Jini-based Management

Gerd Aschemann
Peer Hasselmeyer
Darmstadt University of Technology

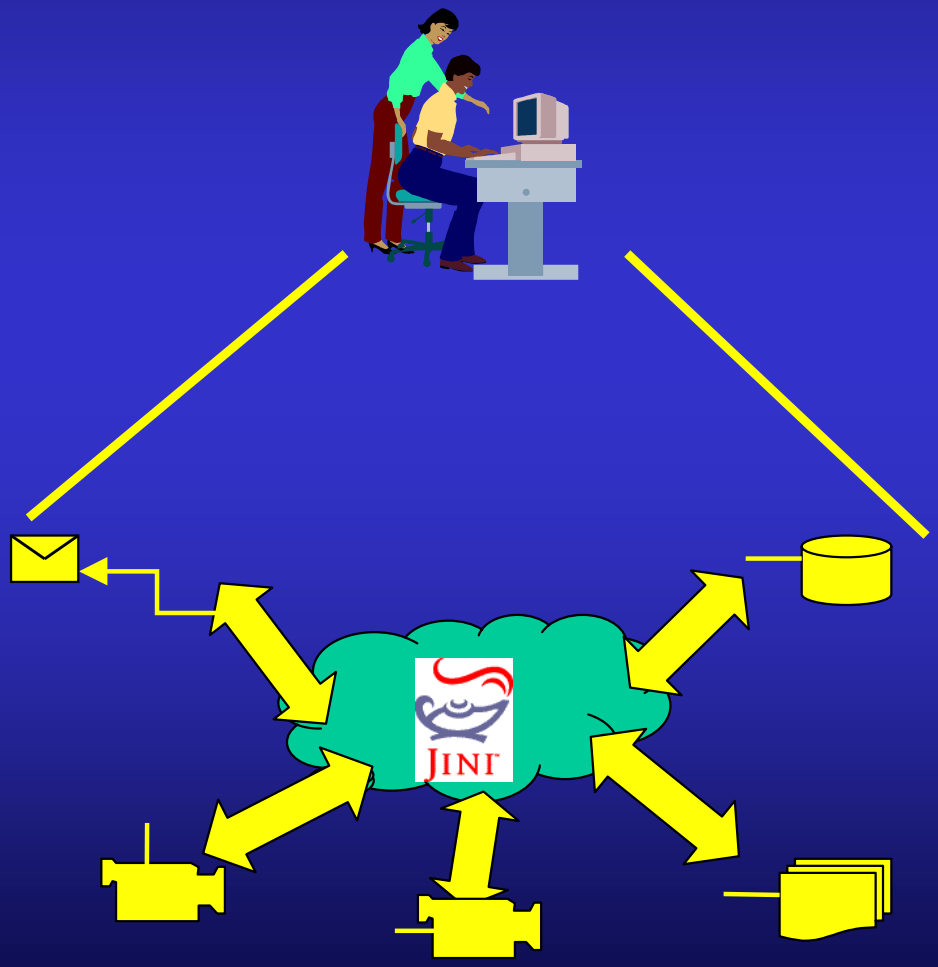
- **J**ava **I**ntelligent **N**etwork **I**nfrastructure
- **J**ini **I**s **N**ot **I**nitials

Part II

Jini and Management
or
Management and Jini

Jini and Management

- Management and Administration of Jini
  - Well-behaved services
  - Standard Admin interfaces
  - Admin applications and GUIs
  - Custom Admin interfaces
  - Integration with management standards and platforms
  - Manageable Jini core services
  - Standardization



Management and Jini

- Jini-based management, an open management federation
  - Self-configuration
  - Join on demand
- Integration of legacy (management) applications and services

Motivation

- Introduce Jini (handled in the first part)
- Distinguish
  - Management of Jini
  - Management through Jini
- The second requires the first
  - Jini management services must be administered themselves
  - Jini as a new distributed systems technology calls for management as well as any other comparable technology (DCE, CORBA, DCOM, WWW, Application-Servers, ...)
- Show the benefits of Jini as a management-enabling technology but also discuss the drawbacks; make use of new concepts
- Describe the integration of the new technology with the established ones, be it management standards or generic distributed systems architectures
- Distinguish
  - administration in the small
  - administration in the large

Administration of Jini services

- Terminology
- Well-behaved services
- Standard Admin interfaces
- Admin applications and GUIs
- Custom Admin interfaces
- Integration with management standards and platforms
- Manageable Jini core services
- Standardization

Terminology

- Administration in the *small scale*: manage one or few particular services directly, i.e., do monitoring and control directly via an integrated user interface or through simple generic management applications.

- Administration in the *large scale*: services provide a programmable interface to allow for automatic monitoring and control through arbitrary management services, e.g., service configuration and fault detection.

Well-behaved services (recap)

- Wait on startup to prevent network storms
- Perform multicast discovery
  - join configured groups
  - and/or join “public” group
- Join preconfigured LUSs by unicast disc.
- Renew leases regularly with all LUSs it is registered with
- On loss of contact to “unicast” LUS retry periodically to reestablish connection
- Provide change of attrs. to every reg. LUS
- Save configuration persistently (ID, groups, attrs., ...)

Performed automatically by

`JoinManager`

except for Persistence

Administrable ...

- Service may implement `net.jini.admin.`**`Administrable`**

```
public abstract interface Administrable {
    public java.lang.Object getAdmin()
        throws java.rmi.RemoteException;
}
```

- **`getAdmin()`** may return a Java object which implements appropriate admin. interfaces for the service
- “Management by Delegation”
  - core service functionality is independent of service management
  - proxy of service is not overloaded with administrative interfaces (speed up download, reduce storage overhead in LUS)
  - most clients do not need to do administration (and should not)
  - fetch administration object on demand

Standard Admin Interfaces

- Sun has only defined very few and simple administration interfaces

Package	Interface	Description
net.jini.admin	**JoinAdmin**	Control participation of service in Join protocol
net.jini.lookup	**DiscoveryAdmin**	Control interaction of a LUS with other LUSs
com.sun.jini.admin	**DestroyAdmin**	Control service termination
com.sun.jini.admin	**StorageLocationAdmin**	Control location of persistent storage of the service
com.sun.jini.reggie	**RegistrarAdmin**	Control logs and leases
com.sun.jini.mahout	**RegistryAdmin**	Control binding of service with RMI registry
com.sun.jini.outrigger	**JavaSpaceAdmin**	??? Even Sun doesn't really know!?

- These need to be extended in the future

Example (1): **DestroyAdmin**

- A component implementing the **DestroyAdmin** interface can be finalized by a client calling it's **destroy()** method

```
public interface DestroyAdmin {
    public void destroy()
        throws java.rmi.RemoteException;
}
```

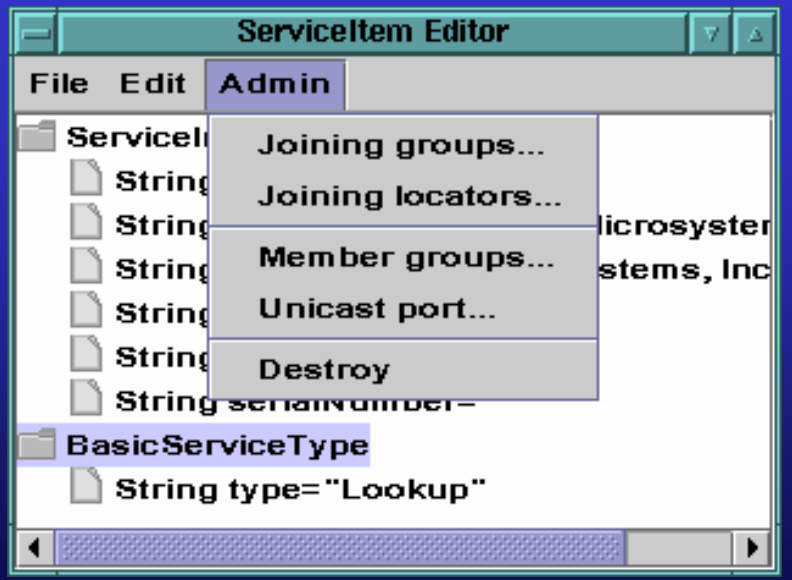
- The specification of this interface requires the component to be completely destroyed, i.e., including its persistent state

Admin Applications and GUIs

- Generic GUI is/should be part of LUS browser, e.g., start Suns example browser with "**`-admin`**" option

```
java \
    -Djava.security.policy=$JINI_HOME/example/browser/policy \
    -Djava.rmi.server.codebase=http://.../jini-examples-dl.jar \
        com.sun.jini.example.browser.Browser -admin
```

- Service should provide "self-contained" UI for administration in the small scale
- Service should provide a programmable management interface for administration in the large



Provide a "self-contained" UI

- Service proxy implements a textual or graphical user interface, e.g., by extending **`Applet`**, **`Frame`** or **`JFrame`**
- *drawbacks*
  - Client cannot use service at all, if it does not contain base classes (here AWT or Swing) or cannot interact with a user in an appropriate way, i.e., if the client does not have an I/O device
  - Download and storage of proxy in LUS wastes bandwidth and other resources (compare administration delegate object)
- Better use
  - User interface delegate object(s)
  - Special **`Entry`** attributes for user interface(s)
- "ServiceUI" working group defines standard

User Interface Delegate Object(s)

- UIs are generated by factories
- Factories are attached to services in entries
- Different kinds of UIs possible
  - e.g. AWT, Swing, Speech, tactile, ...
  - each type has its own factory, e.g. `PanelFactory`, `JComponentFactory`, ...
- Different audiences (“roles”) possible
  - e.g. “main” (user), admin, about, ...
  - represented by interfaces, e.g. `net.jini.lookup.ui.AdminUI`

Recap: **Entry** Attributes...

- When a service proxy object is registered with the LUS, a set of **Entry** objects is associated with the service

```
[...]
  Print proxy = new PrintImpl();
  Entry[] attribute = new Entry[2];
  attribute[0] = new Name("PrintService");
  attribute[1] = new ServiceInfo ("Print Service",
                      "GA", "NOMS Org.",
                      "2000", "", "08/15");
  JoinManager jmgr
        = new JoinManager(proxy, attribute,
                          (ServiceIDListener) this,
                           new LeaseRenewalManager());
[...]
```

- Clients can retrieve services by specifying a template containing a **ServiceID**, **Entry** objects, and **Class** objects

User Interface Delegate Object(s)

- Generic entry:

```
public class UIDescriptor extends AbstractEntry {
    public String role;
    public String toolkit;
    public Set attributes;
    public MarshalledObject factory;

    public final Object getUIFactory(...) {}
}
```

- Different factories for different kinds of UIs, e.g. Panel:

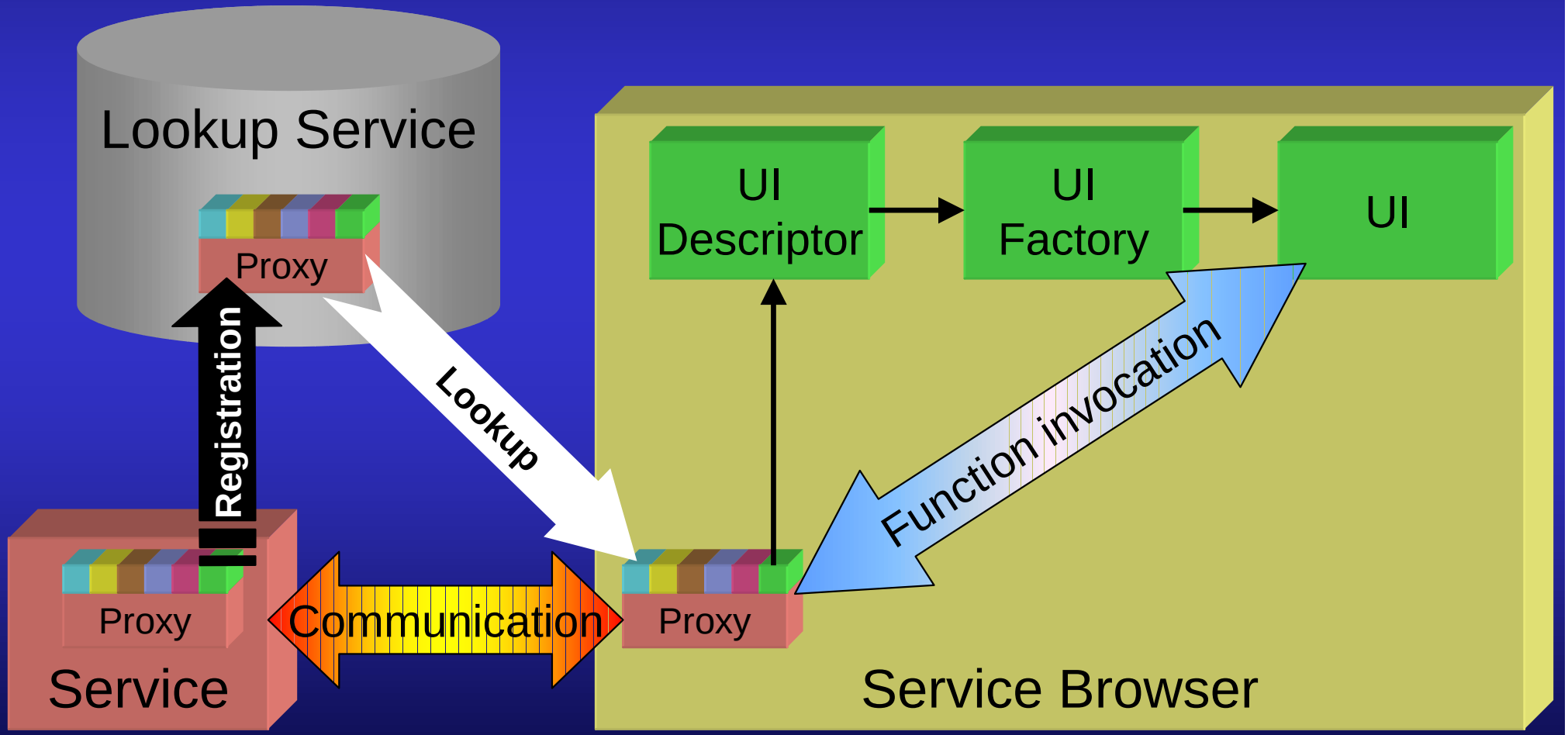
```
public interface PanelFactory extends java.io.Serializable {
  String TOOLKIT = “java.awt”;
  String TYPE_NAME = “net.jini.lookup.ui.factory.PanelFactory”;
  java.awt.Panel getPanel(Object roleObject);
}
```

- `roleObject` depends on role of this UI factory, currently always service’s proxy object

User Interfaces: Deployment

- Service browser can display UI
  - selection depends on available / preferred technology (AWT, Swing, ...) and role
  - choose appropriate entry
  - extract factory
  - create UI
  - display UI
- User interface classes talk directly with service proxy

UI Scenario


Lookup Service
Proxy
UI Descriptor
UI Factory
UI
Registration
Lookup
Function invocation
Proxy
Communication
Proxy
Service
Service Browser

Custom Admin Interfaces

- *Management within the Jini world*
- Besides the pre-defined Admin interfaces current and future services will need additional administration APIs to allow for the management in different environments
  - generic interfaces as we know them from *classical* management standards
    - New Entry types to provide „static“ information about services
    - Monitoring classes to observe objects
    - Classes to control and configure the administrative state of services
  - Java-conformant
    - information modeling
    - remote access

Example 2: `RuntimeInfo`

- New Entry type `RuntimeInfo` holds information about the system hosting the service

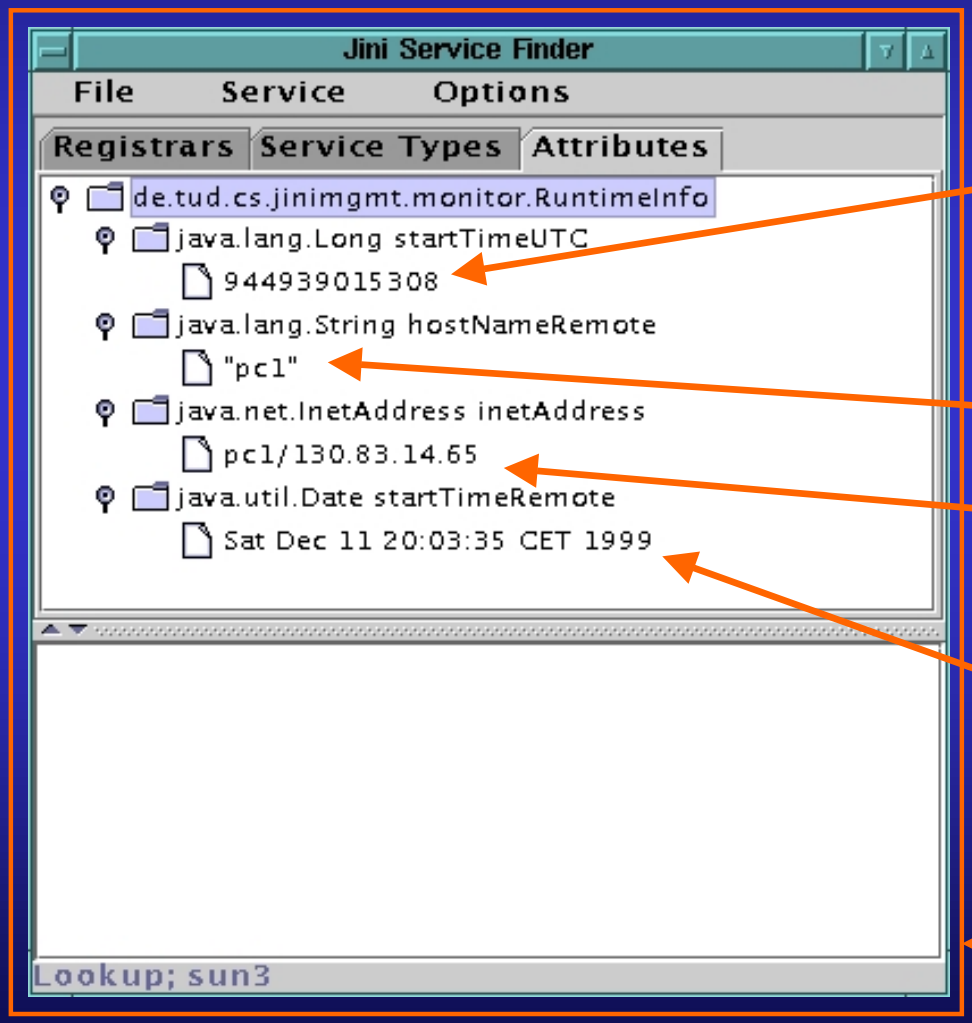
```
package de.tud.cs.jini.mgmt.monitor;
public class RuntimeInfo
    extends AbstractEntry
    implements ServiceControlled
{
    // public serializable fields
    public InetAddress inetAddress;
    public String hostNameRemote;
    public Long startTimeUTC;
    public Date startTimeRemote;
    // Constructors
    public RuntimeInfo() {
        // The default constructor will be used when a
        // serialized instance is reinstantiated
        ...}
```

RuntimeInfo (cont.)

```
        public RuntimeInfo(
            InetAddress inetAddress, String hostNameRemote,
            Long startTimeUTC, Date startTimeRemote) {
            // But we can also set them explicitly
            ...
        }
        ...
}
```

- If such an object is associated with the service and uploaded to the LUS as an attribute, it will be directly visualized by every LUS browser
- If it has settable properties, these may be changed by the browser, but this only changes the local copy of the (serialized/copied) object, not its original ...

`RuntimeInfo` in browser



When was the service started (in Universal Coordinated Time)?

Where is the service running?

- From service perspective
- From client perspective

When was the service launched (in local timezone of server)?

`com.oki.jini.finder.Finder` is just another (nice) Jini LUS browser

Predefined `Entry` Types

- The predefined **`Entry`** types of Jini may help in simple administration of a Jini community:

Package	Class	Description
`net.jini.lookup.entry`	**`Address`**	„Postal address“ of the service
`net.jini.lookup.entry`	**`Comment`**	An arbitrary comment ...
`net.jini.lookup.entry`	**`Location`**	Physical location of the service
`net.jini.lookup.entry`	**`Name`**	The name of the service
`net.jini.lookup.entry`	**`ServiceInfo`**	Some information about the service, e.g., its vendor, version number, etc.
`net.jini.lookup.entry`	**`ServiceType`**	Information which may be viewed in a service browser, like an icon or a tool tip.
`net.jini.lookup.entry`	**`Status`**	The administrative state of the object, either `NORMAL`, `NOTICE`, `WARNING`, or `ERROR`.

- If the service registry is handled by the join manager, the complete set of **`Entry`**-attributes of a service may be up-dated by a single operation

Control and Configure a Service

- Besides service monitoring (see below) control is one main application area of management
- Interfaces for configuration and control are usually very service specific
- However, some generic interfaces are interesting, e.g.,
  - disable/enable services, shutdown
  - setting information about
    - persistent storage location (cf. **`StorageLocationAdmin`** Entry type)
    - location of service (cf. **`LocationInfo`** Entry type)
    - maintenance responsibility, ...

Example 3: `StateAdmin`

- Official **`DestroyAdmin`** interface
  - does not define exact **`destroy()`** semantics
  - destroy means *destroy* (including persistent storage)
  - does not offer graded operations for graceful shutdown or temporary disabling a service
- Define another generic interface to perform such tasks

```
public interface StateAdmin {
    public interface State {
        public final long ENABLED = 0L;
        public final long DISABLED = 1L;
        public final long SHUTTING_DOWN = 2L;
    }
    public State state ();
```

StateAdmin (cont.)

```
    public interface ShutdownHandle {
        public class TooLateException extends Exception {};
        public void cancel()
            throws RemoteException, TooLateException;
    }
    public ShutdownHandle shutdown()
        throws RemoteException;
    public ShutdownHandle shutdown(Date at)
        throws RemoteException;
    public ShutdownHandle shutdown(long ms)
        throws RemoteException;
    public void enable() throws RemoteException;
    public void disable() throws RemoteException;
}
```

Not granting or renewing leases ⇒ perform shut-down when last lease has expired

- Semantics? (not OSI-MF SMF-10164-2!)
- At your disposal!
- Standardization required?

Monitoring Jini Services

- Monitoring is the other main application area of management
- Customized Entry types are simple but at most suitable for static management information (sort of “fire and forget”)
- Better enable service for direct observation
  - Periodical polling of a service (we know this from SNMP)
  - Provide a push model to inform interested parties of administrative state changes by Jini/Java remote events. We know this from ...
    - SNMP: Traps
    - OSI-Mgmt: Notifications
    - CORBA: Events

Expl. 4: `StateChange events`

- New event type **`StateChangeEvent`** indicates that the administrative state (**`StateAdmin.State`**) of the service has changed
- New event listener **`StateChangeEventListener`** for the consumer of such events
- New interface **`StateChangeEventSupplier`** for the producers of such events
  - implemented by the administration object returned by **`getAdmin()`**
- New class **`StateChangeEventKind`** for the sake of different polymorphic supplier interfaces

StateChangeEvent

- A state change is indicated by an event object which simply holds the new state of the indicating service and the reason why the state was changed

```
import net.jini.core.event.RemoteEvent;
public class StateChangeEvent
  extends RemoteEvent {
    protected String reason;
    protected StateAdmin.State newState;

    public StateChangeEvent (
        Object source,
        long eventID,
        long seqNum,
        MarshalledObject handback,
        StateAdmin.State newState,
        String reason) {
    ...
```

It is also a remote event,

so we inherit some fields

The consumer add. needs to know:

- Where the event comes from
- What type of event it was
- The sequence number of the event
- a context handle provided by and sent back to the consumer

StateChangeEventSupplier

- A supplier or generator in Jini terminology provides a register method to add a listener

```
public interface StateChangeEventSupplier
extends java.rmi.Remote {
  public StateChangeEventRegistration
    register (StateChangeEventKind kind,
      RemoteEventListener informMe,
      java.rmi.MarshalledObject handback,
      long leaseRequest)
  throws java.rmi.RemoteException,
    UnknownEventException;
}
```

We are only interested in particular event types (see below)

Why don't we use the more specialized

`StateChangeEventListener`

??? (see below)

The registration is subject to leasing, it will not last forever

The handback is not simply an object but a serialized object stream. So it can contain arbitrary information to restore the context!

StateChangeEventRegistration

- We want to handle the event registration by a particular object; registration is a typical design pattern used in Jini
  - Again: Leasing helps to optimize event source implementation, only alive consumers are notified of new events
  - It should be possible to retract (cancel) a registration

```
public class
    StateChangeEventRegistration
        extends EventRegistration {
    StateChangeEventRegistration (
        long eventID,
        java.lang.Object source,
        Lease lease,
        long sequenceNum) {
            ...
    }
    ...
```

Not consequently done in Jini, i.e., cancel the lease instead

We already know these

Current sequence number at time of registration

Finally: Only a convenience class to contain a multi-value result

StateChangeEventListener

- The object listening to the event must not necessarily be the object which registered with the generator
- It is informed by the occurrence of an interesting event by a call to its **`notify`** method:

```
import net.jini.core.event.RemoteEventListener;
public interface StateChangeEventListener
        extends RemoteEventListener {};
```

Inherits

```
void notify (RemoteEvent event)
    throws UnknownEventException, RemoteException;
```

- The consumer is only specified as an interface not as an object
  - semantics of notify implementation are not generic but individual
  - the implementing class can extend another class

StateChangeEventKind

- A convenience class to enable polymorphic event suppliers (remember: we wanted the admin object to implement them, for a large application it might generate arbitrary types of events)

```
public class StateChangeEventKind
    implements  Serializable, StateAdmin.State {
    protected long kind;
    public static long ANY = -1L;
    public class WrongKindException extends Exception {...}
    public StateChangeEventKind (long kind)
        throws WrongKindException {...}
    ...
}
```

Jini Distributed Events: Forwarding and Filtering

- Jini Remote Events form a 2-tier architecture
- Direct relationship between generator and listener

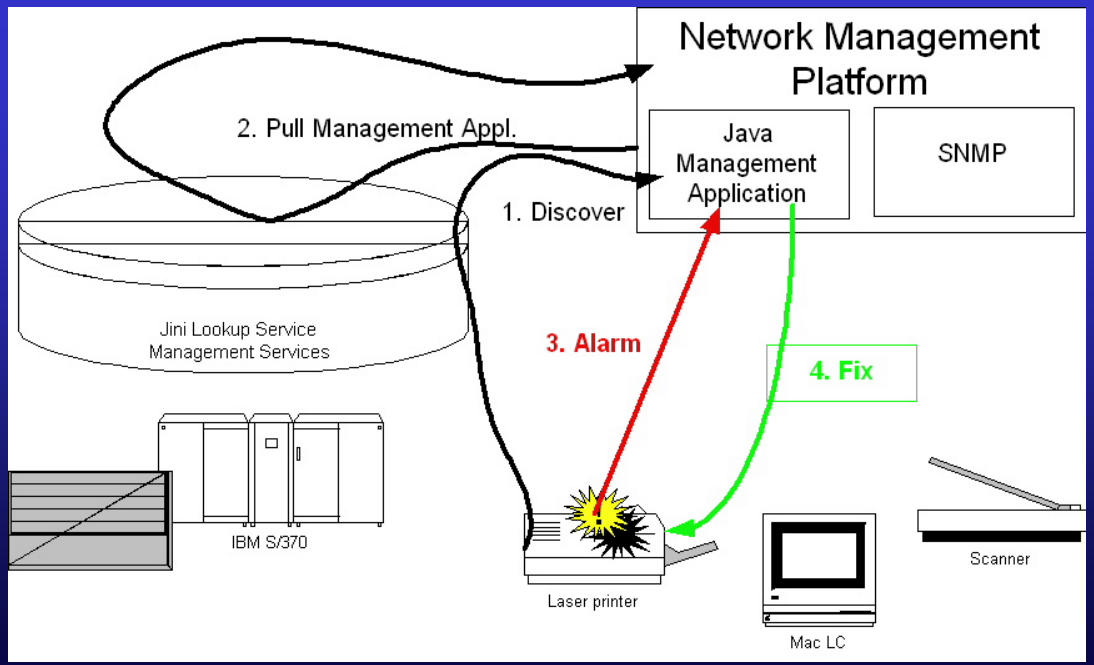
- Sun proposes to extend this to a 3-tier or multi-tier arch.
  - Store-and-Forward-agents (like CORBA-event channel)
  - Notification Filters
  - Notification Mailboxes (kind of a Messaging Service)

Integration with Management Standards and Platforms

- Remember: We are still in the „*Management of Jini*“-part!
- Several possibilities (Jini service in agent role):
  - provide your own (proprietary management interface)
  - integrate with existing products and standards, e.g.,
    - SNMP, e.g. AdventNet Agent Toolkit, Suns Java Dynamic Management Kit (JDMK, provides an SNMP adaptor)
    - OSI/TMN-Management Tools (is there a Java based agent toolkit available? Does JDMK really provide such an adaptor?)
    - CORBA-based (Java 2 technology incorporates parts CORBA technology)
    - CIM/WEBM
  - make use of Suns management frameworks
    - Java Management Extensions (JMX) + JDMK
    - Federated Management Architecture (FMA, see below)

Excursion: **JDMK** and JMX

- Java Dynamic Management Kit (JDMK)
  - a „Universal Agent Toolkit“
  - distributed management intelligence (smart agents), software distribution

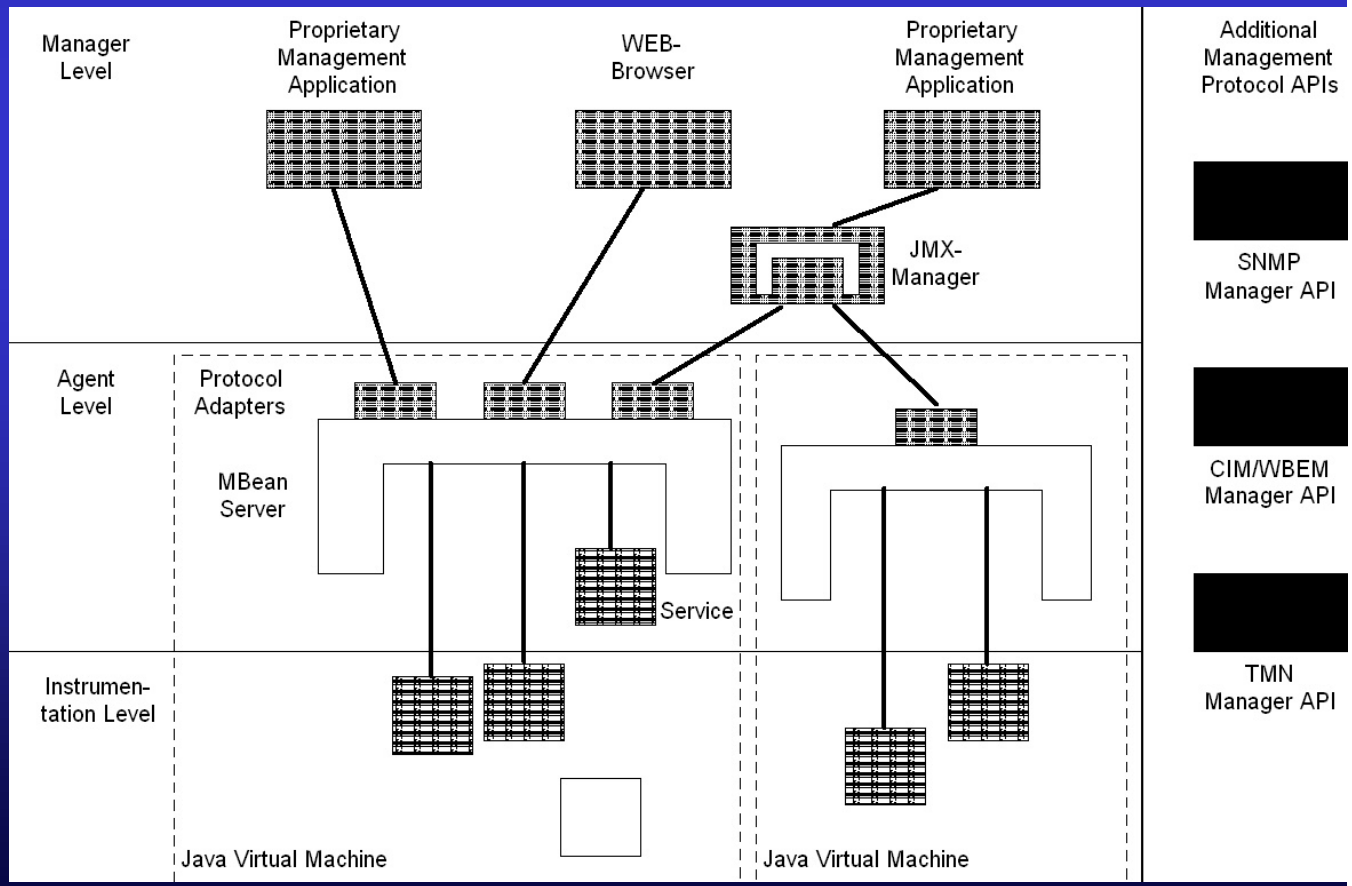


  - based on JavaBeans comp.
    - mini applications or building blocks for mgmt. apps.
    - customize by IDE
    - plug into agent framework
  - protocol adaptors (HTTP, RMI, SNMP)
  - tools (MIB compiler, stub/object generator)

Excursion: JDMK and **JMX**

- Java Management Extensions (JMX)
- three level model/architecture
  - instrumentation level
    - managed resources
    - represented by Manageable Beans (MBeans)
  - agent level
    - MBean server
    - protocol adaptors
  - manager level
    - (not yet specified)

- additional services on agent and manager level
- additional protocols (SNMP, ...)
- ...



Jini-based Management

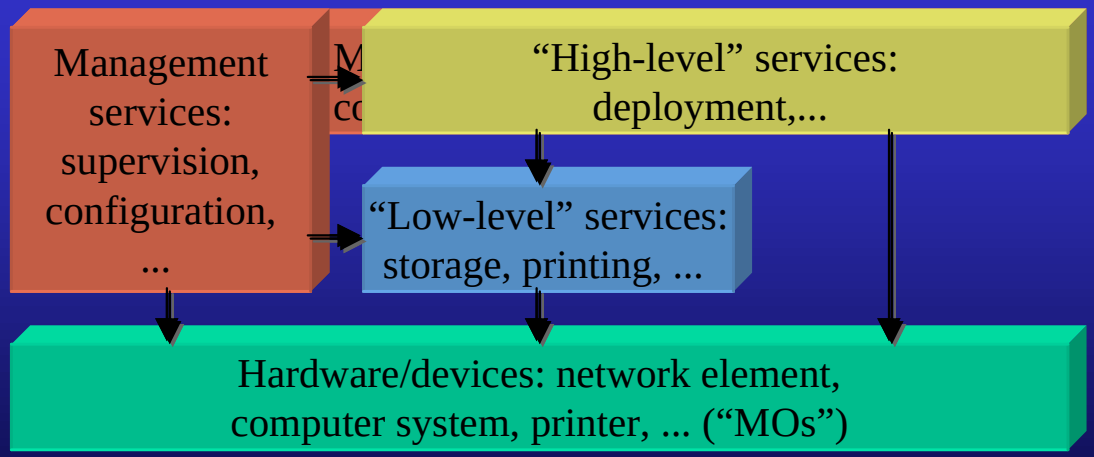
- Use Jini for management purposes
- Motivation, Requirements
  - Extensibility
  - Exchange/Migration of Components
  - Separation of Concerns
  - FaultTolerance
  - Scalability
  - Arbitrary Management Structures
  - Integration of Existing Systems
  - Drawbacks and Problems
- Using Jini Concepts
  - Leases
  - Distributed Events
  - Lookup and Join
  - Transactions
- Vision: A Jini Management Federation
- Example: Integration of legacy devices into a Jini-based management architecture

Management with Jini

- Why Jini?
- Benefits:
  - extensibility: easy, fast
  - exchange of components on-the-fly (no power-down, reset, etc.)
  - easy migration of services
  - separation of concerns/functionality
  - fault tolerance included (by redundancy)
  - scalability?
  - arbitrary hierarchical/non-hierarchical, centralized/distributed management structures possible

Management with Jini

- Different paradigms:
  - "traditional" management: focussed on network elements and systems
  - Jini: service-oriented → service management



Extensibility (1)

- Extensibility: adding functionality (= services)
- here: adding functionality at run-time
- Two kinds of services:
  - “real” services (e.g. print spooler, video-on-demand, ...)
  - management capabilities (e.g. printer configuration, traffic supervision, ...)
- Components can be added to running system
  - no need to stop/restart entire system
- Components find each other automatically (lookup/join)
  - location independent

Extensibility (2)

- Load can easily be distributed
  - add same component on different machine
  - but: depends on application
- Components can be deployed only when needed
  - one size doesn't fit all
  - don't waste disk space, computation power, and setup time on services you don't need
- Components can be individually bought or leased
- Example: adding accounting service for printing

Exchange/Migration of Components (1)

- Service updates and migration (and additions) are handled the same way:
  - start new version at new place
  - stop old service
- Clients automatically use new service instance
- Problem: finding out when clients are finished using old service
  - clients have to use leasing for service access
- Sequence:
  - start new service
  - remove from lookup service (cancel its lease)
  - wait till client's leases expired (maybe cancel them)
  - shutdown

Exchange/Migration of Components (2)

- Example:
  - print accounting server is too slow (and new version is available anyway)
  - start new accounting software on powerful machine (both are accessing the same database service)
  - let old accounting service “disappear”
  - stop old service

Separation of Concerns

- Trend in software engineering: components
- Each service should only handle its own small management area
- Each component:
  - less complex
  - easier implementation
  - robust implementation
- Components can be individually deployed, bought, leased, …
- Components can be reused
  - can serve for different clients

Fault Tolerance (1)

- Different areas
- Service entries are leased:
  - automatically removed from lookup service when stopped / crashed / no connection
  - clients always see (more or less) recent set of available services
  - lookup service does not get overloaded with stale services
- Redundant services
  - easy to deploy (just start and forget)
  - only available ones are used
  - supervision already handled by infrastructure:
    - register with lookup service for events
    - lease expiry results in removal of service entry
    - qualified listeners are notified
    - appropriate action can be taken

Fault Tolerance (2)

- Jini (and RMI) does not hide distribution
  - programmer is forced to cope with exceptions:
    - ignore them (not recommended)
    - think about them
    - add meaningful handling: e.g. use different service
  - results in more robust (and user-friendly) software
  - in small clients: 10-30% more code
- “new” error conditions
  - service not available
  - multiple services available

Scalability

- Basic scheme:
  - server overloaded → start same service on another server
- Problem: load distribution
- Application dependent:
  - every service is the same: client randomly selects one service instance (e.g. telephone directory)
  - services differ: more than one service has to be contacted by client (e.g. service provider with different costs)
    - possible solution: load balancing in service proxy

Arbitrary Management Structures

- Jini allows for any kind of management distribution:
  - hierarchical, non-hierarchical
  - distributed, centralized
  - everything in between is possible
- Changes can be made on-the-fly
- Example:
  - one (or more) central DHCP service
  - if needed, department can run their own service
  - problem: selection of “more specialized” service

Integration of Existing Systems

- Direct access of legacy management standards (SNMP, OSI/TMN-Mgmt., CIM/WEBM, ...), as well as non-management distributed systems (CORBA) through Java
- CORBA $\leftrightarrow$ SNMP gateway exists
- CORBA $\leftrightarrow$ Jini is easy
  - Jini doesn't care about the protocol spoken between service and client
  - CORBA gateway has to be extended by a Jini layer for:
    - registering with lookup service
    - maybe supplying a GUI

Drawbacks & Problems

- Every component needs a JVM or a proxy
  - JVMs currently require large amounts of memory
  - not only (management and “regular”) services, but also network elements
- Using large numbers of services:
  - lookup service able to handle large number of services?
  - lease renewal requires significant amount of bandwidth?
- Security
  - yet unsolved
  - which service is allowed to talk to which?
  - who is allowed to start services?
  - who is allowed to administer services?

Using Jini Concepts

- Jini combines a number of concepts
  - leases
  - distributed events
  - lookup & join
  - transactions
- Show deployment areas of concepts
- Concepts can be used without Jini

Leases

- Alternative to pinging
  - e.g. equipment supervision
  - object migration / RMI needed
  - additional feature: cancellation
  - both parties agree on duration
- Resource handling
  - lease-based access to limited resources
  - resources can be reclaimed and redistributed after known time
- Charging
  - service usage can be metered by lease duration
  - only time-based usage

Distributed Events

- Equipment supervision
  - emitted under certain conditions, e.g. overheated, excessive bit error rate, security alert, ...
- Notifications
  - like notifications in CMIP (or traps in SNMP, but typed data)
  - usually registration required ($\rightarrow$ no unsolicited notifications)
  - registering involves moving an object
    - object can be simple RMI object
    - object might have further functionality, e.g. filter events at the source, send to different recipient depending on type, ...

Lookup & Join

- Service localization
- Makes software components realizable
  - extensibility at run-time
  - service migration
  - component updates
  - fault tolerance & scalability by redundancy

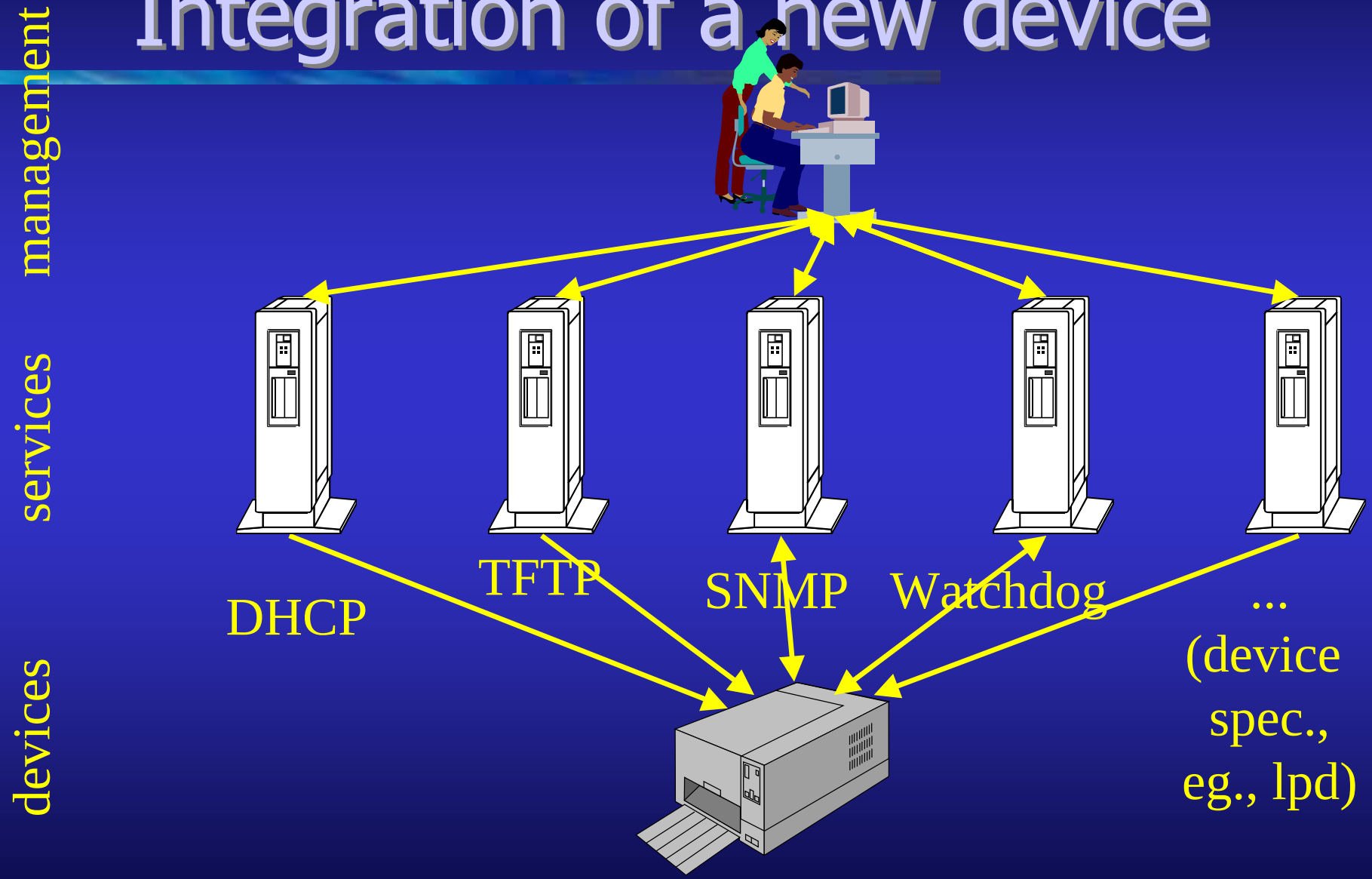
Transactions

- Atomicity most important
  - a certain number of actions has to be completed
  - “all or nothing” semantics
  - probably distributed among a number of machines
- Examples:
  - establishing a connection via a number of switches
    - if no resources left at one switch the whole connection fails
    - already allocated resources at other switches can be released
  - coordinated software updates
    - update changes protocol of communicating machines
    - all machines must be updated “simultaneously”
    - update must be reversible (at least until end of transaction)

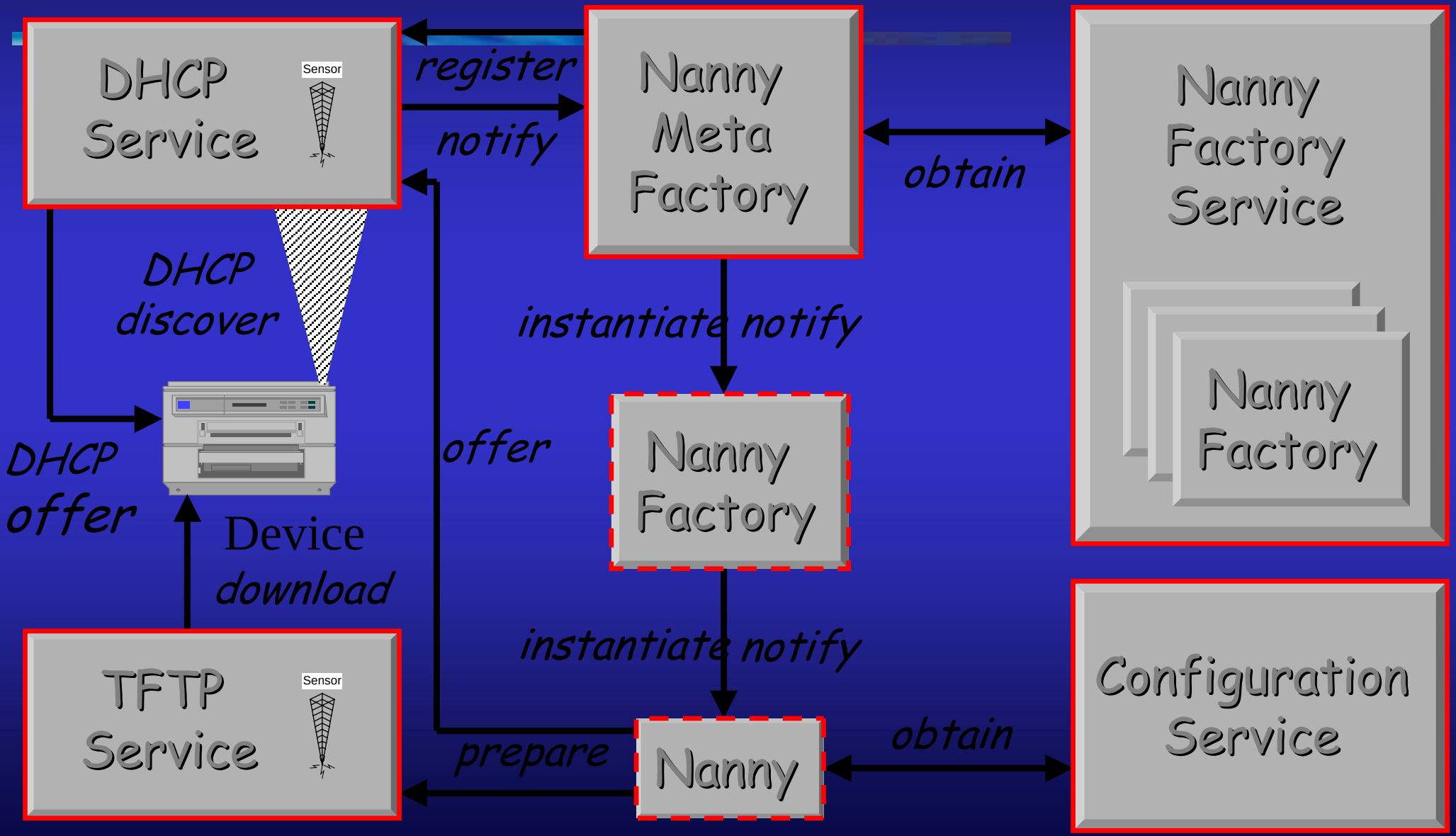
Example: Nannies

- Shows:
  - extensibility: new factories, sensors, etc.; components find each other: no configuration needed
  - scalability: as many factories, sensors etc. as you like/need
  - migration, updates
  - separation of concerns: simple services, each focussed on one functionality

Addressed example problem: Integration of a new device



Nanny-Architecture



Architecture Details

- Nannies
  - vendor-specific, device type-specific
  - provide intelligence for the integration of the new device
  - *go to Vendor's Web-portal to obtain NannyFactory*
- NannyFactory, NannyFactoryService
- NannyMetaFactory, a Jini-enabled service
- Network Device Bay
- Sensors and Binders
- Integrate existing configuration service
- Events/Notifications
  - DeviceUnknown
  - DeviceConnect
  - DeviceDisconnect
  - DeviceTimeout

Jini-based management

- Split up functionality into small components (separation of concerns)
- Reduce services to core functionality, e.g., DHCP protocol engine
  - address management
  - DHCP lease management
  - + Nannies
  - + Configuration service
  - + Persistence service
  - + Topology service
  - + ...

- Loosely coupled services
- Aggregate on demand
- Allow for re-implementation of service semantics, e.g., policy driven address assignment in DHCP
- Integrate non-Jini (management) services
- Integrate low level configuration into Jini

Jini-based management federation (realization)

- Split up functionality of services, e.g.,
  - Sensor, vs.
  - Binder, vs.
  - Higher order semantics
- Identify meaningful events (state changes) + interfaces
- Nannies
- Use and integrate existing services, e.g., configuration management.

Example 5: **DHCPSensor**

- A **DHCPSensor** detects DHCP messages on a network and forwards them to interested listeners, e.g., a Nanny factory

```
package de.tud.cs.jini.mgmt.dhcpservice;
public class DHCPSensor
    extends UnicastRemoteObject
        implements ServiceIDListener,
                                Sensor,
                               Landlord
{
    public SensorRegistration register(
            EventType theEvent,
            RemoteEventListener informMe,
            java.rmi.MarshalledObject handback,
            long leaseRequest)
        throws RemoteException
    {
        ...
```

- Jini Service
- Sensor (abstract interface)
- Grants leases for Registrations

Events on the network are propagated through the Jini federation as Jini Remote Events

Jini-based management federation (cont.)

- Define interfaces for managed (management) services
- Provide additional services:
  - Console (multiple operators)
  - Protocol gateways
    - management protocols
      - CORBA, SNMP (SNMP/CORBA gateway, JIDM)
      - CMIP (JIDM?),
      - WEBM/CIM (??)
    - arbitrary network protocols: DHCP, TFTP, LPD, SMTP, HTTP (proxies), application level, ...
  - Monitoring
  - ...

Excursion: Jiro and FMA (1)

- Suns Jiro is/was a Jini-based "management architecture" for Storage Area Networks (SAN)
- But devolved to a test field for yet another multi-purpose management architecture ...
- ... the Federated Management Architecture (FMA)
  - 3-tiered architecture (again)
    - Clients
    - Services
    - Resources
  - Information Model: CIM-based
  - Strongly coupled to Java RMI
- "Dynamic Management Services"
  - represented by their interface
  - hosted by Management "Stations"
  - dynamic class loading (JARs)
  - dynamic binding
    - "Referents" (target object)
    - "Acceptors" (make a Referent remotely accessible)
    - "Proxies" (access a "Referent" object through its Acceptor)
    - "Client"
  - Proxies and Acceptors with automatic rebinding

Excursion: Jiro and FMA (2)

- Enhanced Security Concept!!!
- Transaction-oriented
- Covers important issues of distributed systems
  - Concurrency
  - Synchronization, Deadlocks
  - Partial failure
  - Distributed state, Contexts
  - Persistence
  - Exception handling (nested)
- Base services:
  - Controller service
  - Log service
  - Event service
  - Scheduling service

Excursion: Jiro and FMA (3)

- Backed by many SAN-vendors, which have running prototypes:
  - StorageTEK
  - Exabyte
  - Hitachi
  - Fujitsu
  - Veritas
  - Legato
  - Ancor

Standardization

- Standardization is necessary to allow for different vendors to integrate their solutions
- Jini-based management is currently an upcoming technology
- Sun
  - currently provides three different approaches for Java-based management:
    - Java Management eXtensions (JMX)
    - Java Device Management Kit (JDMK)
    - Jiro/Federated Management Architecture (FMA)
  - which partly overlap
  - has failed with a former approach (JMAPI, which has partly merged to JMX

Summary/Conclusion

- Jini is an interesting new technology
  - runtime extensibility, fault tolerance, scalable
  - requires management
    - self-configuration („Zero-Administration“) vs.
    - external administration
  - provides new options and possibilities for management
- Must be integrated with legacy and proprietary environments
- Needs standardization
- New challenges for the management community

Suggested Reading (1)

- AdventNet Agent Toolkit; AdventNet, Inc.; 1999 (`http://www.adventnet.com/products/agentbuilder/`)
- Java Dynamic Management Kit, A White Paper; Sun Microsystems, Inc.; 1998 (`http://www.sun.com/software/java-dynamic/wp-jdmk/`)
- Java Management Extensions (JMX); Sun Microsystems, Inc. (`http://java.sun.com/products/JavaManagement/`)
- Jini Technology and the Java Dynamic Management Kit Demonstration; Sun Microsystems, Inc.; 1999 (`http://www.sun.com/software/java-dynamic/wp-jdmk.kit/`)

Suggested Reading (2)

- Jiro/Federated Management Architecture (FMA) Specifications; Sun Microsystems, Inc.; 1999 (`http://www.jiro.com/specs.html`)
- The ServiceUI Project; Bill Venners; Feb/2000; (`http://www.artima.com/jini/serviceui/`)

Suggested Reading (3)

- Gerd Aschemann, Svetlana Domnitcheva, Peer Hasselmeyer, Roger Kehr, Andreas Zeidler: A Framework for the Integration of Legacy Devices into a Jini Management Federation. Proc. DSOM'99, Springer-Verlag, pp. 257-268, 1999 (`http://www.informatik.tu-darmstadt.de/VS/Publikationen/`)
- Gerd Aschemann, Roger Kehr, Andreas Zeidler: A Jini-based Gateway Architecture for Mobile Devices. Proc. JIT'99, Springer-Verlag, pp. 203-212, 1999 (`http://www.informatik.tu-darmstadt.de/VS/Publikationen/`)