Jini

Gerd Aschemann
Peer Hasselmeyer
Darmstadt University of Technology

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

Jini: "Zero Administration"

- "Plug and Play"
- At home: desired, needed, useful, possible
- At work:
  - small workgroups: mostly like at home
  - large software systems: administration needed
- Service providers: vital

➔ Jini federations need administration

- Jini can be used for "traditional" management as well

Overview

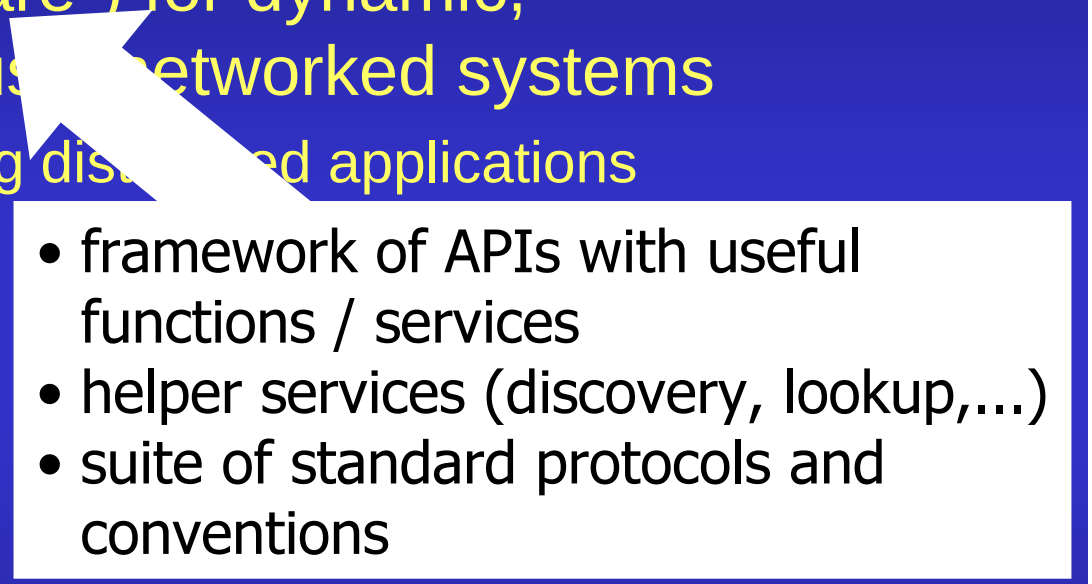
- Jini, what's that?
  - motivation
  - overview
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Summary

Jini

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

Jini

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



Jini

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

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

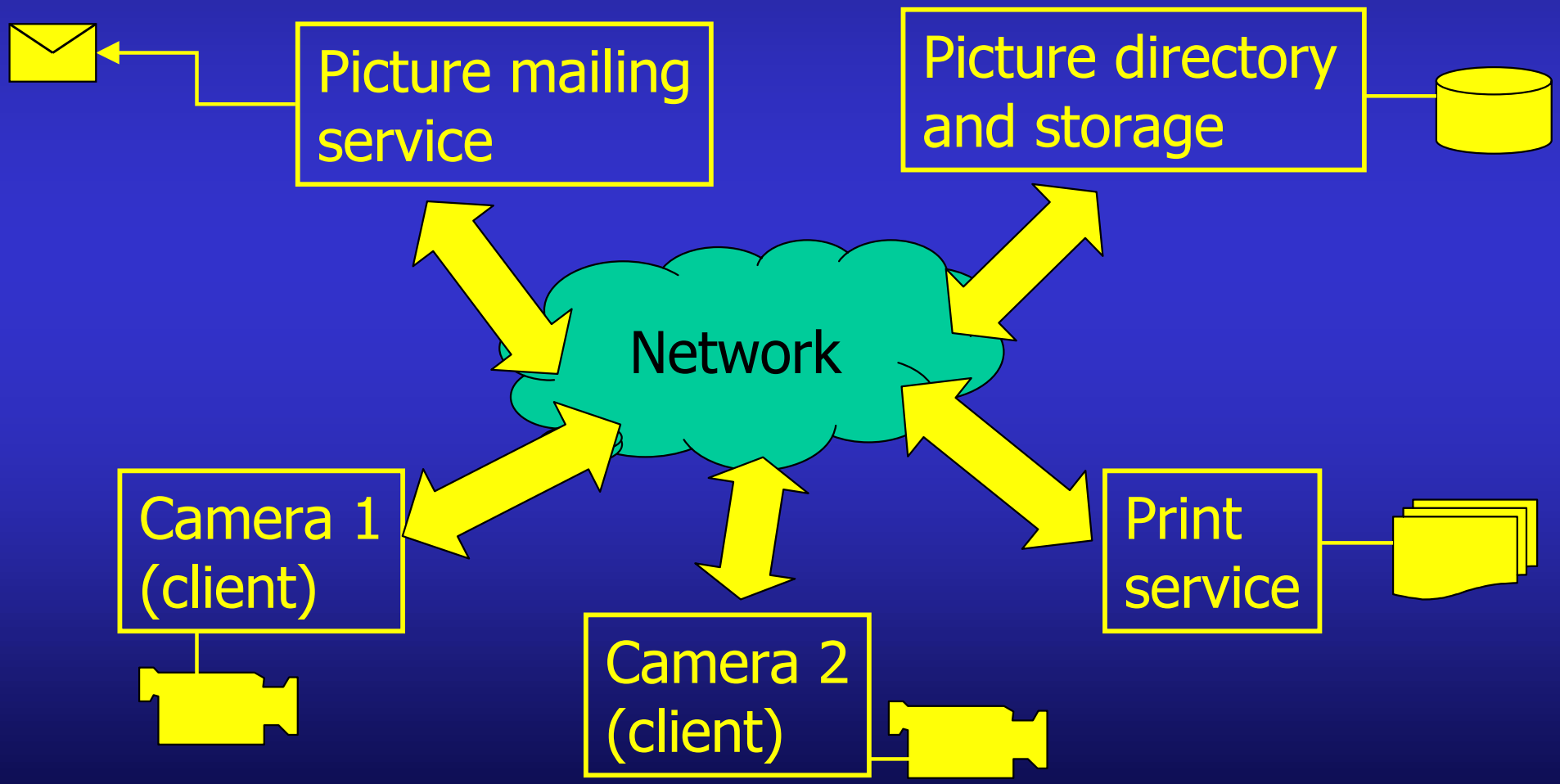
Jini

- Infrastructure (“middleware”) for dynamic, cooperative, spontaneously networked systems
  - facilitates writing / realizing distributed applications
- 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
  - everything is a service (hardware / software / user)
  - Jini system is a federation of services

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”)

A Jini Federation



What Kind of Services?

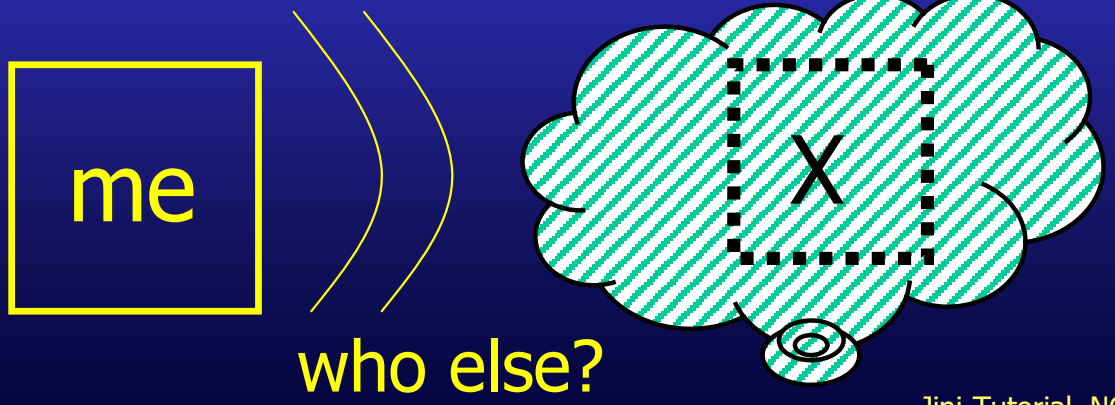
- Devices:
  - printer, fax machine, ...
  - storage, persistency, configuration, ...
  - computation, ...
- Software:
  - spell checking, format conversion, ...
  - online banking, stock trading, ...
  - tourist guide, local maps, hotels, restaurants, ...

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, …
- 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
  - objects must be able to find each other
  - must be fast, easy, and automatic



What Does Tomorrow Look Like?

- 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

Ubiquitous Networks

Bluetooth

It is not just a question of connecting to the other side of the globe.

It is as much a question of connecting to the other side of the room.

- New network technologies, e.g.:
  - Bluetooth / Wireless LAN
  - HAVi
  - Powerline Networking

"We are fast approaching a time when users will find it necessary to be able to connect their digital AV appliances to create home entertainment networks."

- Bandwidth usage of fiber increases rapidly
- Expectations:
  - connectivity will be ubiquitous
  - will be free or really cheap
- Conclusion:
  - "everything" could potentially be connected to "the net"
    - even only temporarily connected things (e.g. contactless smart cards and "smart labels")
  - service orientation (in business models as well)

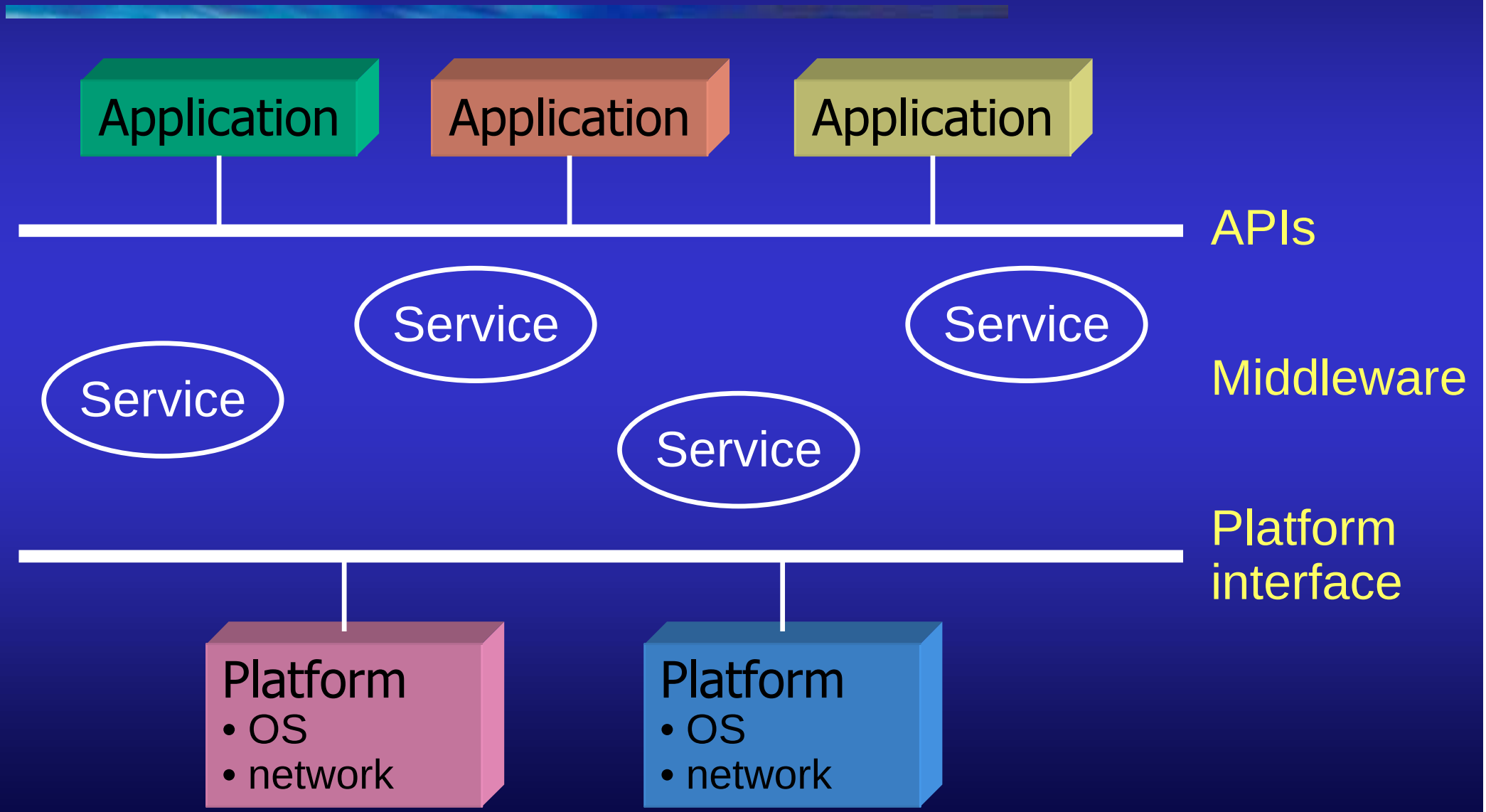
Challenges for Ubiquitous Networking

- How does the toaster talk to the ABS control system?
  - problem: heterogeneity of hardware, OS, ...
  - problem: varying resources, environments
  - uniform “language”? (e.g. Java bytecode, 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

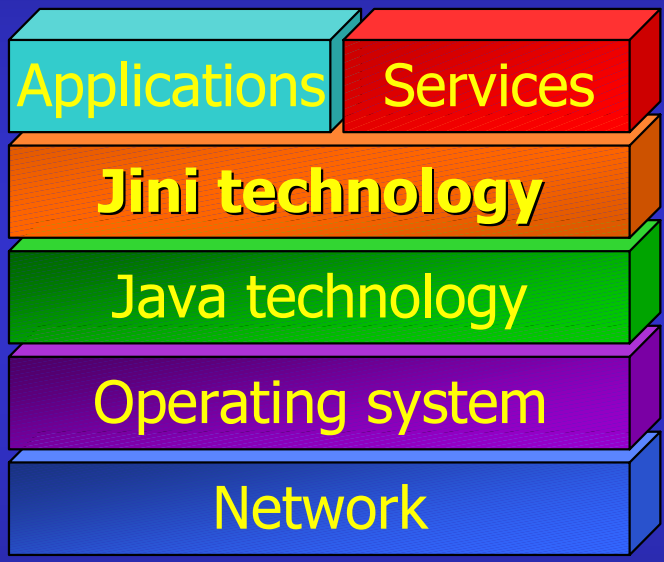

Application
Application
Application
APIs
Service
Service
Service
Service
Middleware
Platform interface
Platform
• OS
• network
Platform
• OS
• network

Middleware: Partial Failure

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

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

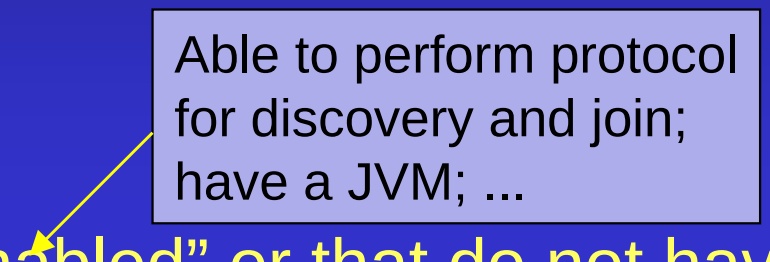
- Extensions regarding networked systems

	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

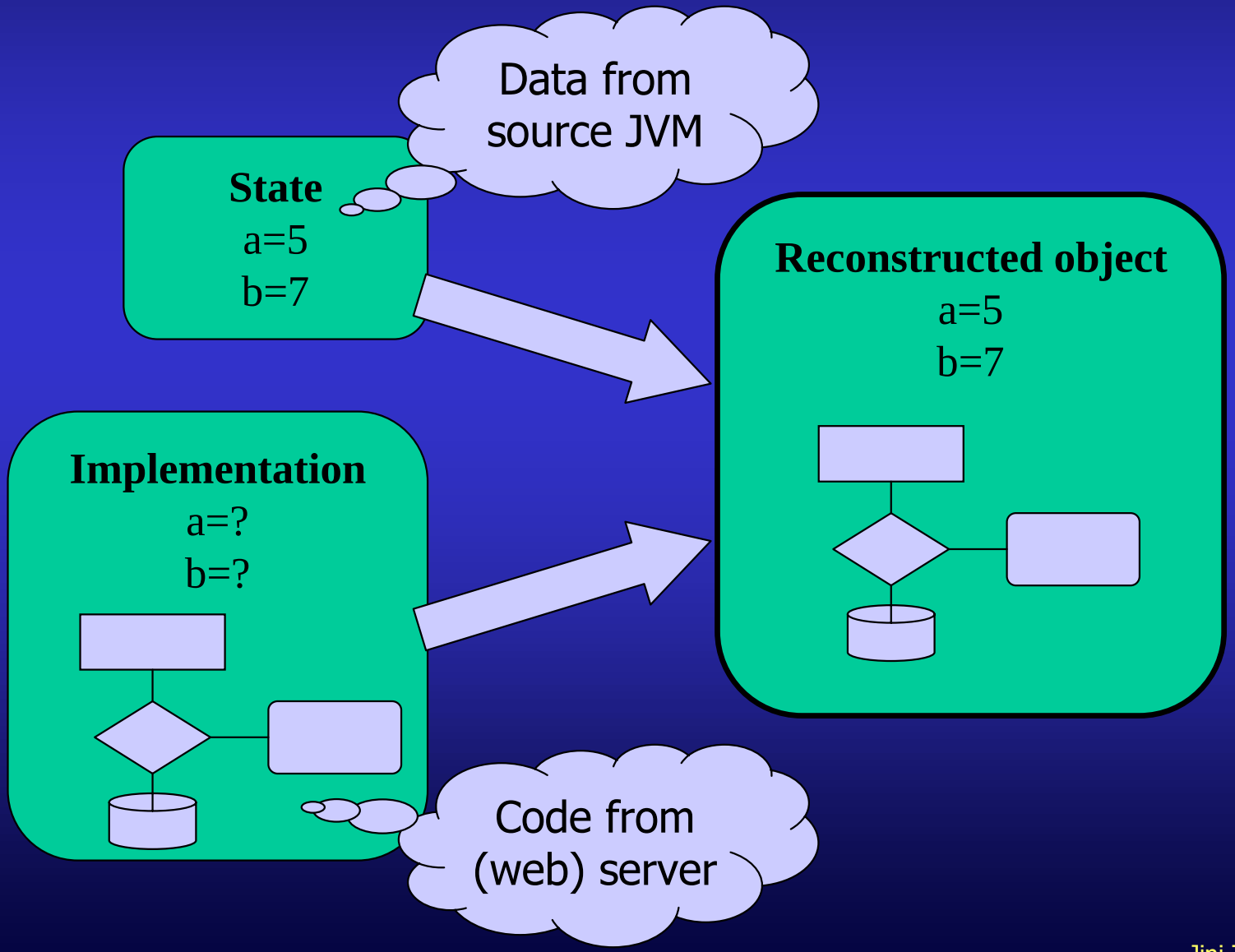
- Jini requires JVM (as bytecode interpreter) and RMI
  - homogeneity in a heterogeneous world
  - is this realistic?
- But: devices that are not "Jini-enabled" or that do not have a JVM, can be managed by a software proxy which resides at some place in the net
  - e.g. "Device Bay"
- Safety of Java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



Code Mobility

- Proxy objects are sent from service provider to client
- Objects consists of two parts:
  - code (in Java class files)
  - state (values of attributes, execution pointers)
- RMI (`MarshalledObject`) 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)

Code Mobility



Codebase

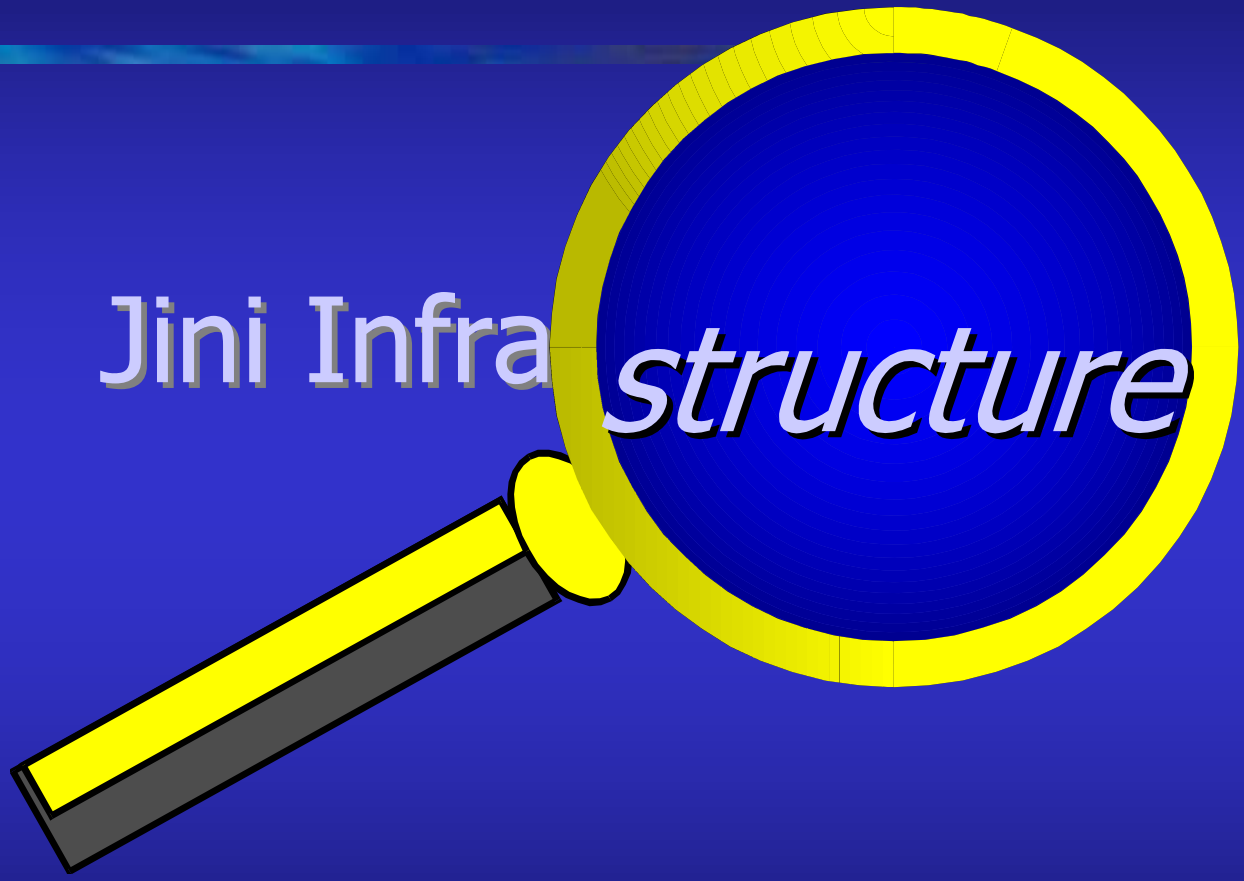
- Code can be downloaded, but: where from?
- Location of code is transferred together with its state (“codebase”)
- Codebase is a list of URLs
- URL might be
  - directory containing the class tree
  - JAR file containing the classes
- Codebase is set as a property when starting a JVM (“`-Djava.rmi.server.codebase=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

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
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Summary

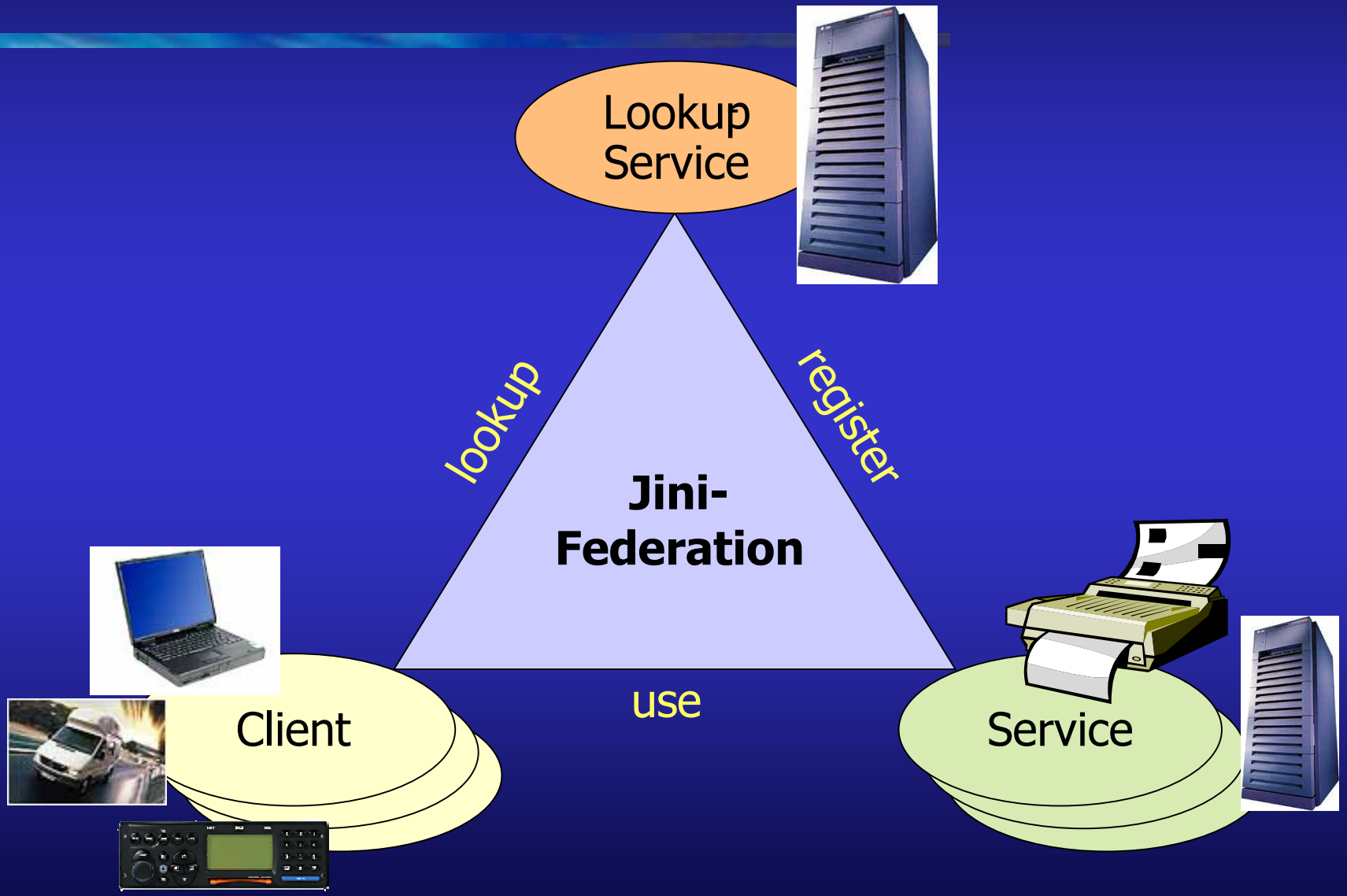
Jini Infrastructure

- 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



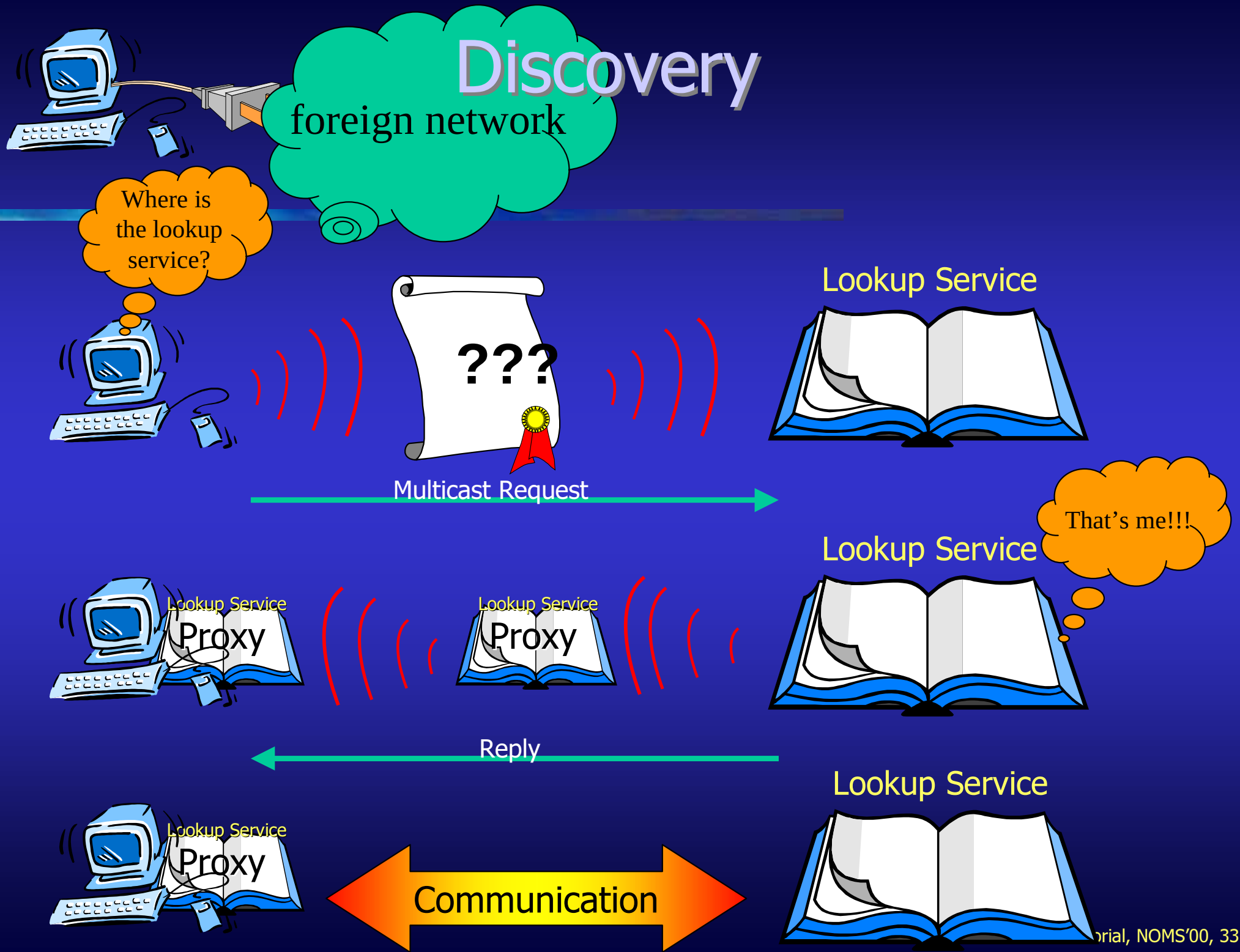
Lookup Service

- 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)

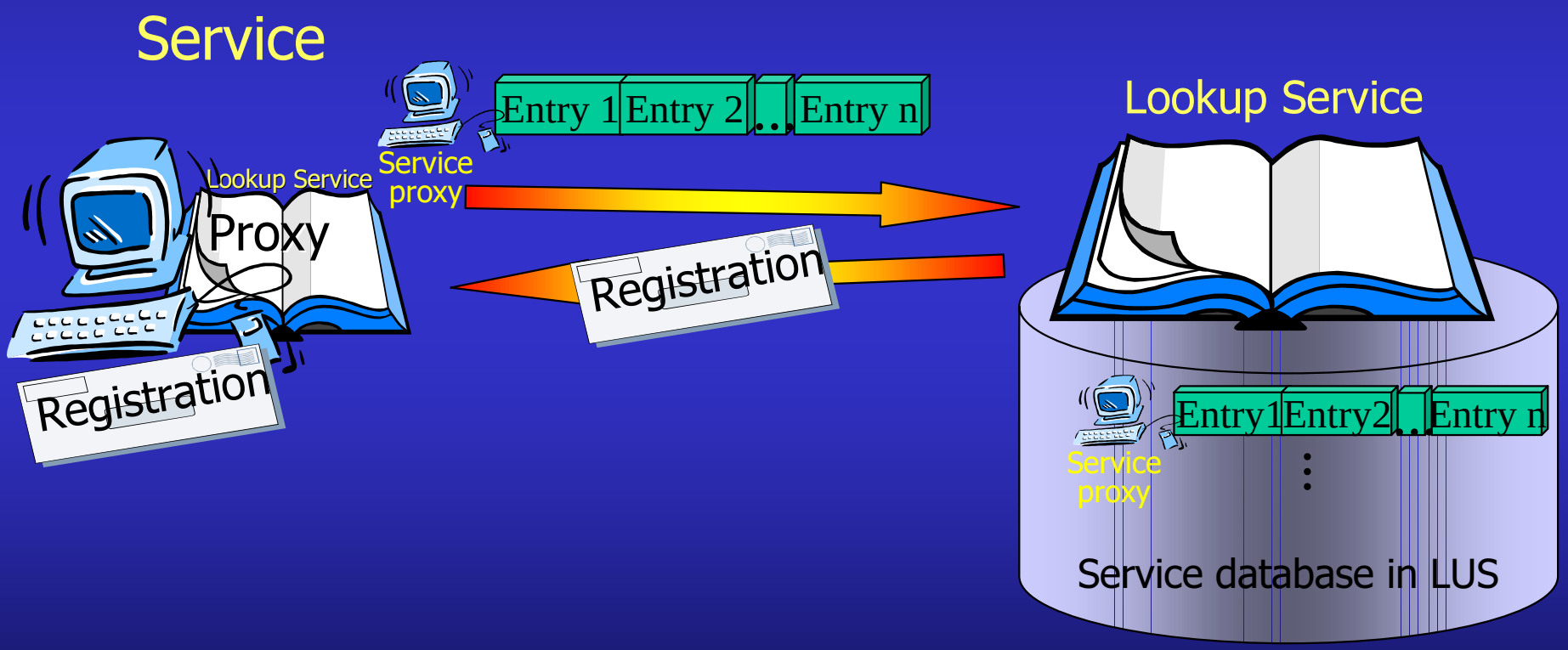
Discovery


foreign network
Where is the lookup service?
???
Lookup Service
Multicast Request
That's me!!!
Lookup Service
Lookup Service
Proxy
Lookup Service
Proxy
Reply
Lookup Service
Lookup Service
Proxy
Communication

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

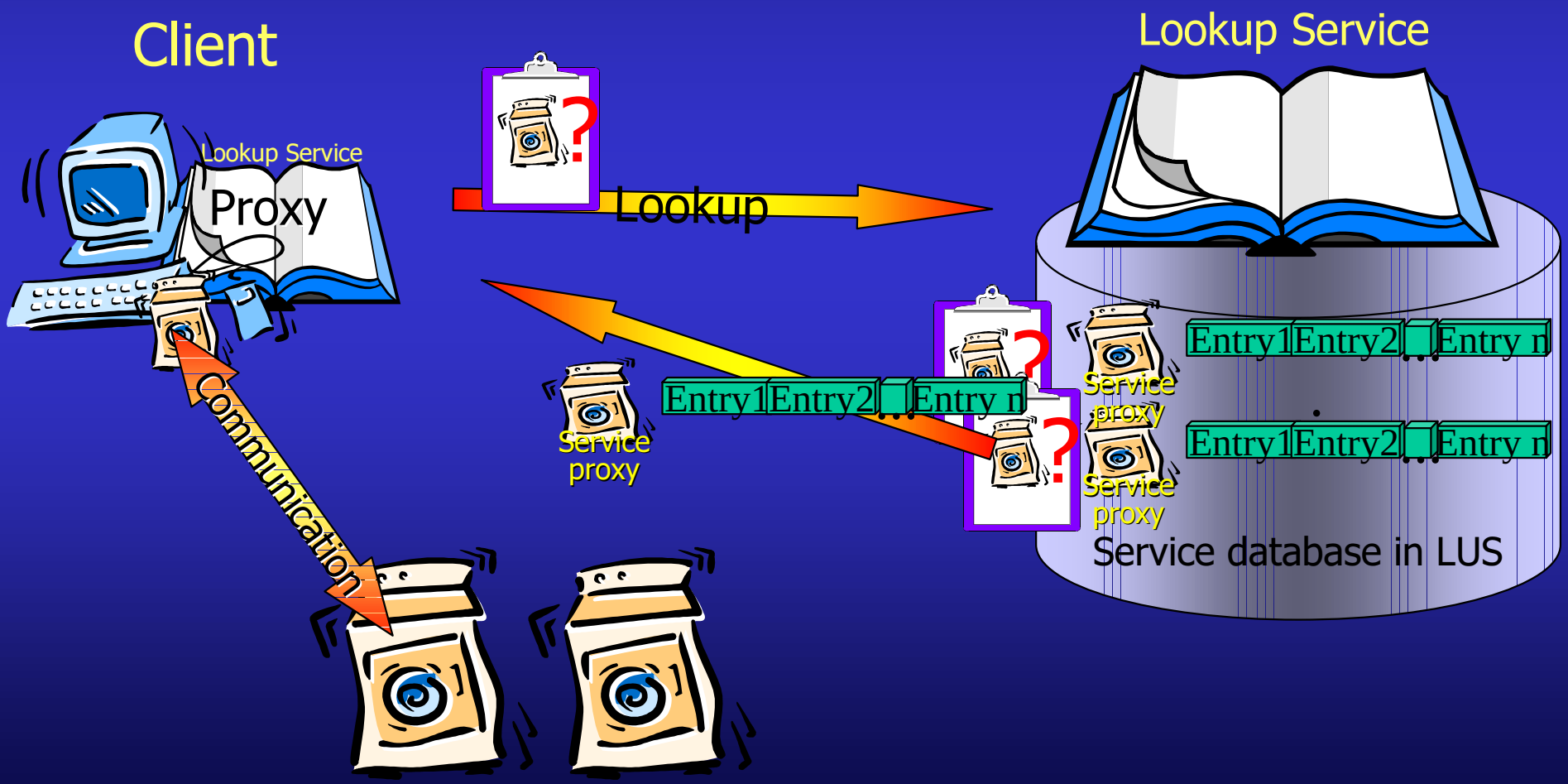
Join


Service
Lookup Service
Proxy
Registration
Service
proxy
Entry 1
Entry 2
Entry n
Registration
Lookup Service
Service
proxy
Entry1
Entry2
Entry n
Service database in LUS

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 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

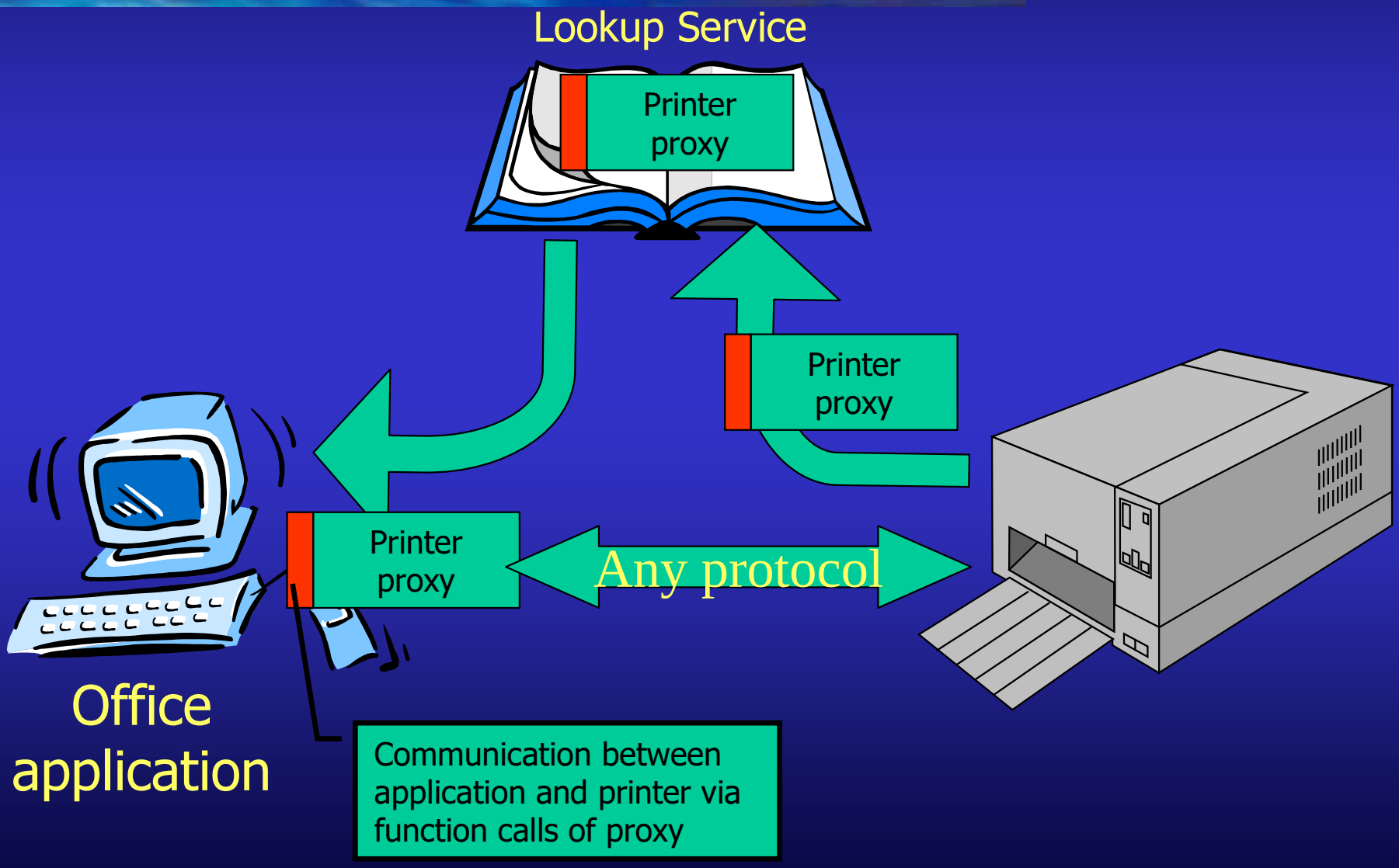
Lookup


Client
Lookup Service
Proxy
Lookup
Lookup Service
Entry1 Entry2 ... Entry n
Entry1 Entry2 ... Entry n
Service proxy
Service proxy
Service proxy
Service database in LUS
Communication

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
  }
[...]
}
```

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());
[...]
 }
}
```

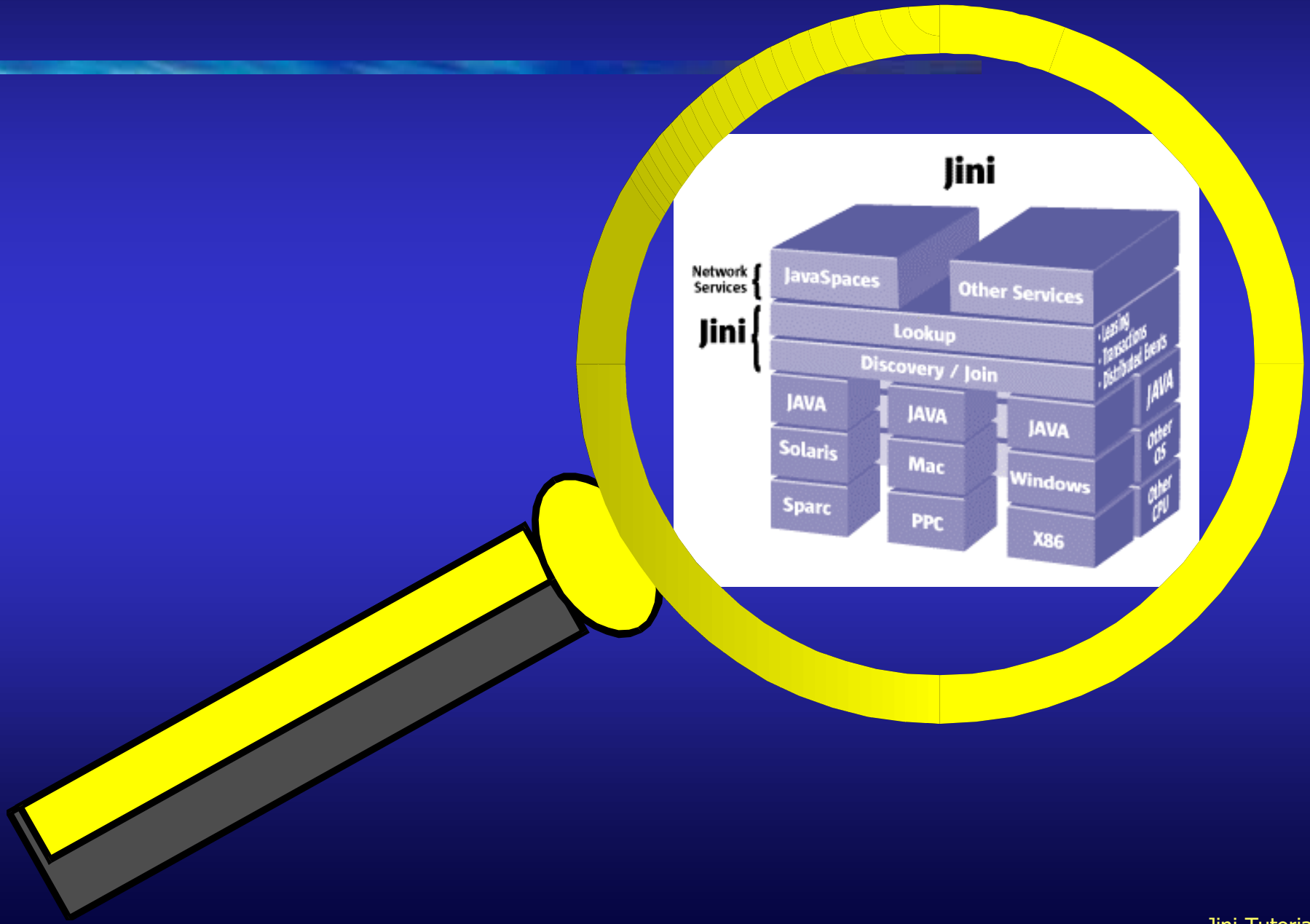
Unicast Client

```
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
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Summary

Overview 2

- Detailed look at
  - discovery
  - join
  - lookup
  - entries
  - lookup service
    - proxies
    - groups
  - leases
  - distributed events

Jini infrastructure

Jini programming model

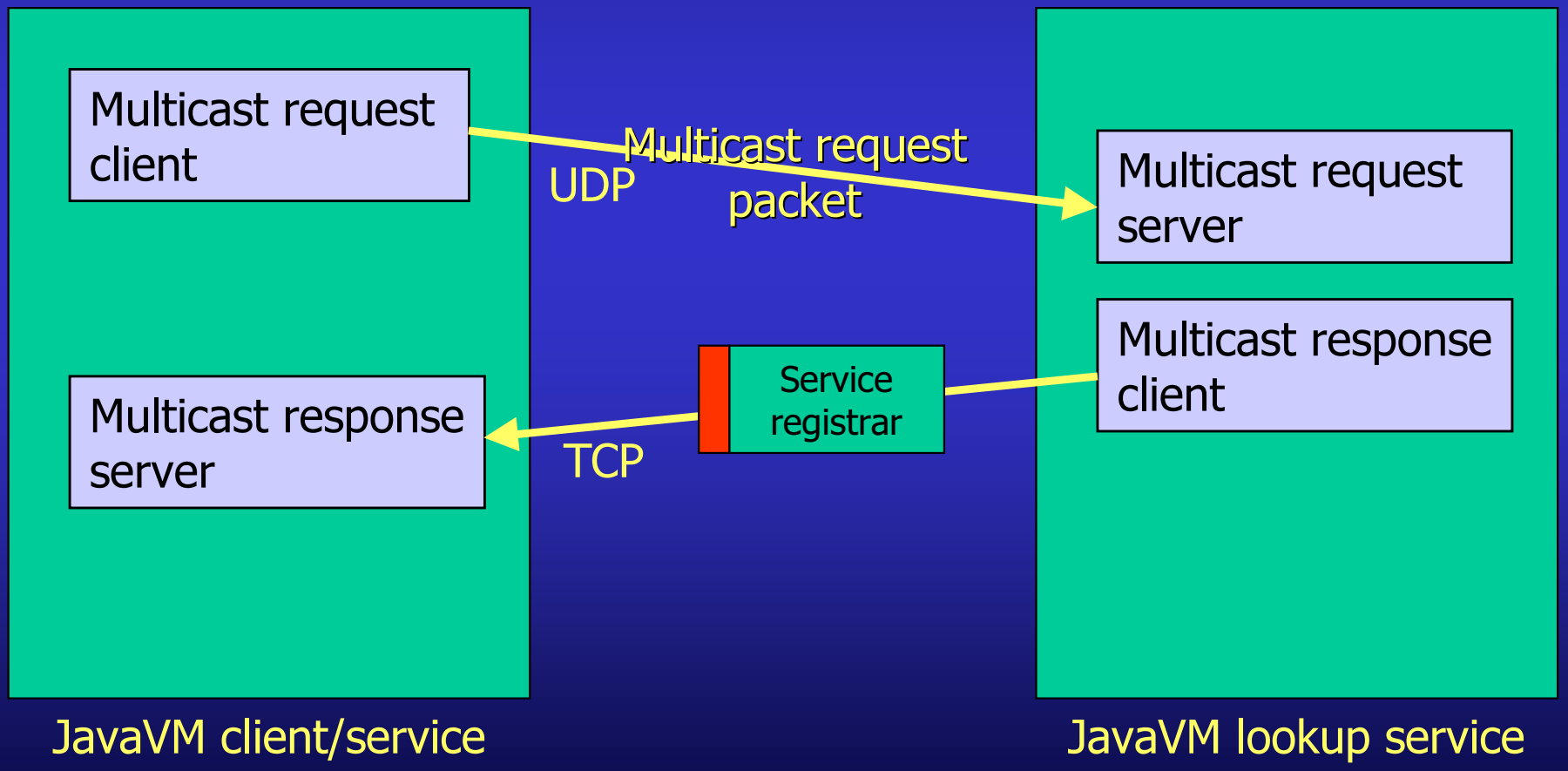
Discovery

- 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

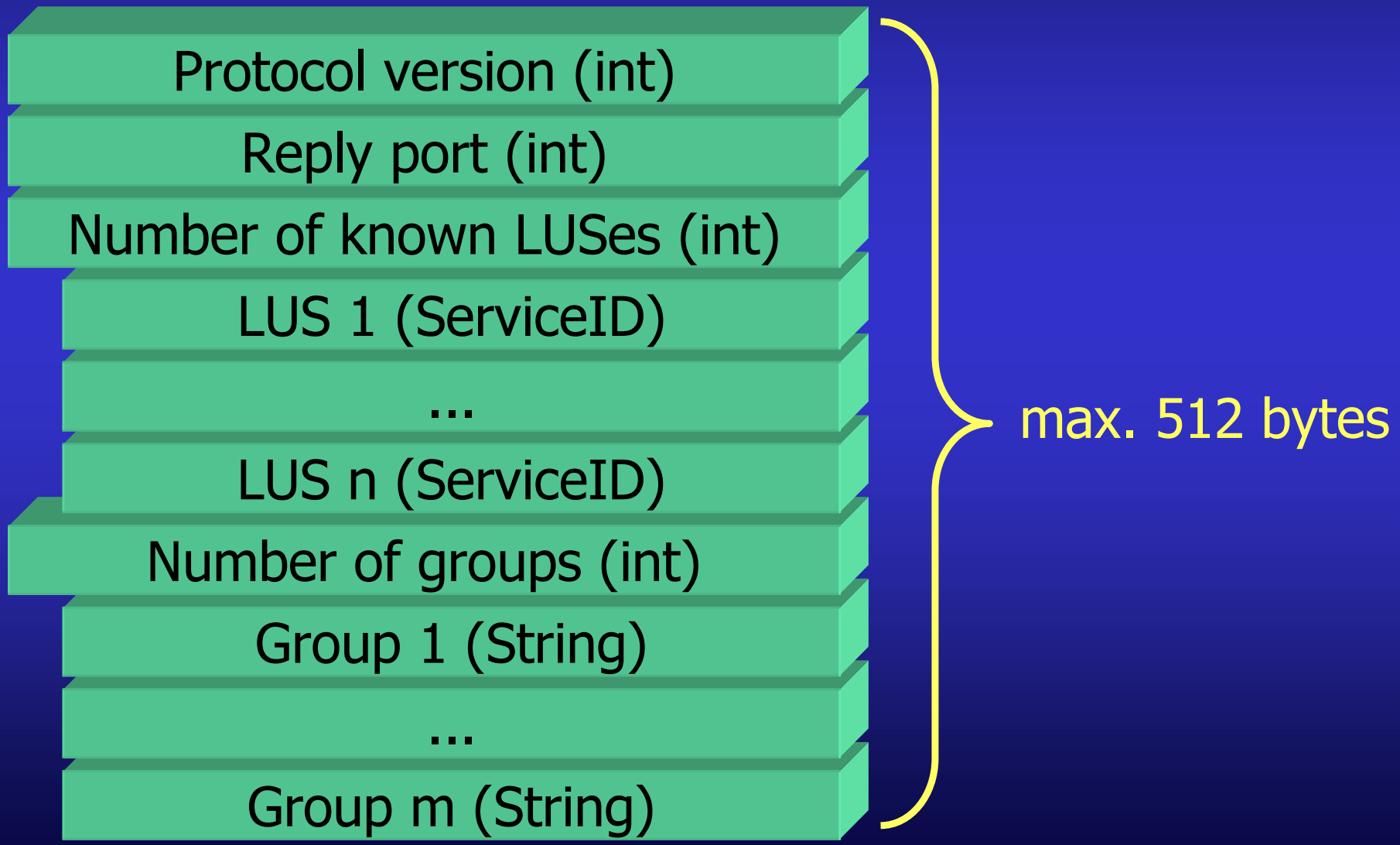
Multicast Request Protocol

- 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

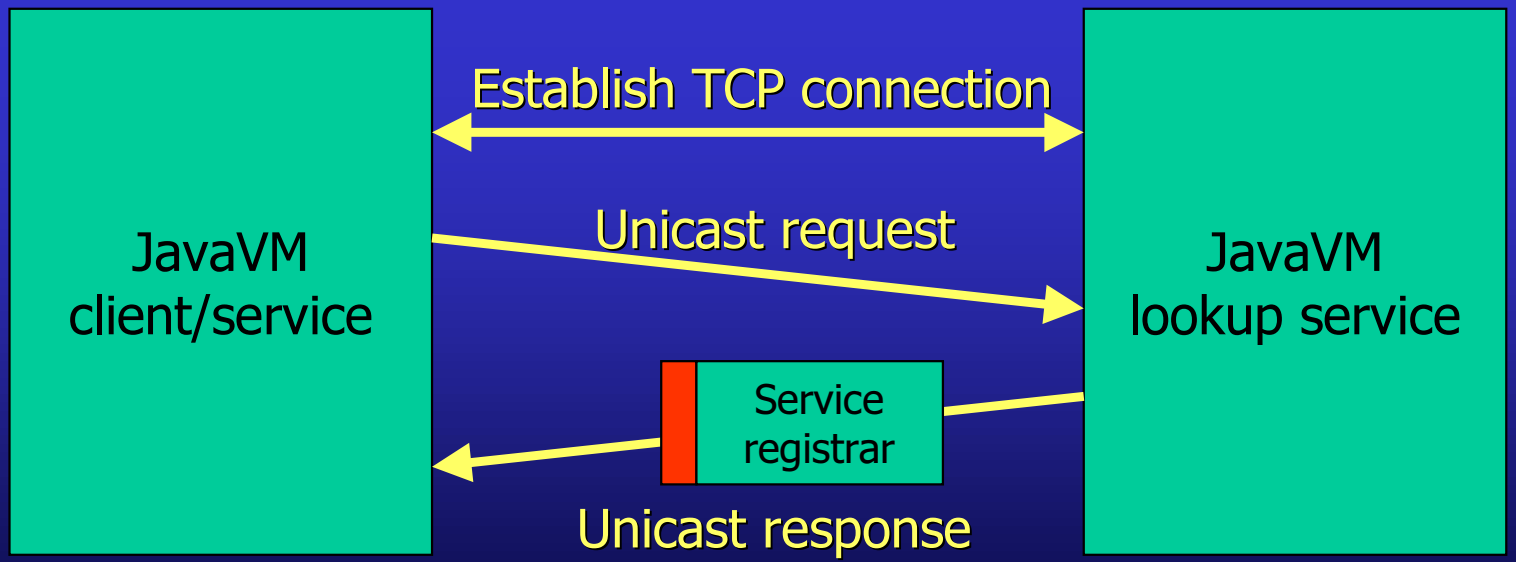


Multicast Request Packet



Unicast Discovery Protocol

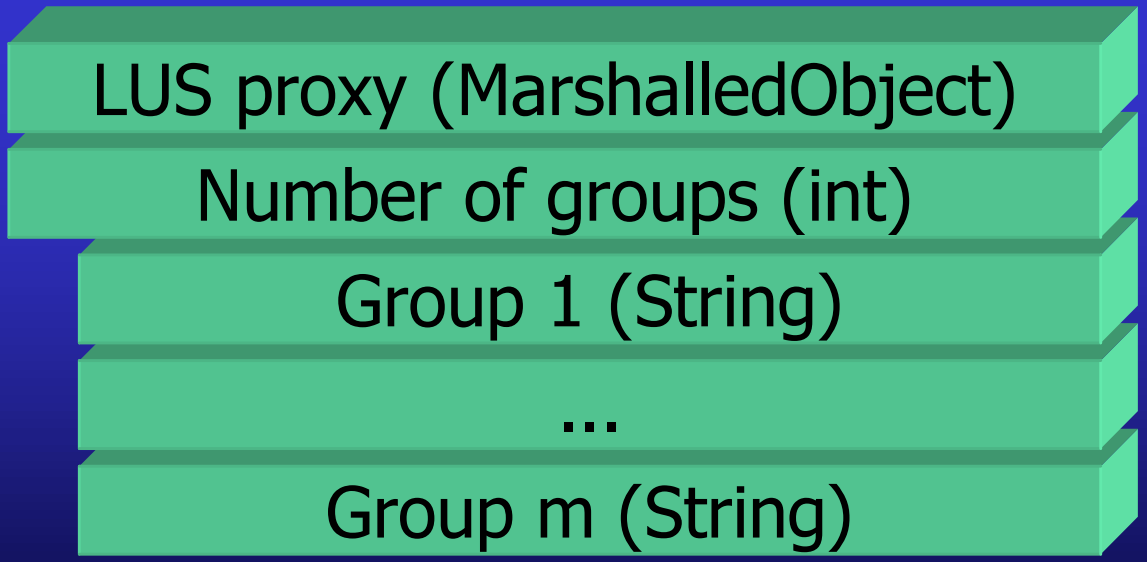
- Used to contact lookup services with known locations
- Uses TCP (unicast) connections to port 4160
- Simple request-response protocol



Unicast Packets

- Unicast Request (client → lookup service)

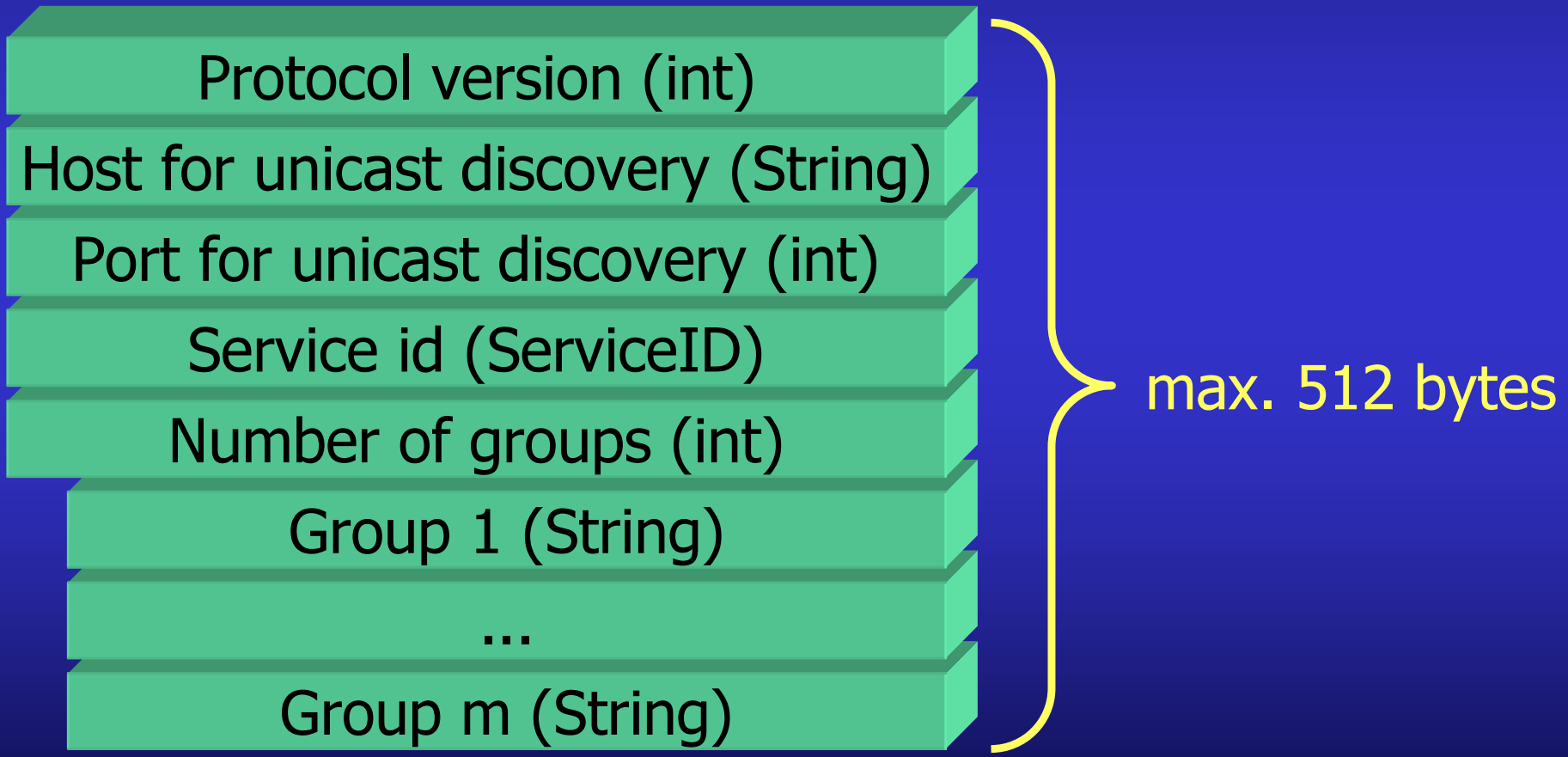
- Unicast Response (lookup service → client)



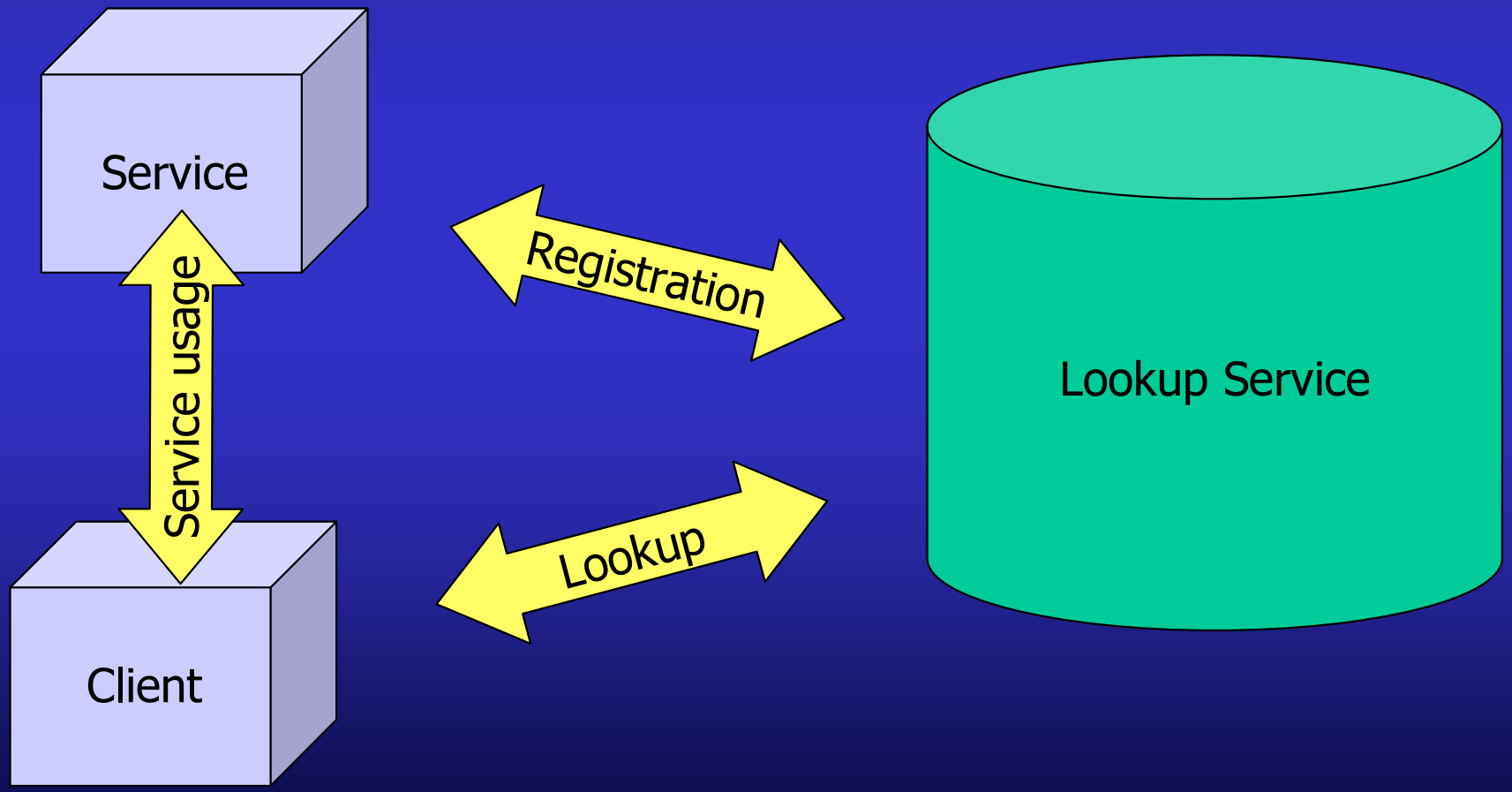
Multicast Announcement Protocol

- 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



Lookup Service Details



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: 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

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.: PrinterParams (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;
         [...]
   }
```

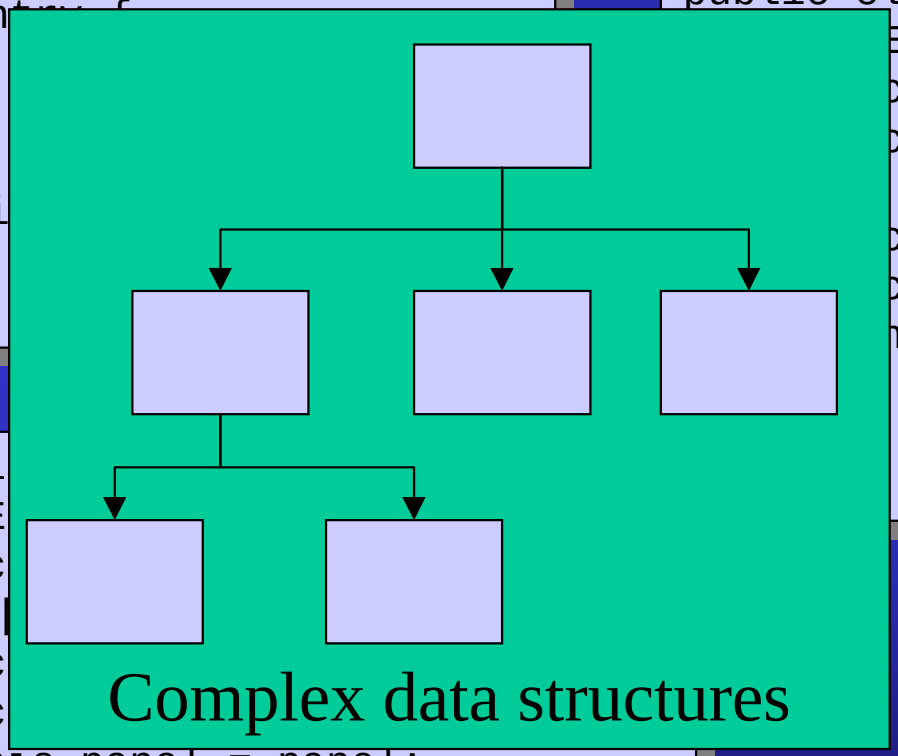
```
public class AWTGUIEntry extends
AbstractEntry {
   public Panel panel;
         [...]
   public GUIEntry() {}
   public GUIEntry(Panel panel) {
       this.panel = panel;
         [...]
   }
}
```

Entries (Examples)

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

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

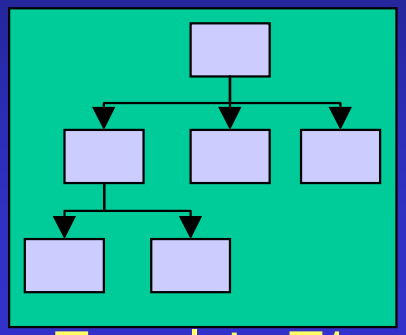
```
public cl
AbstractE
   public
   public
   public
       this.panel = panel;
         [...]
   }
}
```



Template Matching (Examples)

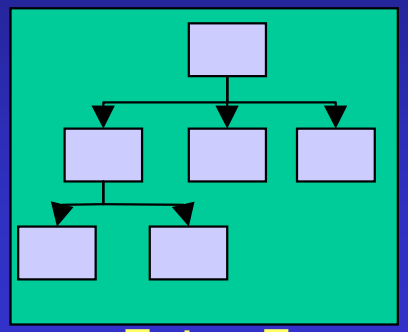
- `ServiceTemplate(null, Print.class, null)`
  - matches all services that implement interface Print
  - attributes are ignored (wildcard `null` matches everything)
- `ServiceTemplate(serviceID, null, null)`
  - matches at most one service
- A `ServiceTemplate` filled with entries matches exact data 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)

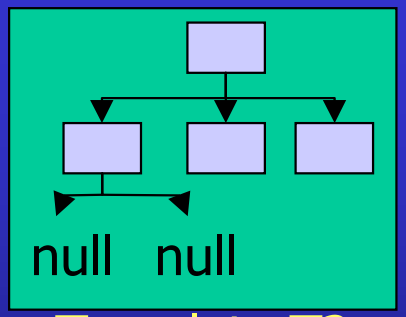


Template T1

matches

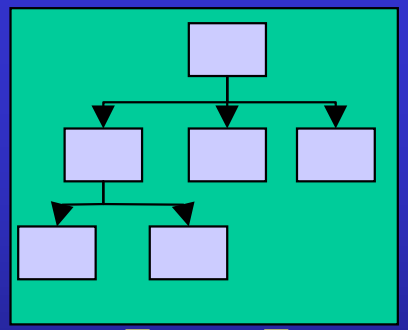


Entry E

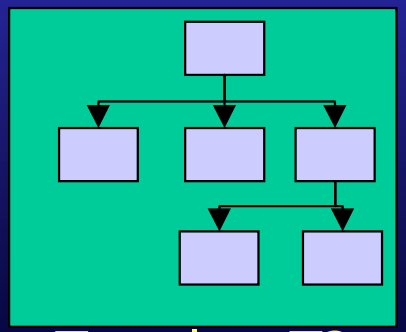


Template T2

matches

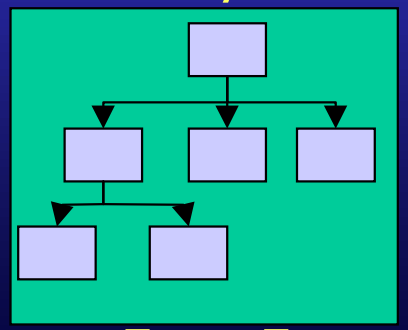


Entry E



Template T3

does NOT match



Entry E

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)
```

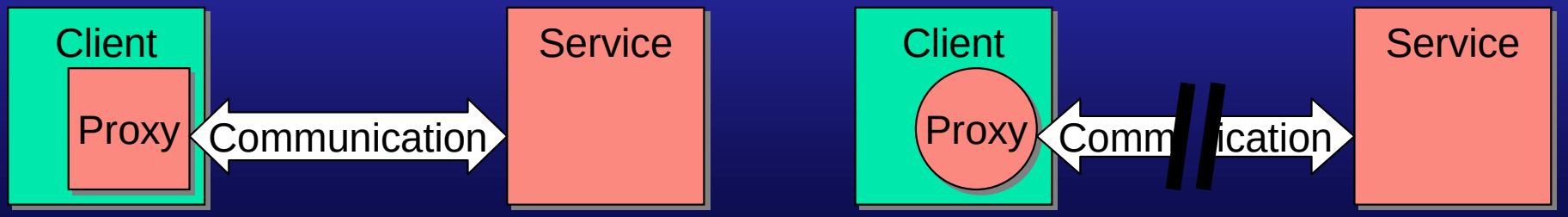
- Service-ID is a 128 bit “universally unique identifier”
  - generated by the lookup service when registering the first time
  - service has to reuse it for all later registrations
  - service has to make it persistent

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

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
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Summary

Leases

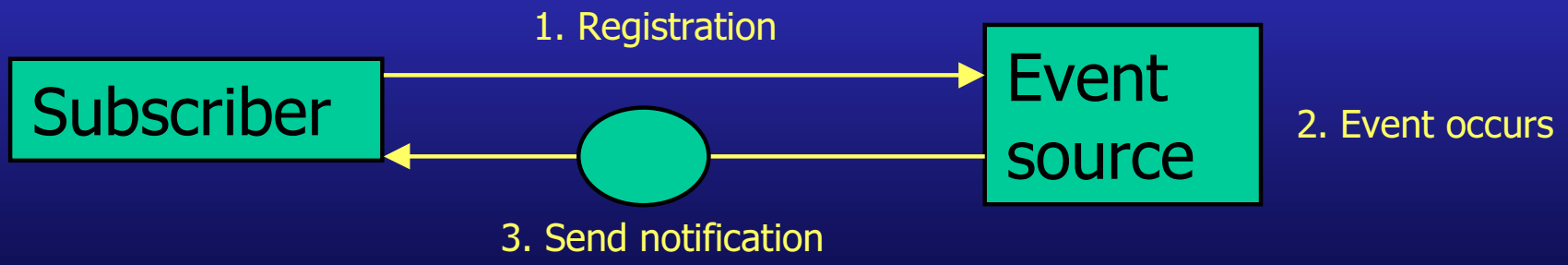
- Leases are contracts between two parties
- Leases introduce the notion of time
  - resource usage is restricted to a certain time frame
  - interaction is modeled by repeatedly expressing interest in some resource:
    - I'm still interested in X
      - renew lease periodically
      - lease renewal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

What To Use Leases For?

- 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:

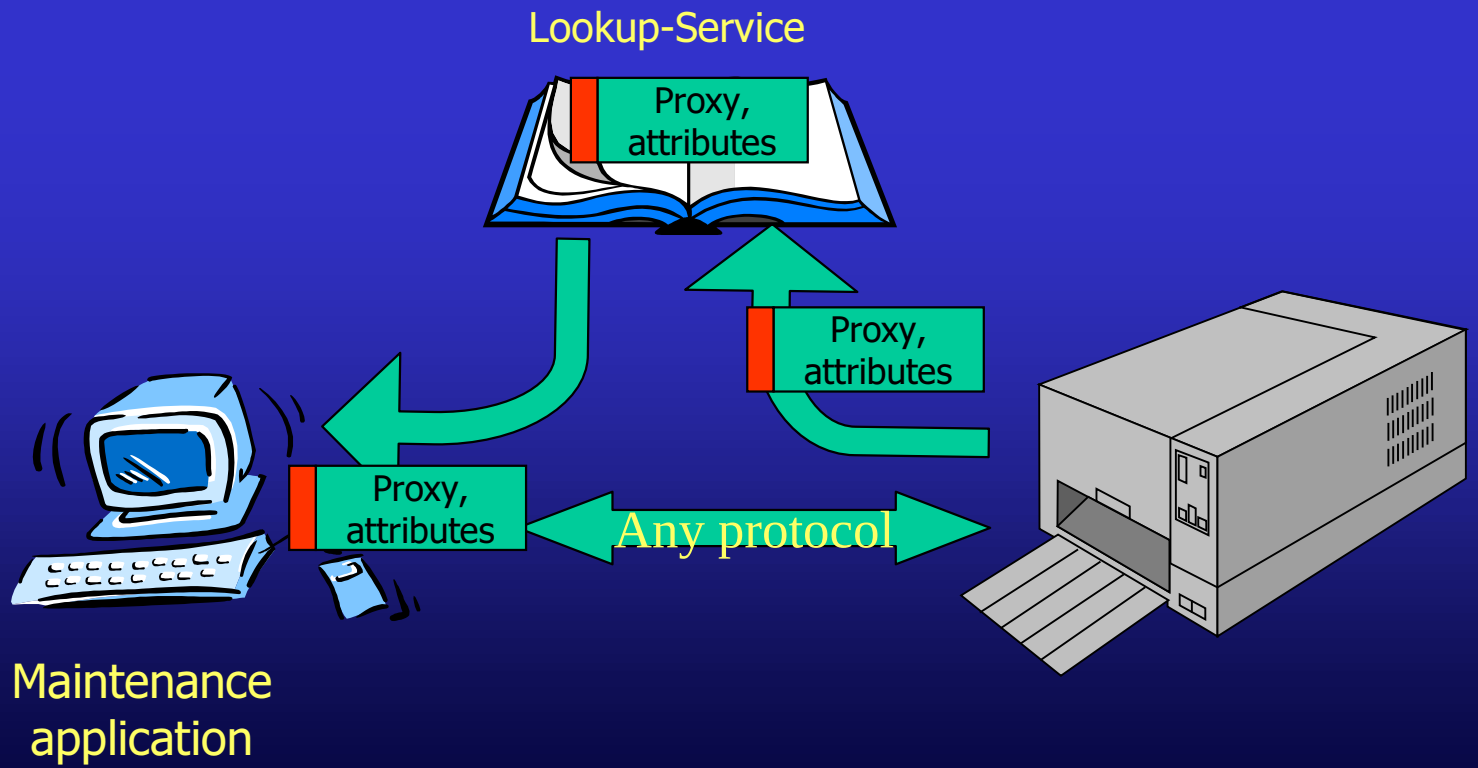


Distributed Events

- 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)

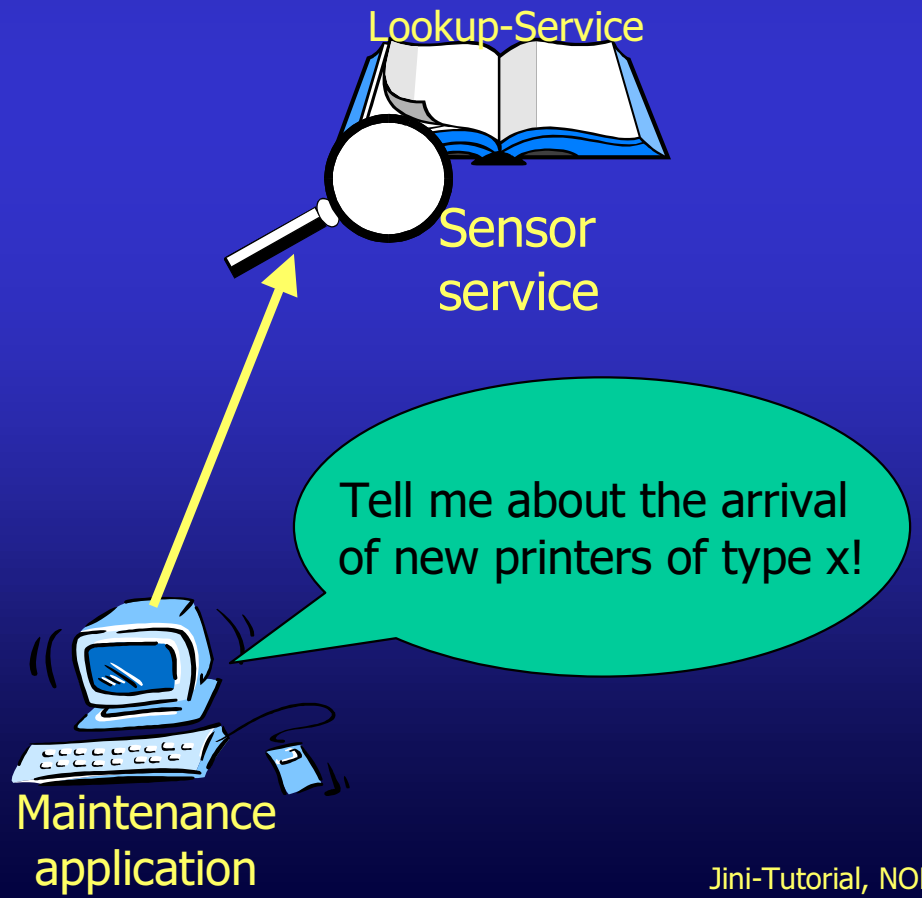
Distributed Events (Example)

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



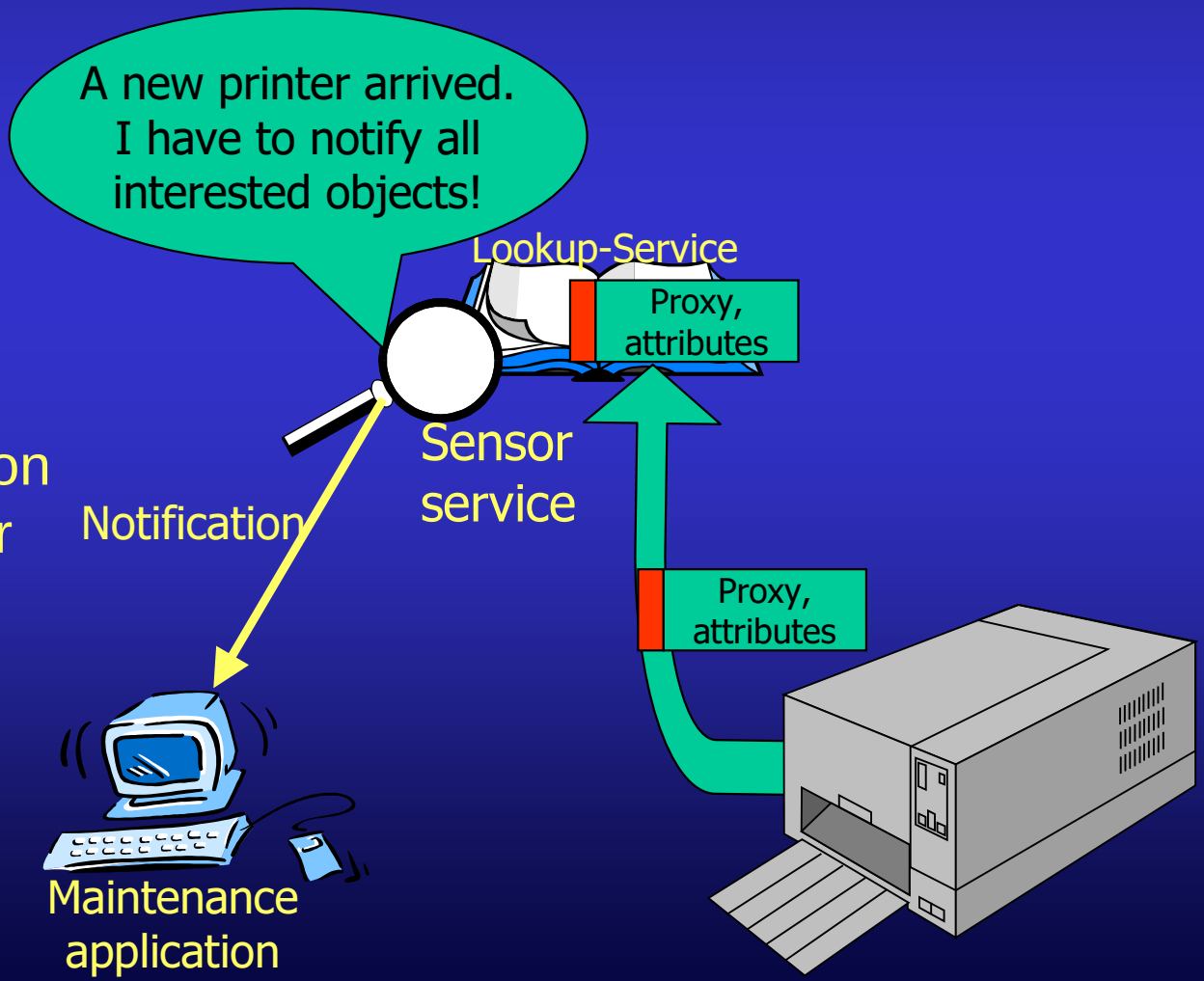
Distributed Events (Example)

- 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

Overview

- Jini, what's that?
  - motivation
  - overview
- Jini infrastructure
  - lookup service
  - discovery & join protocols
  - programming example
  - detailed infrastructure
- Jini programming model
  - leasing
  - distributed events
- Summary

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, lookup
  - 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
- 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 of Part 1...

Contact:

Gerd Aschemann

Darmstadt University of Technology

Email:

ascheman@informatik.tu-darmstadt.de

Homepage:

http://www.informatik.tu-darmstadt.de/VS/Mitarbeiter/Aschemann/

Peer Hasselmeyer

Darmstadt University of Technology

Email:

peer@ito.tu-darmstadt.de

Homepage:

http://www.ito.tu-darmstadt.de/staff/Peer/