Name, Location, dpi, etc.)
Lookup by well-known text identifier (convention: print services are in `/devices/printers/`)	Lookup by specifying the (well-known) service type
Reference might have unknown type (fax machine in /devices/printers/)	Reference always has known interface (base or subtype thereof)
Standardized naming conventions	Standardized interfaces
Usually no expiration of entries (heartbeat, keep-alive)	Services have to renew their entries in the lookup service periodically (leasing)
Identified by (static) address; groups can be modeled by addresses	Discovery; group concept

Lookup Service: Proxy Interface

Used by service providers

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

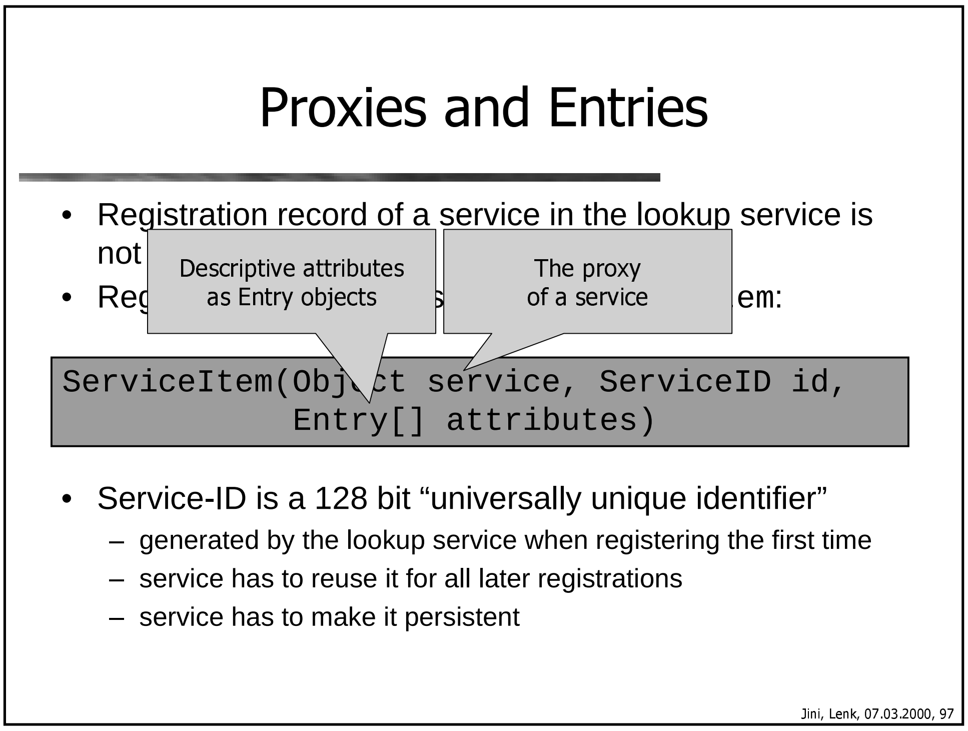
Used by clients

Proxies and Entries

- Registration record of a service in the lookup service is not “just a name”
- Registration record consists of a `ServiceItem`:

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

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



Proxies and Entries

- Registration record of a service in the lookup service is not
- Reg s em:

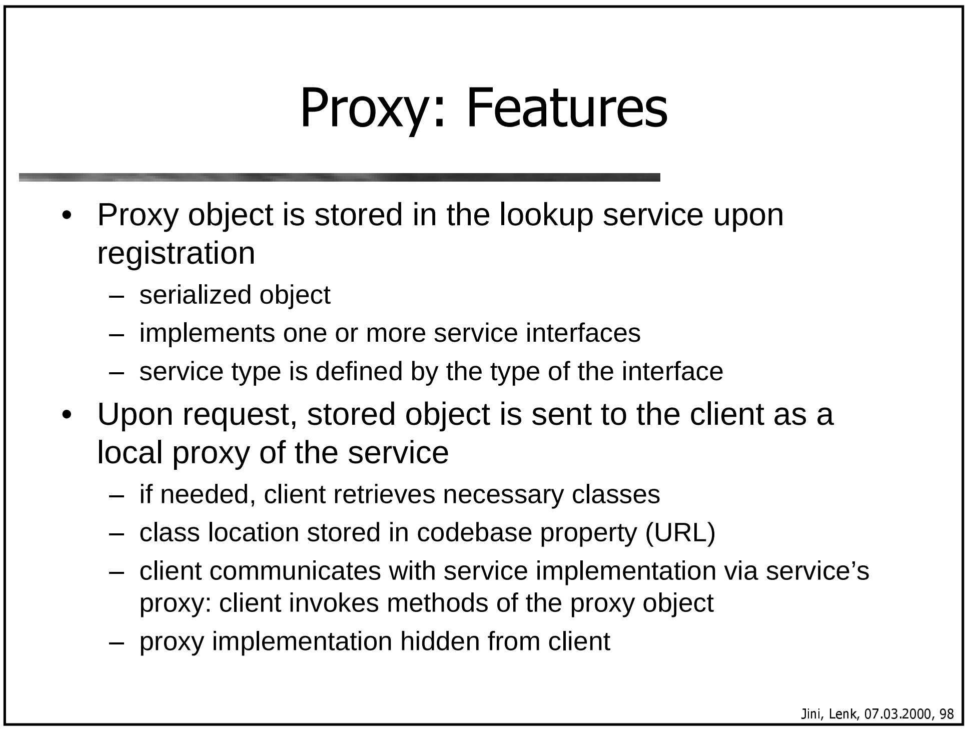
Descriptive attributes as Entry objects

The proxy of a service

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

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

Jini, Lenk, 07.03.2000, 97



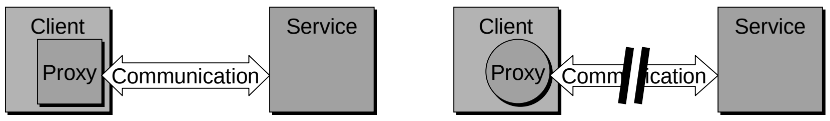
Proxy: Features

- 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
- Upon request, stored object is sent to the client as a local proxy of the service
  - if needed, client retrieves necessary classes
  - class location stored in codebase property (URL)
  - client communicates with service implementation via service's proxy: client invokes methods of the proxy object
  - proxy implementation hidden from client

Jini, Lenk, 07.03.2000, 98

Proxy: Implementation

- Implementation of service functionality is not stipulated
- Partition of service functionality depends on service implementer's choice
- Parts of or whole functionality may be executed by the client (within the proxy)
- 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



Overview

- Jini, what's that?
  - motivation
  - overview
- RMI
  - introduction
  - example
  - serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions and the transaction manager
  - JavaSpaces
- Summary

Leases

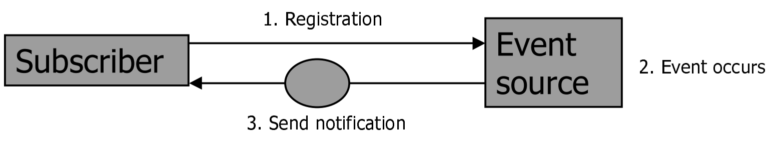
- Leases are contracts between two parties
- Leases introduce the notion of time
  - – resource usage is restricted to a certain time frame
  - – interaction is modeled by repeatedly expressing interest in some resource:
    - I’m still interested in X
      - – renew lease periodically
      - – lease 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 enable intelligent resource allocation
- Leases enable intelligent service removal

Why Leases?

- For allocating hardware and software resources
  - – examples: persistent storage, input/output devices, group communication: participation is leased
- Inside Jini
  - – distributed “garbage collection”
    - registrations with lookup service are leased
    - resource allocations are leased (e.g. transactions)
    - lease expired $\rightarrow$ “garbage”
  - – event notification registrations
- Time-based charging for service use (maybe...)

Distributed Events

- Objects in a JVM can register interest in certain events of another object in a different VM
  - JVMs can be on different machines connected by a network
    - network failure
    - crossing of event notifications
    - late and lost messages
- “publisher/subscriber” model
- Architecture:

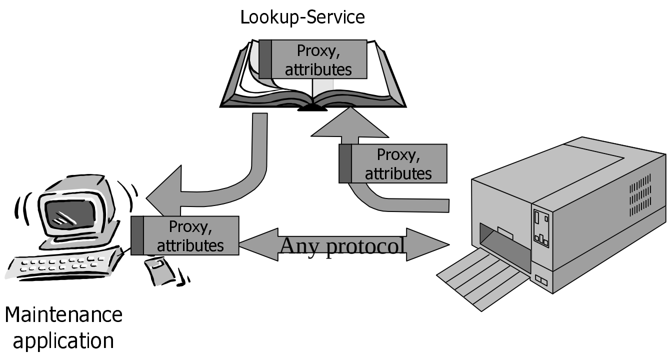


Distributed Events

- Participants
  - event source
    - indicates the occurrence of a certain event by emitting a notification
    - sends notifications to all listeners registered for this event
  - remote event listener
    - object that wants to be notified about a certain kind of events
  - object that registers the remote event listener
    - usually the same as the listener, but not mandatory
    - “store and forward” agent
    - “event mailbox”
  - remote events
    - objects of class `RemoteEvent` (or subclass)

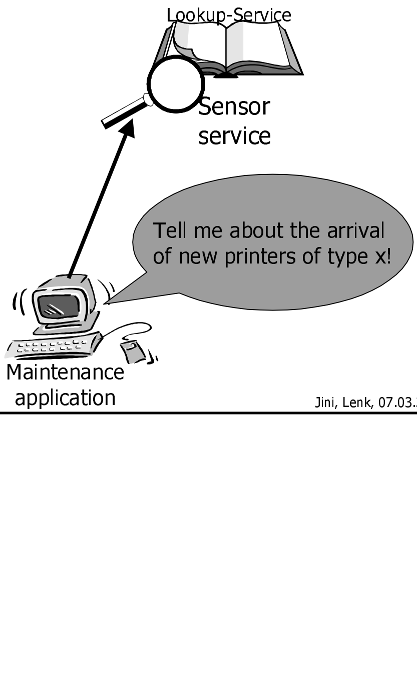
Distributed Events (Example)

- Again: printer was plugged in
  - printer registers itself with local lookup service
- Now: maintenance application wants to update software



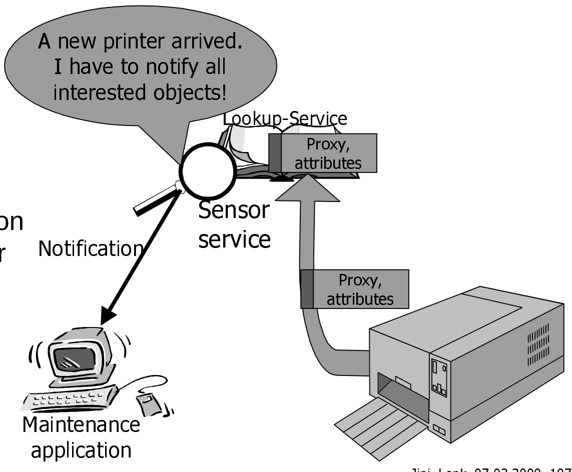
Distributed Events (Example)

- Maintenance application is run on demand, search for printers is “out-sourced”
  - “sensor service” looks for certain services on behalf of the maintenance application
  - application registers for events showing the arrival of certain types of printers
  - sensor observes the lookup service
  - notifies application as soon as matching printer arrives
  - realized by distributed events



Distributed Events (Example)

- Now: printer arrives, registers with lookup service
  - printer performs discovery and join
  - sensor finds new printer in lookup service
  - checks if there is an event registration for this type of printer
  - notifies all interested objects
  - maintenance application retrieves printer proxy and updates software



Distributed Events (Example)

- Realization (interfaces):
  - application implements interface **`RemoteEventListener`**
    - can now receive notifications
    - `notify(RemoteEvent theEvent)` is only method
  - sensor could implement the interface:

```
public interface LUSSensor extends Remote {
  public EventRegistration register(
                    ServiceTemplate theEvent,
                    MarshalledObject handback,
                    RemoteEventListener toInform,
                    long leaseLength)
        throws UnknownEventException, RemoteException;
}
```

Event type to be registered for.

“handback” is returned to the notified object in every notification. It can easily attach arbitrary information to its registration.

Reference to the object to be notified.

Registrations are leased.

Distributed Events (Example)

- Realization (pseudo code):
  - maintenance application looks for sensors and registers for event notifications:

```
[...]
// pseudo code: look for sensors and get p
LUSSensor sensor = doLookup(LUSSensor);
// describes events
Entry[] attributes = new Entry[10];
// [...] initialize array
ServiceTemplate toLookup = new ServiceTemplate(null,
                               classToLookup, attributes);
// event registration
EventRegistration registration =
sensor.register(toLookup, null, this, Lease.FOREVER);
[...]
```

Represents whole Lookup

ServiceTemplate for event description

ServiceTemplate is event type

Handback is not used here

Application implements RemoteEventListener

Lease is unlimited

Distributed Events (Example)

- Realization (pseudo code):
  - LUSSensor informs maintenance application about new printer

```
[...]
// LUSSensor found matching service in lookup service,
// Inform the application
toInform.notify(new RemoteEvent(this, eventID, seqNum, null));
[...]
```

Handback

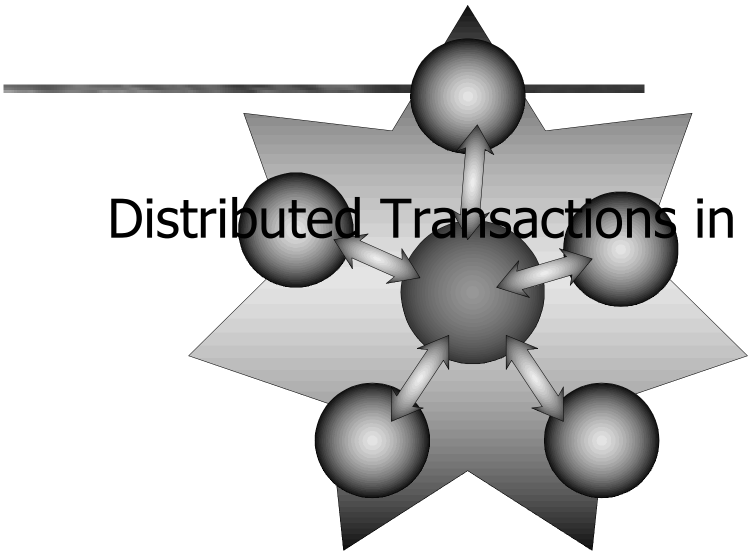
Recipient

Event source

Event

Sequence number

  - event source informs the recipient of an event by sending a RemoteEvent
  - RemoveEvent identifies event unambiguously by tuple <event source, event id>
  - notification is synchronous
    - recipient has to accept event, store it, and return from notify()-method
  - maintenance application looks up printer service



Distributed Transactions in Jini

Overview

- Jini, what's that?
  - motivation
  - overview
- RMI
  - introduction
  - example
  - serialization
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Jini services
  - transactions and the transaction manager
  - JavaSpaces
- Summary

Transactions: Properties

- No “traditional” transaction model
- No enforced semantics
  - participants implement semantics
  - system only provides synchronization mechanism
  - system distributes information about status of transactions
- Design goal: maximum flexibility, minimum number of interfaces

Two-Phase Commit Protocol

- Transaction encapsulates a number of operations
- Central: a manager
  - consistency: each transaction participant will ultimately perform a “commit” or an “abort”
- Every object can participate in a transaction:
  - implements interfaces
  - not only traditional applications possible
- Example: transfer money from one account to another

Distributed Transactions in Jini...

- …are no transactions in a traditional sense
- “Lightweight” transaction
- ACID properties
  - atomicity / consistency / isolation / durability
  - each participant implements these properties how he sees fit
  - reason: two phase commit protocol not only in traditional transaction context
    - e.g.: transient objects do not need persistency
    - main property is atomicity
    - the other properties are “sometimes” optional
- Transactions are leased from the manager

Transactions: Participants

- Transaction manager
  - Jini service
  - coordinates transaction
  - implements interface `TransactionManager`
- Clients
  - initiate transactions
- Transaction participants
  - have active role in transactions
  - implement interface `TransactionParticipant`
  - participation is shown by `join` operation

Transactions: Semantic Objects

- Transactions have no a priori semantics
  - class `Transaction` tells objects to use their standard transaction semantics
- Participants need the same “view” on the transaction
  ➔ semantic objects
  - encapsulate transaction ID
  - their type defines the kind of transaction
- Services only accept known transaction types
- Example: DBTransaction
  - requires database semantics
  - transient objects not allowed (durability)

Two-Phase Commit Protocol: Details

- Start of a transaction
  - get transaction manager from lookup service
  - implements interface `TransactionManager`
  - call to `create()` starts transaction
    - usually indirectly via semantic object factory
- Participation in a transaction
  - participants find out about transaction upon first invocation
    - has to report to the transaction manager
    - calls its `join()` method
  - implements interface `TransactionParticipant`

Two-Phase Commit Protocol: Details

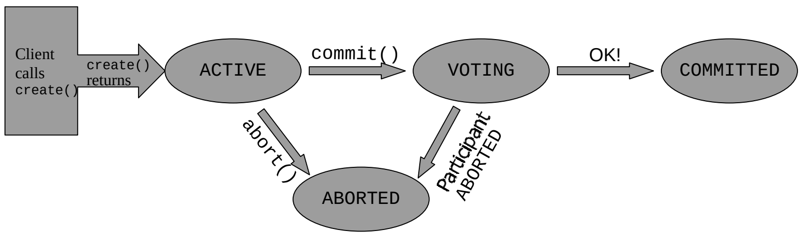
- St
  –
  –
  –

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

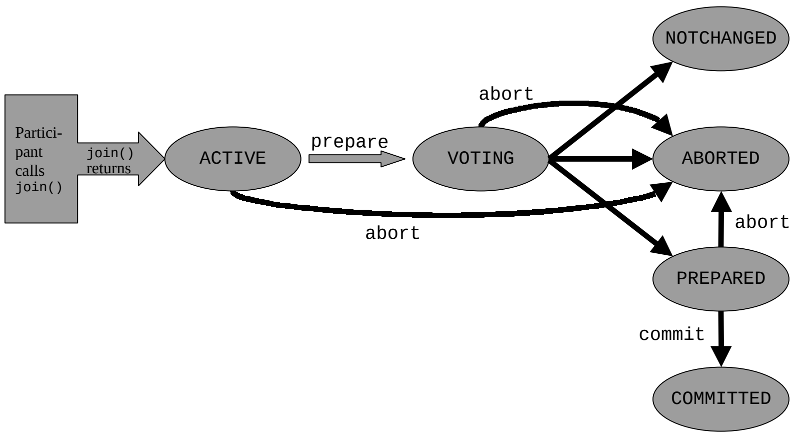
- Participation in a transaction

```
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)
}
```

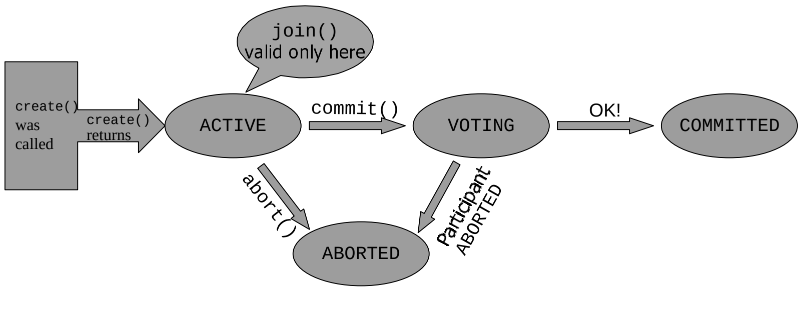
Transactions: Example (Client)

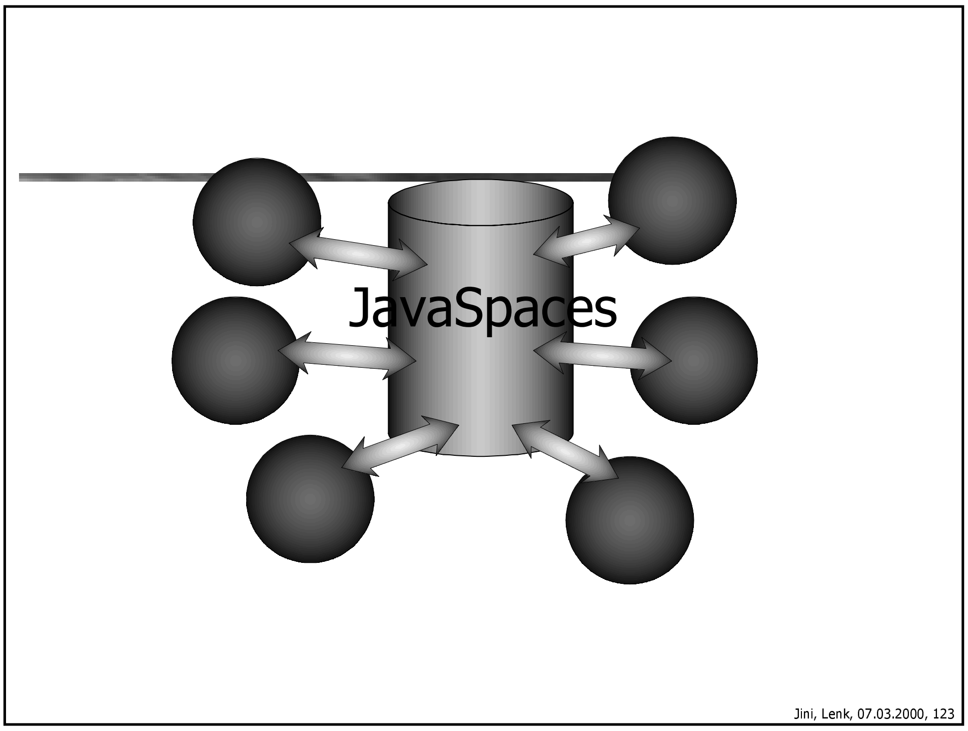


Transactions: Example (Participant)



Transactions: Example (Manager)





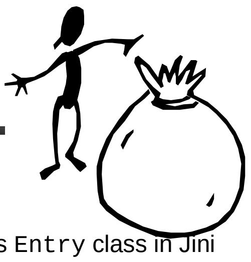
What are JavaSpaces?

- Tool for developing distributed applications
- Platform for exchanging objects between distributed applications (“shared blackboard”)
- Realizes distributed persistency
- Jini service
  - implemented completely in Java
  - uses RMI (in current implementation)
  - client gets local proxy from service (via lookup service)

What for?

- Models “object flow”
  - e.g. producer / consumer applications
- Job-oriented view
  - jobs / events are put into the space and picked up “eventually”
- Build-in “good” properties
  - “reliable storage system”
  - concurrent access possible
  - write / read are atomic operations
  - access within transaction possible

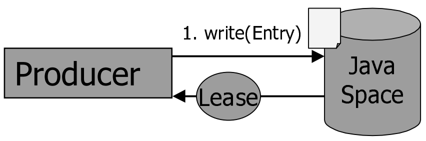
Objects



- JavaSpaces = “a bag full of objects”
  - entries in a JavaSpace service:
    - `net.jini.core.entry.Entry` - known as `Entry` class in Jini
    - strongly typed by Java type system
    - two entries are not equal, even if they encapsulate the same data types
      - Entry A {Integer, Integer} ≠ Entry B {Integer, Integer}
  - classes: include data/state and methods → behavior
  - may become active on the client side (security?)

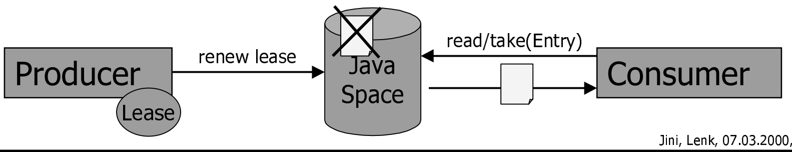
Basic Operations: Write

- `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
  - – easier implementation?
    - recovery after failure of producer
    - mobility of client



Basic Operations: Read

- `read/readIfExists`
  - – uses template (`Entry`)
    - `null` returns everything
    - matches subclasses as well
  - – exact value match
    - if more than one match, select and return one at random
  - – `readIfExists` returns reference to copy of the entry, `null` if no match
  - – `read` waits until matching entry was found
- `take/takeIfExists` = `read` + removal of entry



Basic Operations: Notify

- Uses Jini distributed events
- Clients can register with JavaSpace services
  - implements `RemoteEventListener` interface
  - registers for certain events by supplying a “matching template”
- JavaSpace service notifies listeners
  - if a matching entry is written to the space, all listeners will be notified (order not specified)
  - registration is leased
  - “first come, first serve” for registered listeners on take

Peculiarities

- All operations can be part of a transaction
  - sometimes necessary
- 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
    - entries may not be lost

JavaSpaces and Linda

- Design based on Linda tuple spaces
  - David Gelernter (Yale University)
- Differences:
  - strong type checking and objects (methods/behavior)
    - matching on whole tuple, not just entries in tuple
    - templates match subclasses as well
    - all (serializable) data types can be used as entries
  - multiple spaces possible
  - leasing
  - no “eval” - a JavaSpace is just a repository

Overview

Jini: Summary

- Vision:
  - everything will be networked
  - everything will (be able to) communicate
  - communication will be cheap (or free)
  - mobility will be important feature
- Problem:
  - infrastructure should adapt to devices, not the other way round
  - spontaneity as paradigm
  - incorporation of small devices
  - distribution
    - partial failure
    - communication via networks

Summary

- Solution (?)
  - Jini as infrastructure for service-oriented, ubiquitous networks
  - RMI for abstracting from the network
  - discovery & join, lookup
  - leases, distributed events, transactions
  - service as the main abstraction
- Challenges for Jini:
  - interfaces
    - standards (e.g. printer, ...)
  - distribution
    - “always in the back of your head”
    - code required for intelligent error handling (network problems, failed/missing services, ...)

Conclusion

- Right direction
  - ubiquitous networks
  - mobility
- A number of good ideas
  - simplicity
  - “less is more” → flexibility
  - discovery & join
  - extension of name services by describing attributes
  - leases, transactions → recurring design patterns
- Individual concepts are not new, but together they form new possibilities (“the whole is more than its parts”)

But...

- Resource usage
  - each service usually requires a JVM
  - JavaSpaces tend to grow quickly
- Performance
  - Java/RMI
- Small devices
  - JVM and RMI required on device
  - adaptation to resource-restricted environment necessary (how?)
  - proxy objects are moved to device (memory...)
- Standardized (base) interfaces
  - allow for type-safe invocation of methods
- What about the competitors (SLP, UPnP, e’’speak, ...)?

Problem Areas

- Security
  - important especially in dynamic environments
  - user requires confidentiality
    - communication
    - data
    - e-commerce
  - services use other services on behalf of the user
    - principals, delegation
    - what about charging for services?
  - Java RMI security extension does not seem to be the solution
- Scalability
  - does Jini scale to a global level?

Hope

- Specialized Jini hardware
  - costs per chip are important (< 10$)
  - performance
- Increasing power of hardware (e.g. PDAs)
  - is it the answer to all problems?
- Concepts for integrating (dumb) devices
  - proxies, device bay, ...
- Open source movement
  - Jini comes with a “half-open” license
  - initiatives of vendors to standardize service interfaces more important (printer, storage, network management, ...)
- Faster Java ;-)

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...)

End

Contact:

Peer Hasselmeyer
Darmstadt University of Technology
peer@ito.tu-darmstadt.de
http://www.ito.tu-darmstadt.de/staff/Peer/

Friedemann Mattern
ETH Zürich
mattern@inf.ethz.ch
http://www.inf.ethz.ch/~mattern