

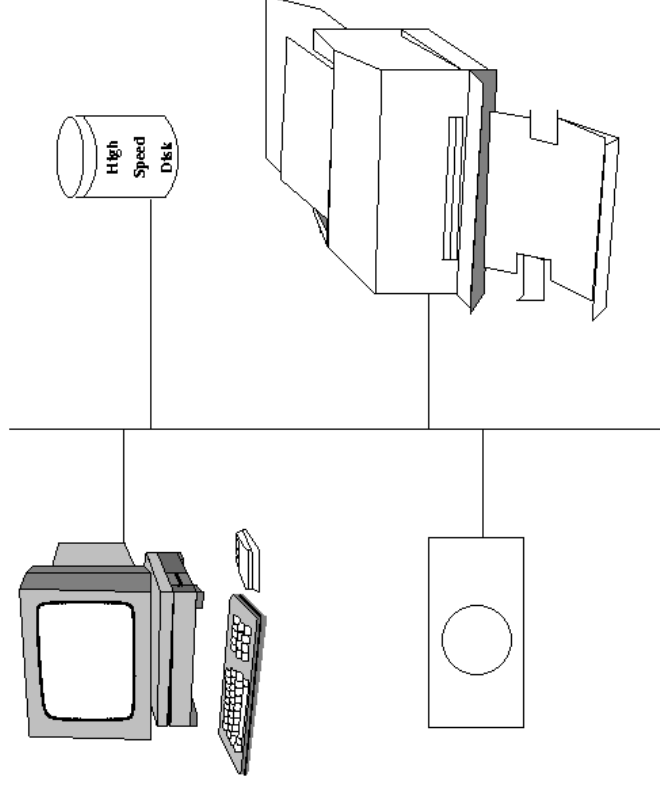
## A tutorial on Jini

Jan Newmarch  
jan@ise.canberra.edu.au  
University of Canberra  
Monash University

## OVERVIEW

## What is Jini?

- Jini brings "network plug and play" to hardware devices and software services
- Jini is in pure Java
- It makes hardware and software available as Java networked services
- "Legacy" systems can be wrapped in a Java proxy
- Jini has competitors including CORBA Trader, DCOM, HP's Chai, Enterprise Java Beans, JNDI, etc



## Jini Federation

- A Jini federation or "djinn" consists of a number of clients, services and service locators
- Typically connected by TCP/IP
- Code is moved around from service to locator to client using Java 1.2 extensions to RMI

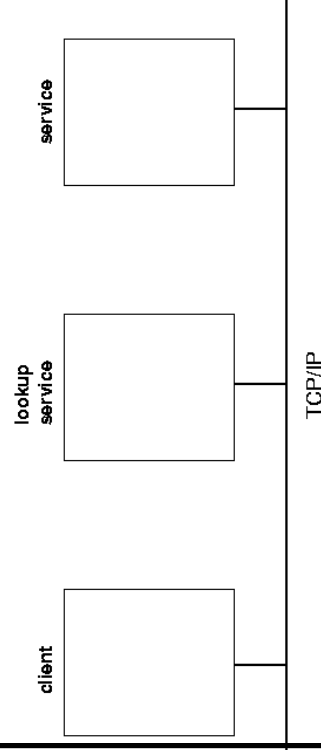


Figure 1: Components of a Jini federation

## Service Registration

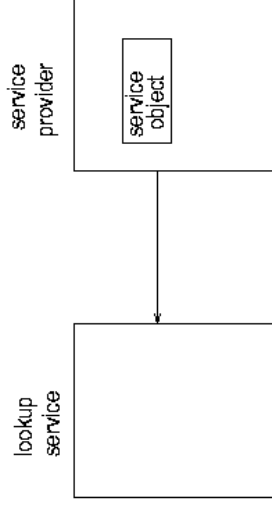


Figure 2: Querying for a service locator

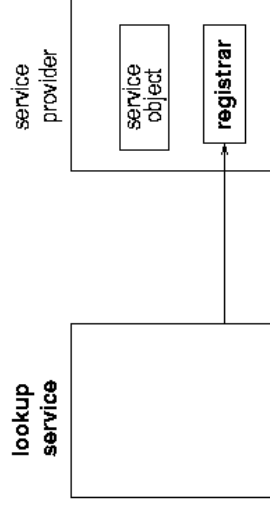


Figure 3: Registrar returned

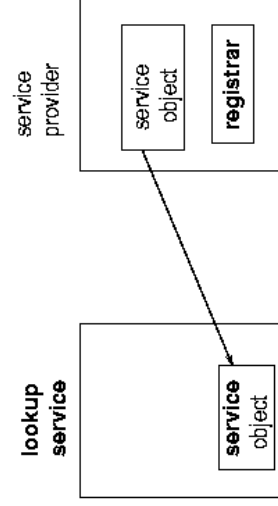


Figure 4: Service uploaded

## Client Lookup

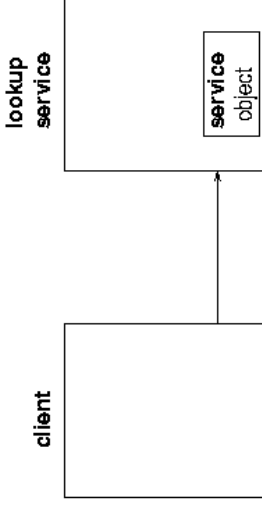


Figure 5: Querying for a service locator

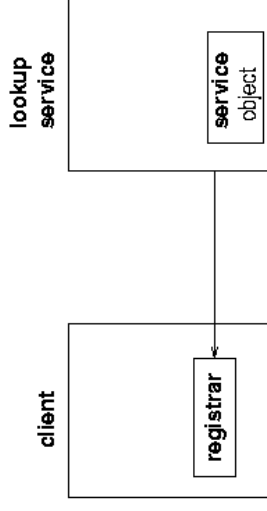


Figure 6: Registrar returned

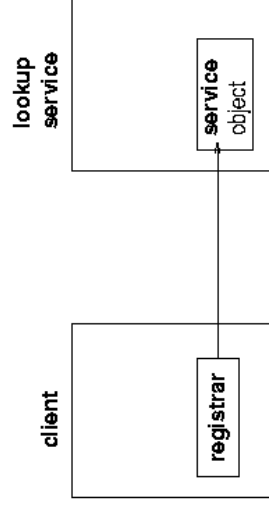


Figure 7: Asking for a service

## Support Services

- Jini objects are moved in "marshalled" form
- Class files may need to follow these for proper activation
- The class files may need to be available from HTTP servers
- Services may use RMI Activation and may need to have rmid running
- Security permissions will need to be set using JDK 1.2 security model

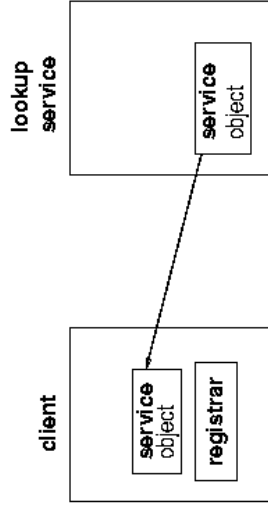
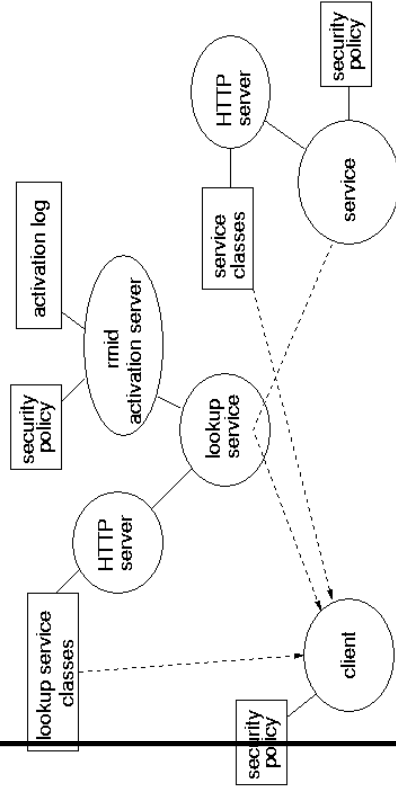
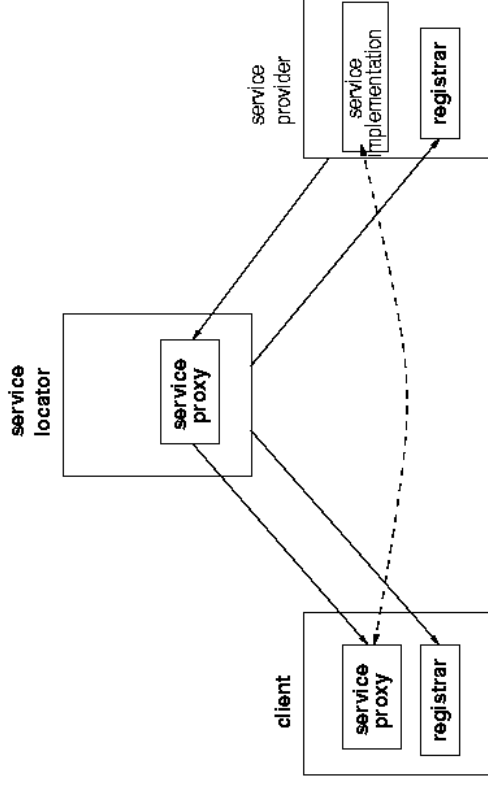


Figure 8: Service returned



## Proxies

- A toaster will be controlled by some service software
- The service software will be moved across the network to the client
- The mobile code must be a proxy for the real service code
- The proxy may be explicitly written or be an RMI proxy stub
- Remote calls from the proxy may be RMI Remote calls, or by some other mechanism such as private TCP connection



## Attribute Registration

- A service is uploaded to a locator as an instance of a class
- Additional information such as address may also be uploaded as instances of Entry classes
- Service registration is done using a leasing system

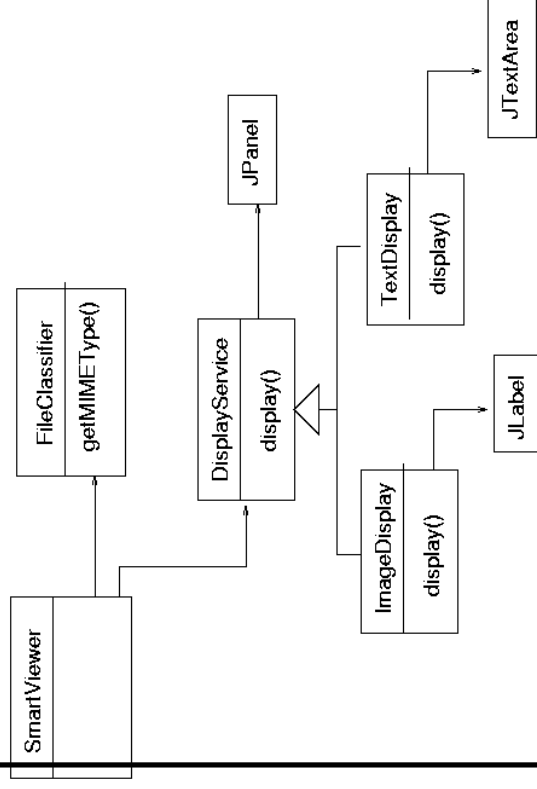
## Service Location

- A client asks for a service as an instance of a class, typically of an interface
- A client may ask for additional Entry attributes
- Matching on fields is *exact*
- You can ask for a printer with a certain speed, but not for one *greater than* a certain speed
- Additional services may be used if complex boolean expressions are needed

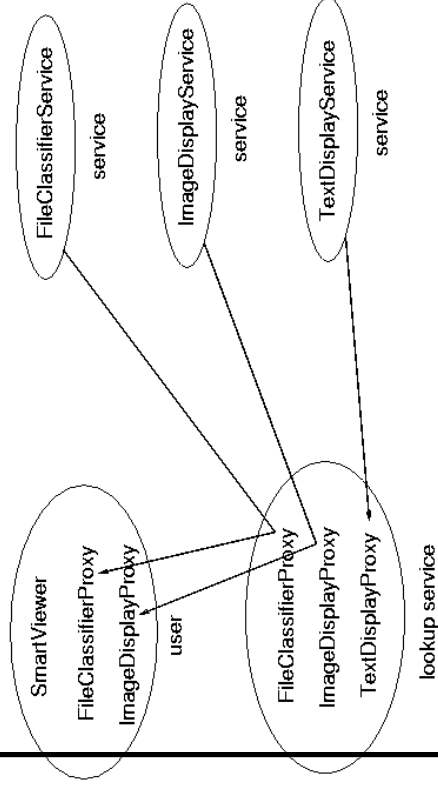
## Leasing

- Because services can come and go, network connections can be lost, etc, a distributed garbage collection mechanism is needed
- Services which register are granted a lease
- Registration lasts for as long as the lease
- When the lease expires, the service is removed
- Leases can be renewed

## Non-distributed Application



## Distributed Version



## Client Structure

Internally a client will look like

- prepare for discovery
- discover a lookup service
- prepare a template for lookup search
- lookup a service
- call the service

## Server Structure

- prepare for discovery
- discover a lookup service
- create information about a service
- export a service
- renew leasing periodically

## DISCOVERING A LOOKUP SERVICE

### Multicast discovery

- Broadcast discovery is done using the class `LookupDiscovery`

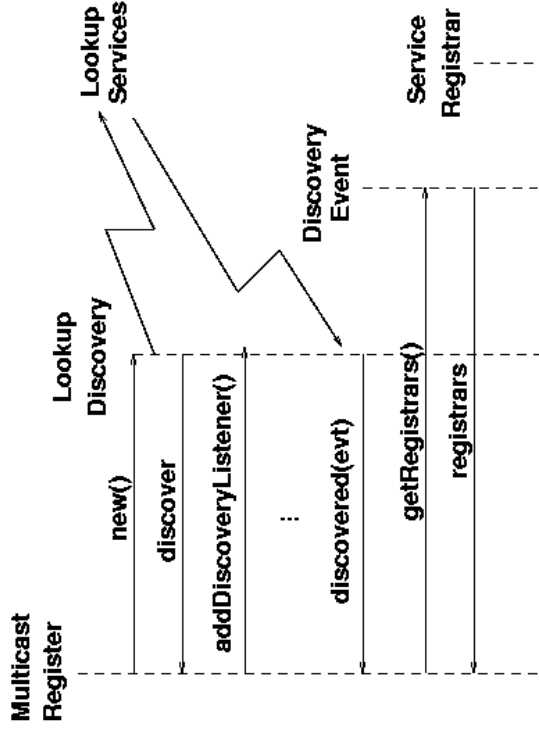
```
public class LookupDiscovery {  
    public LookupDiscovery(String[] groups);  
  
    addDiscoveryListener (LookupDiscoveryListener l);  
}
```

- Replies are handled asynchronously by a listener  

```
interface LookupDiscoveryListener {  
    public void discovered(DiscoveryEvent e);  
    public void discarded(DiscoveryEvent e);  
}
```

- The event contains registrar information  

```
public class DiscoveryEvent {  
    Registrar[] getRegistrars();  
}
```



## Multicast discovery program

```
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;

public class MulticastRegister implements DiscoveryListener {

    static public void main(String argv[]) {
        new MulticastRegister();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch (java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public MulticastRegister() {
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch (Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();
        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];

            // the code takes separate routes from here for client or service
            System.out.println("found a service locator");
        }

        public void discarded(DiscoveryEvent evt) {
        }

    } // MulticastRegister
}
```

## ServiceRegistrar

- Unicast and multicast lookup methods return a ServiceRegistrar. This is a proxy for the lookup service

```
public interface ServiceRegistrar {
    public ServiceRegistration register(ServiceItem i,
                                     long leaseDuration);

    public Object lookup(ServiceTemplate tmpl);
    public ServiceMatches lookup(ServiceTemplate tmpl, int maxMatches);
}

LookupLocator getLocator();
```

- The register() method is used by a *service*
- The lookup() methods are used by a *client*. The first returns a single service, the second a set of services
- The getLocator() method can say where the lookup service is

## ENTRIES

## Entry class

- Entries are additional information about a service, such as location, name, cost
- This information is not part of the service object itself
- The additional information can be used to select between otherwise identical services  
e.g. a *colour* printer, a *four-slice* toaster
- Entries implement Entry

```
public interface Entry {}
```

- There are a bunch of convenience implementations such as
  - Address
  - Comment
  - Name
  - ServiceInfo

## Creating entries

- You can create your own classes, or use the given ones
- Typical code

```
Entry[] entries = new Entry[] {new Comment("best toaster made"),  
                                new Name("Classy Toaster")};
```



## Interfaces, implementations and entries

- A client looks for an instance of an interface
- It may qualify the search with entries
- A service exports an implementation of this interface
- A service will also export additional entry information, as much as it can

## CLIENT

## Problem domain

- Files have a type, that can be described by a MIME type

```
public class MIMEType implements java.io.Serializable {
    protected String contentType;
    protected String subType;

    public MIMEType(String contentType, String subType) {...}

    public String toString() {...}
}
```

(It may need to be serialized, to move it from one machine to another)

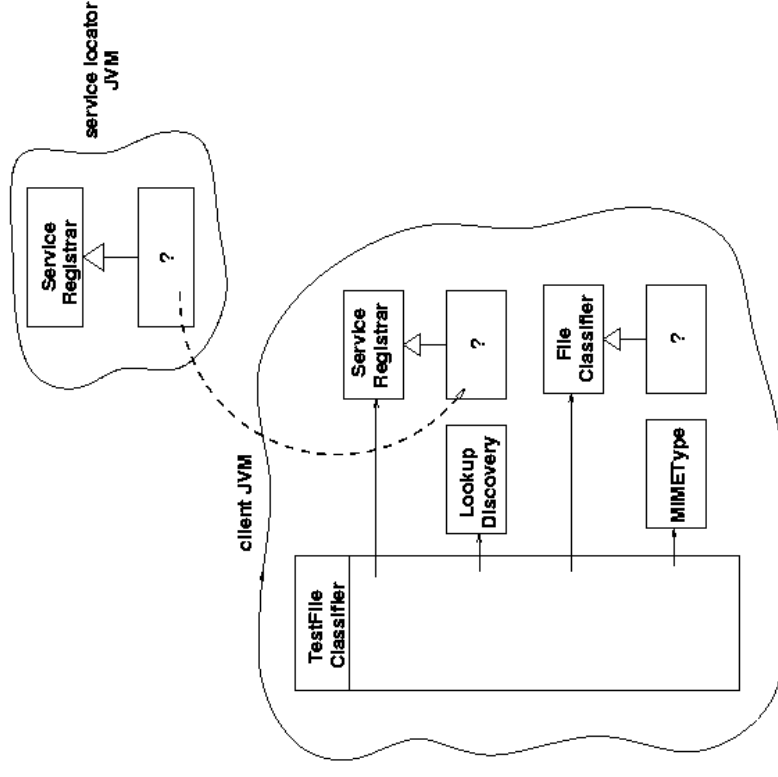
- A file classifier can take a filename and use this to work out the MIME type, using an internal table

```
public interface FileClassifier {
    MIMEType getMIMEType(String filename)
    throws java.rmi.RemoteException;
}
```

(The method may be executing remotely, so can throw a remote exception)

## Client and service locator JVM's

The two JVM's look like



## Multicast client

Use this pattern when you don't know where the lookup locator(s) are:

```
package client;

import common.FileClassifier;
import common.MIMETYPE;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;

public class TestFileClassifier implements DiscoveryListener {

    public static void main(String argv[]) {
        new TestFileClassifier();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch (java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestFileClassifier() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch (Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {FileClassifier.class};
```

```
FileClassifier classifier = null;
ServiceTemplate template = new ServiceTemplate(null, classes,
    null);

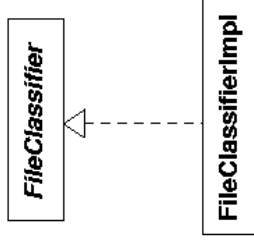
for (int n = 0; n < registrars.length; n++) {
    System.out.println("Service found");
    ServiceRegistrar registrar = registrars[n];
    try {
        classifier = (FileClassifier) registrar.lookup(template);
    } catch (java.rmi.RemoteException e) {
        e.printStackTrace();
        System.exit(2);
    }
    if (classifier == null) {
        System.out.println("Classifier null");
        continue;
    }
    MIMETYPE type;
    try {
        type = classifier.getMIMETYPE("file1.txt");
        System.out.println("Type is " + type.toString());
    } catch (java.rmi.RemoteException e) {
        System.err.println(e.toString());
    }
    // System.exit(0);
}

public void discarded(DiscoveryEvent evt) {
    // empty
}
} // TestFileClassifier
```

## PROXY IMPLEMENTATION CHOICES

## Proxy is the Service

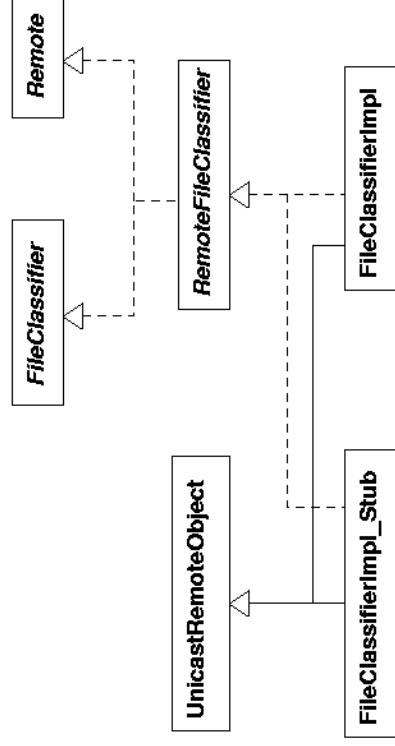
- The complete service may be uploaded, so there is no separate proxy



- The client uses `FileClassifier`
- The server creates a `FileClassifierImpl`
- The server exports the `FileClassifierImpl`
- This is a “fat” proxy model, with no processing being done on the server side
- This is suitable for software services that do not need to maintain state on the server side

## RMI Proxies

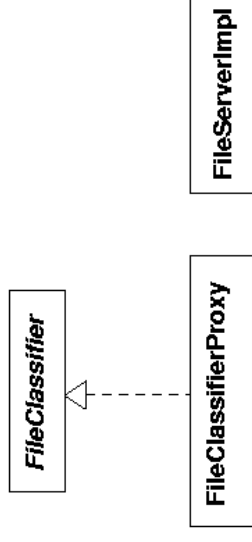
- If RMI stubs are used to implement the proxy, it looks like



- The client uses `FileClassifier`
- The server creates a `FileClassifierImpl`
- The RMI runtime ensures that a `FileClassifierImpl_Stub` is exported to the service locator
- This is a "thin" proxy, with no processing being done on the client side
- This is closest to the standard RPC model between client and service

## Non-RMI Proxy

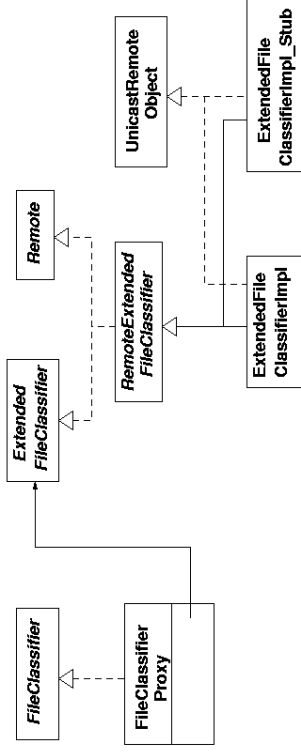
- If RMI is not used, the proxy must be explicitly created



- The client uses `FileClassifier` (like before - it doesn't care what the proxy is)
- The server creates a `FileClassifierImpl` and a `FileClassifierProxy`
- The server exports a `FileClassifierProxy`
- The proxy and service communicate using some protocol
- This can have processing done on both sides
- This is good for client-server problems communicating on their own ports with their own protocol. The proxy is the heart of the client

## RMI and Non-RMI Proxies

- You can use both RMI and a non-RMI proxy



- The client uses **FileClassifierProxy**
- The server creates a **ExtendedFileClassifierImpl** and a **FileClassifierProxy**
- The server exports a **FileClassifierProxy** and an **ExtendedFileClassifierImpl\_Stub**
- The proxy uses the stub locally (on the client), which uses RMI to talk back to the service
- This can have processing done on both sides, using an RPC-like model for communication

## PROXY IS THE SERVICE

## Service implementation

```
package option2;

import common.MIMETYPE;
import common.FileClassifier;

public class FileClassifierImpl implements FileClassifier {

    public MIMETYPE getMIMETYPE(String fileName) {
        if (fileName.endsWith(".gif")) {
            return new MIMETYPE("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMETYPE("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMETYPE("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMETYPE("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMETYPE("text", "html");
        } else
            // fill in lots of other types,
            // but eventually give up and
            return null;
    }

    public FileClassifierImpl() {
        // empty
    }
} // FileClassifierImpl
```

## File classifier server

```
package option2;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import com.sun.jini.lease.LeaseRenewalManager;
import com.sun.jini.lease.LeaseListener;
import com.sun.jini.lease.LeaseRenewalEvent;

public class FileClassifierServer implements DiscoveryListener,
    LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new FileClassifierServer();

        // keep server running forever to
        // - allow time for locator discovery and
        // - keep re-registering the lease
        try {
            Thread.currentThread().sleep(Lease.FOREVER);
        } catch (java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public FileClassifierServer() {
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch (Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
```

```

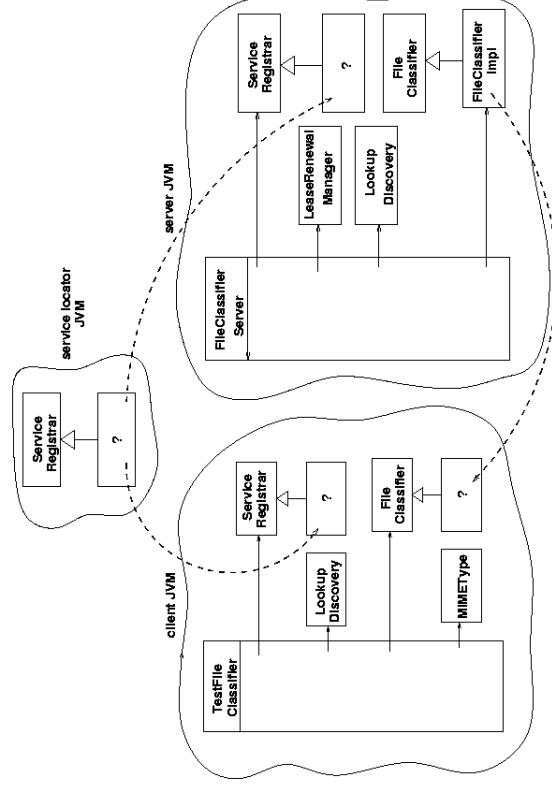
ServiceRegistrar[] registrars = evt.getRegistrars();
for (int n = 0; n < registrars.length; n++) {
    ServiceRegistrar registrar = registrars[n];
    ServiceItem item = new ServiceItem(null,
        new FileClassifierImpl(),
        null);
    ServiceRegistration reg = null;
    try {
        reg = registrar.register(item, Lease.FOREVER);
    } catch (java.rmi.RemoteException e) {
        System.err.println("Register exception: " + e.toString());
    }
    System.out.println("service registered");
    // set lease renewal in place
    leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
}

public void discarded(DiscoveryEvent evt) {
}

public void notify(LeaseRenewalEvent evt) {
    System.out.println("Lease expired " + evt.toString());
}
} // FileClassifierServer

```

## All the JVM's





## SECURITY

### JDK 1.2 Security

- Jini security uses the security model of JDK 1.2, with no changes or extensions
- Security is based on granting permissions (access rights), in security files
- Debug the application with a slack policy

```
grant {  
    permission java.security.AllPermission " ", " " ;  
};
```

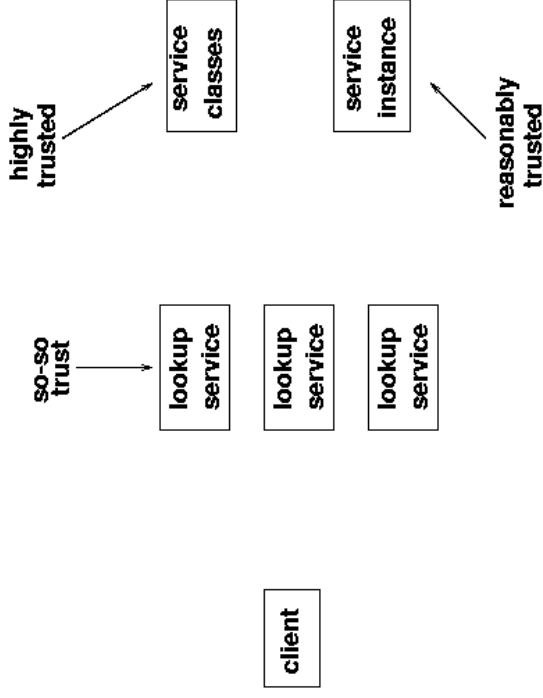
- Leave it like this of production only at your peril

## Service requirements

```
grant {  
    permission net.jini.discovery.DiscoveryPermission "*";  
    // multicast request address  
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";  
    // multicast announcement address  
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";  
  
    // RMI connections  
    permission java.net.SocketPermission "*.dstc.edu.au:1024-", "connect,accept";  
    permission java.net.SocketPermission "130.102.176.249:1024-", "connect,accept";  
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,accept";  
  
    // reading parameters  
    // like net.jini.discovery.debug!  
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";  
};
```

## Client requirements

The client is most at risk as it imports remote objects into its address space



## Client policy

```
grant {
    permission net.jini.discovery.DiscoveryPermission "***;
    // multicast request address
    permission java.net.SocketPermission "224.0.0.1.85", "connect,accept";
    // multicast announcement address
    permission java.net.SocketPermission "224.0.0.1.84", "connect,accept";

    // RMI connections
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,accept";
    permission java.net.SocketPermission "*.dstc.edu.au:1024-", "connect,accept";
    permission java.net.SocketPermission "130.102.176.249:1024-", "connect,accept";

    // DANGER
    // HTTP connections - this is where external code may come in - careful!!!
    permission java.net.SocketPermission "127.0.0.1:80", "connect,accept";
    permission java.net.SocketPermission "*.dstc.edu.au:80", "connect,accept";

    // reading parameters
    // like net.jini.discovery.debug!
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
};
```

## REMOTE EVENTS

## Event models

- Java has several event models
- Input events are generated by user actions such as key press, mouse click. These are placed into an event queue by one thread, removed by another
- Semantic events are generated by GUI components, such as mouse click in a Button generates an `ActionEvent`
- Property changes in objects generate `PropertyChangeEvent` for Java beans
- JNI events are designed to work in a network environment
- Events may be lost
- Event delivery may have time delays, and state information may be out of date when it arrives
- A listener may have disappeared or be unreachable, so listeners need to be on leases
- Multiple event types (such as AWT events) require class availability while simple types don't

## Remote events

- Event listeners must implement

```
public interface RemoteEventListener {
    public void notify(RemoteEvent evt) throws ...;
}
```
- Where

```
public class RemoteEvent {
    public long getID();           // distinguish event types
    public long getSequenceNumber(); // like a timestamp
    public MarshallableObject getRegistrationObject(); // handback
}
```
- There is no equivalent within JNI to `addActionListener()`, `addPropertyChangeListener()`, etc. Each object must look after its own listener list and decide when and how to call it. But...
- Registration of a remote event listener must return

```
public class EventRegistration {
    public EventRegistration(long eventId, Object source,
                             Lease lease, long seqNum);

    public long getID();
    public Object getSource();
    public Lease getLease();
    public long getSequenceNumber();
}
```

## A single listener list

A list with only one listener on it can be done by

```
protected RemoteEventListener listener = null;

public EventRegistration addRemoteListener(RemoteEventListener listener)
    throws java.util TooManyListenersException {
    if (this.listener == null {
        this.listener = listener;
    } else {
        throw new java.util TooManyListenersException();
    }
    return new EventRegistration(0L, this, null, 0L);
}
```

## Single listener event notification

The source object can send an event to its listener by

```
protected void fireNotify(long eventId,
                          long seqNum) {
    if (listener == null) {
        return;
    }

    RemoteEvent remoteEvent = new RemoteEvent(this, eventId,
                                              seqNum, null);
    listener.notify(remoteEvent);
}
```

## Mutable file classifier

Allow the file classifier to change its list of MIME type mappings, and notify listeners when it does so

```
package mutable;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.EventRegistration;
import java.rmi.RemoteException;
import net.jini.core.event.UnknownEventException;

import javax.swing.event.EventListenerList;

import common.MIMETYPE;
import common.MutableFileClassifier;
import java.util.Map;
import java.util.HashMap;

public class FileClassifierImpl extends UnicastRemoteObject
    implements RemoteFileClassifier {

    /**
     * Map of String extensions to MIME types
     */
    protected Map map = new HashMap();

    /**
     * Listeners for change events
     */
    protected EventListenerList listenerList = new EventListenerList();

    public MIMETYPE getMIMETYPE(String fileName)
        throws java.rmi.RemoteException {
        MIMETYPE type;
        String fileExtension;
        int dotIndex = fileName.lastIndexOf('.');
        if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
            // can't find suitable suffix
            return null;
        }
        fileExtension= fileName.substring(dotIndex + 1);
        type = (MIMETYPE) map.get(fileExtension);
        return type;
    }

    public void addType(String suffix, MIMETYPE type)
        throws java.rmi.RemoteException {

```

```
        map.put(suffix, type);
        fireNotify(ADD_TYPE);
    }

    public void removeMIMETYPE(String suffix, MIMETYPE type)
        throws java.rmi.RemoteException {
        if (map.remove(suffix) != null) {
            fireNotify(REMOVE_TYPE);
        }
    }

    public EventRegistration addRemoteListener(RemoteEventListener listener)
        throws java.rmi.RemoteException {
        listenerList.add(RemoteEventListener.class, listener);
        return new EventRegistration(0, this, null, 0);
    }

    /** Notify all listeners that have registered interest for
     *  notification on this event type. The event instance
     *  is lazily created using the parameters passed into
     *  the fire method.
     */
    protected void fireNotify(long eventId) {
        RemoteEvent remoteEvent = null;

        // Guaranteed to return a non-null array
        Object[] listeners = listenerList.getListenerList();

        // Process the listeners last to first, notifying
        // those that are interested in this event
        for (int i = listeners.length - 2; i >= 0; i -= 2) {
            if (listeners[i] == RemoteEventListener.class) {
                RemoteEventListener listener = (RemoteEventListener) listeners[i+1];
                if (remoteEvent == null) {
                    remoteEvent = new RemoteEvent(this, eventId,
                                                0L, null);
                }
                try {
                    listener.notify(remoteEvent);
                } catch (UnknownEventException e) {
                    e.printStackTrace();
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }
        }
    }

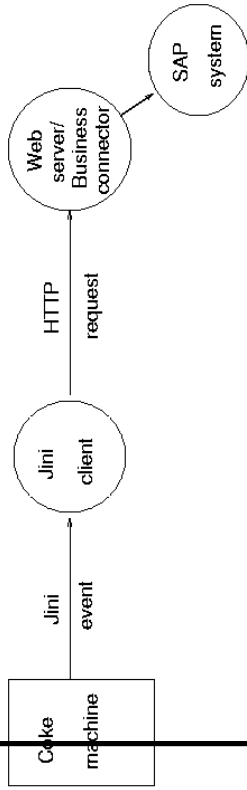
    public FileClassifierImpl() throws java.rmi.RemoteException {
        // load a predefined set of MIME type mappings
        map.put("gif", new MIMETYPE("image", "gif"));
        map.put("jpeg", new MIMETYPE("image", "jpeg"));
        map.put("mpg", new MIMETYPE("video", "mpeg"));
    }

```

```
map.put("text", new MIMETYPE("text", "plain"));
map.put("html", new MIMETYPE("text", "html"));
}
} // FileClassifierImpl
```

## Realistic Use of Events

- Interfacing a Coke machine to an SAP inventory system



## INTERFACING TO HARDWARE

## Lego MindStorms

- MindStorms is a “Robotics Invention System”
- A 16-bit computer called the RCX (a 16MHz Hitachi H8/3297), with 32k RAM
- The RCX can control 3 motors and read from 3 sensors
- Additional Lego parts - gears, wheels, sensors, motors
- Software can be downloaded by an infra-red link on a serial port
- Programs can be downloaded and run, or “immediate” commands can be run, sent on the IR link
- Normally programmed from a visual programming environment or from VB
- On a PC, interface to RCX is managed by an OCX
- Low-level machine-code is sent across the IR link



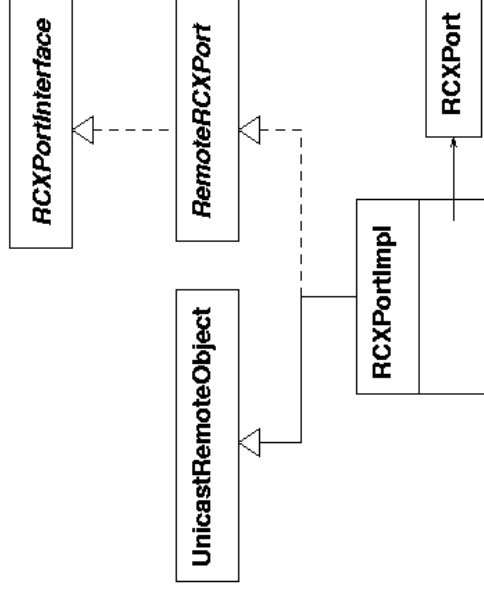
## RCX programmed from Java

- The RCX can be programmed using the IR link attached to the serial port
- Machine code is sent on this serial port
- Java has a standard extension to drive the serial port, the `CommAPI`
- Dario Laverde has a Java class above this, to encapsulate talking to the RCX

```
package rcx;

public class RCXPort {
    public RCXPort(String port);
    public void addRCXListener(RCXListener rl);
    public boolean open();
    public void close();
    public boolean isOpen();
    public OutputStream getOutputStream();
    public InputStream getInputStream();
    public synchronized boolean write(byte[] bArray);
    public void processRead();
    public String getLastError();
    public void showTable();
    public static byte[] parseString(String str);
}
```

## MindStorms as a Jini Service



## RCX Port Impl

This is best done as an RMI proxy to the service on the server. The service has an RCXPort as attribute, and passes on messages to it.

```
package rcx.jini;

import java.rmi.server.UnicastRemoteObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import rcx.*;
import java.io.*;
import java.util.*;

public class RCXPortImpl extends UnicastRemoteObject
    implements RemoteRCXPort, RCXListener {

    protected String error = null;
    protected byte[] message = null;
    protected RCXPort port = null;
    protected RemoteEventListener listener = null;
    protected long messageSeqNo, errorSeqNo;

    public RCXPortImpl()
        throws java.rmi.RemoteException {

        Properties parameters;
        String portName = null;
        File f = new File("parameters.txt");
        if (!f.exists()) {
            f = new File(System.getProperty("user.dir")
                + System.getProperty("path.separator")
                + "parameters.txt");
        }
        if (!f.exists()) {
            try {
                FileInputStream fis = new FileInputStream(f);
                parameters = new Properties();
                parameters.load(fis);
                fis.close();
                portName = parameters.getProperty("port");
            } catch (IOException e) {
            }
        } else {
            System.err.println("Can't find parameters.txt with \"port=...\" specified");
            System.exit(1);
        }

        port = new RCXPort(portName);
        port.addRCXListener(this);
    }

    public boolean write(byte[] byteCommands)
        throws java.rmi.RemoteException {
        return port.write(byteCommands);
    }
}
```

```
public byte[] parseString(String command)
    throws java.rmi.RemoteException {
    return RCXPort.parseString(command);
}

/**
 * Received a message from the RCX.
 * Send it to the listener
 */
public void receivedMessage(byte[] message) {

    this.message = message;

    // Send it out to listener
    if (listener == null) {
        return;
    }

    RemoteEvent evt = new RemoteEvent(this, MESSAGE_EVENT, messageSeqNo++, null);
    try {
        listener.notify(evt);
    } catch (net.jini.core.event.UnknownEventException e) {
        e.printStackTrace();
    } catch (java.rmi.RemoteException e) {
        e.printStackTrace();
    }
}

/**
 * Received an error message from the RCX.
 * Send it to the listener
 */
public void receivedError(String error) {
    // System.err.println(error);

    // Send it out to listener
    if (listener == null) {
        return;
    }
    this.error = error;
    RemoteEvent evt = new RemoteEvent(this, ERROR_EVENT, errorSeqNo, null);
    try {
        listener.notify(evt);
    } catch (net.jini.core.event.UnknownEventException e) {
        e.printStackTrace();
    } catch (java.rmi.RemoteException e) {
        e.printStackTrace();
    }
}

/**
 * Expected use: the RCX has returned a message,
 * and we have informed the listeners. They query
 * this method to find the message for the message
 * sequence number they were given in the RemoteEvent.
 * We could use this as an index into a table of messages.
 */
}
```

```

public byte[] getMessage(long msgSeqNo) {
    return message;
}

/**
 * Expected use: the RCX has returned an error message,
 * and we have informed the listeners. They query
 * this method to find the error message for the error message
 * sequence number they were given in the RemoteEvent.
 * We could use this as an index into a table of messages.
 */
public String getError(long errSeqNo) {
    return error;
}

/**
 * Add a listener for RCX messages.
 * Should allow more than one, or throw
 * TooManyListeners if more than one registers
 */
public void addListener(RemoteEventListener listener) {
    this.listener = listener;
    messageSeqNo = 0;
    errorSeqNo = 0;
}
} // RCXPortImpl

```

## CONCLUSION

## Problems and Issues

- Objects moved around must be serializable
- Some common objects are not serializable e.g. Swing Toolbar, Swing JTextArea
- Objects with DLLs are not serializable e.g. MPEG players
- JDK 1.2 memory requirements are enormous compared to JDK 1.1
- Jini v1.0 has some bugs
- Limited documentation (but several books in print, including mine from 2001)
- J2ME (micro edition Java) doesn't support Jini yet

## Conclusion

- Jini works
- The Jini programming model is cleaner than some others
- Hardware can be brought into the Jini world fairly easily
- Software services can be wrapped and brought into Jini
- Some things (like version maintenance in client/server) look much easier
- Speed and memory are problems
- Standard interfaces are required

## URLs

- Jini: <http://www.jini.org>
- Jini FAQ: <http://www.artima.com/jini/faq.html>
- Lego MindStorms: <http://www.legomindstorms.com>
- Unofficial Lego MindStorms: <http://www.crynr.com/lego-robotics/>
- Jini tutorial: <http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml>
- Motorola Piano: <http://www.mot.com/GSS/SSTG/piano/>

---

*Jan Newmarch (<http://pandonia.canberra.edu.au>)*

[jan@ise.canberra.edu.au](mailto:jan@ise.canberra.edu.au)  
Last modified: Mon Oct 16 13:53:55 EST 2000  
Copyright ©Jan Newmarch