A tutorial on Jini

Jan Newmarch
jan@ise.canberra.edu.au
University of Canberra
DSTC (Distributed Systems Technology CRC)

OVERVIEW

 © Jan Newmarch

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

 © Jan Newmarch

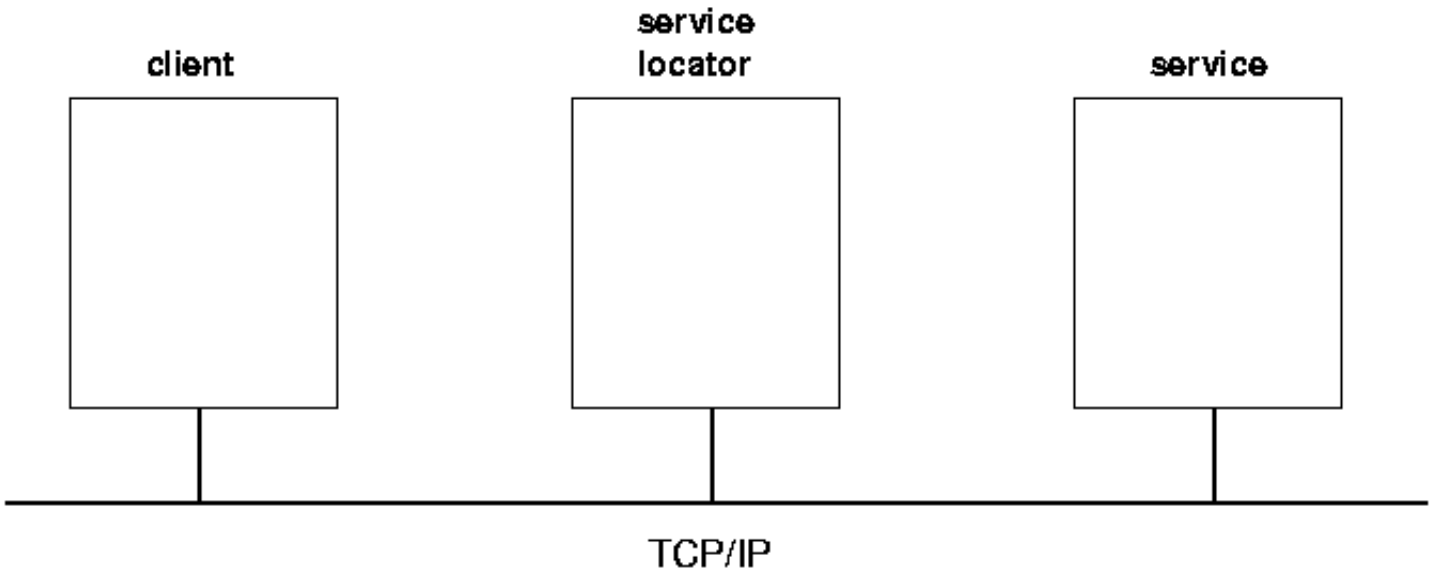


Figure 1: Components of a Jini federation

Service Registration

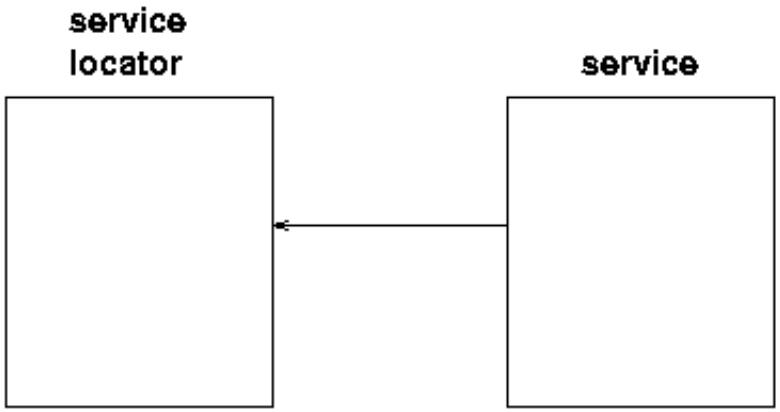


Figure 2: Querying for a service locator

 © Jan Newmarch

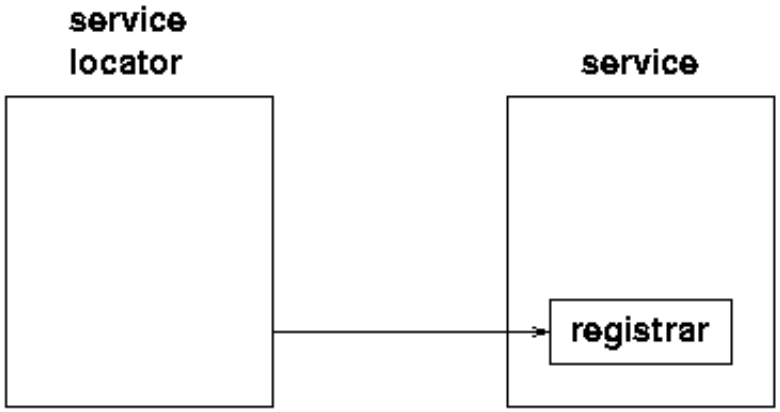


Figure 3: Registrar returned

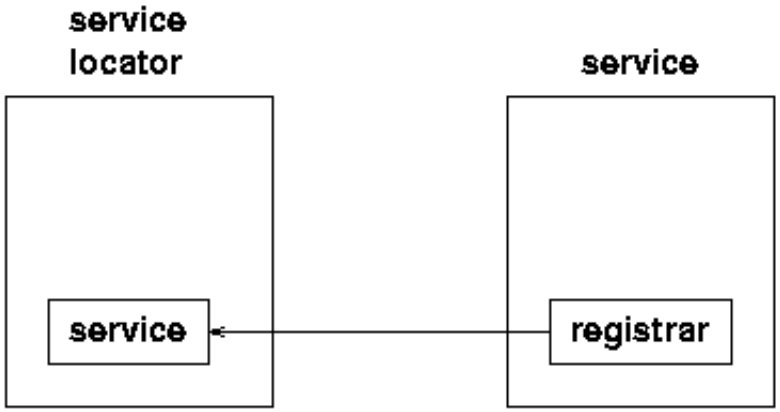


Figure 4: Service uploaded

Client Lookup

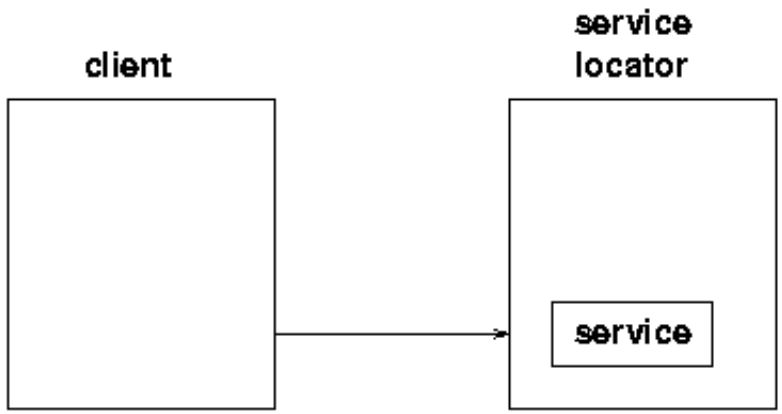


Figure 5: Querying for a service locator

 © Jan Newmarch

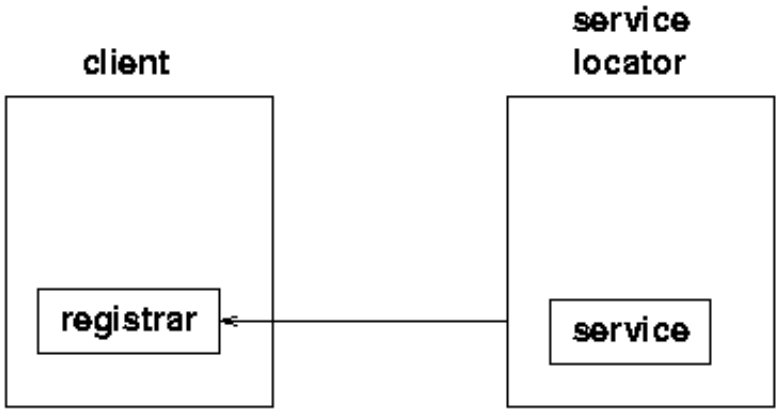


Figure 6: Registrar returned

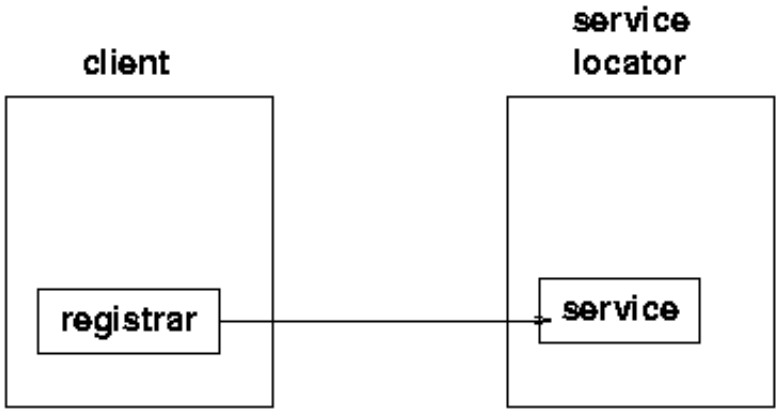


Figure 7: Asking for a service

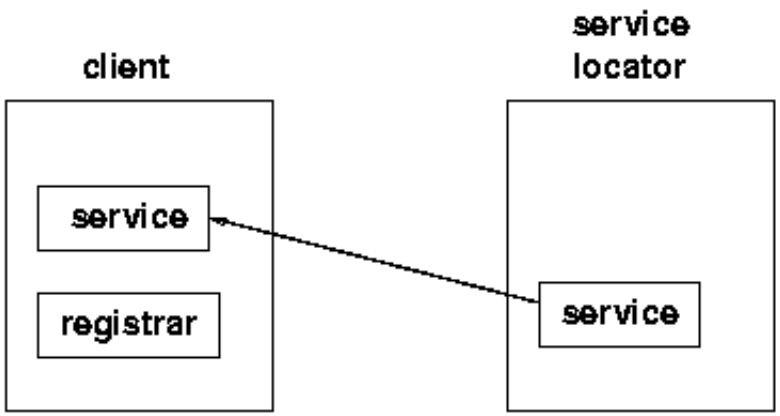


Figure 8: Service returned

 © Jan Newmarch

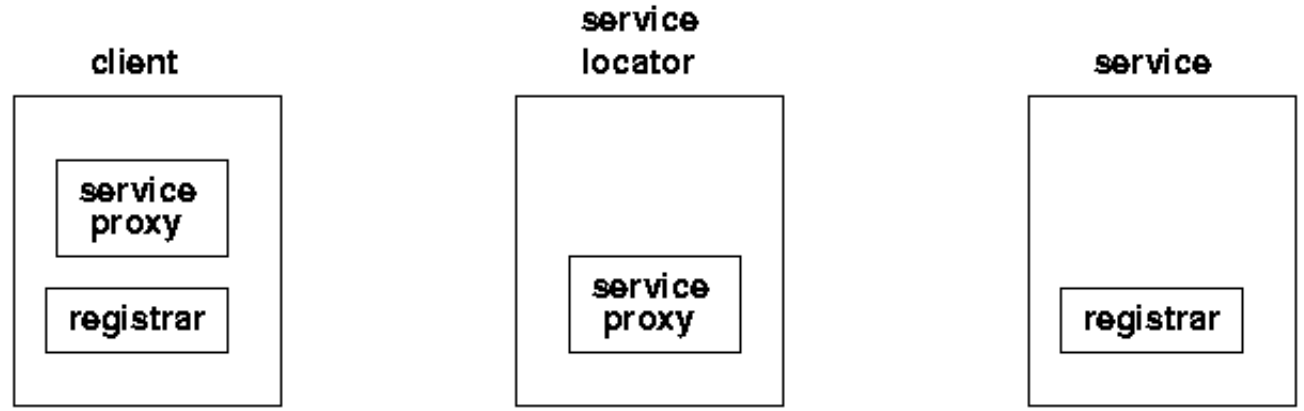
Support Services

- 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
- RMI Naming can be used to locate remote objects
- RMI Naming may need an rmiregistry or rmid running
- Security permissions will need to be set using JDK 1.2 security model

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

 © Jan Newmarch



Proxy Service

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

 © Jan Newmarch

Service Location

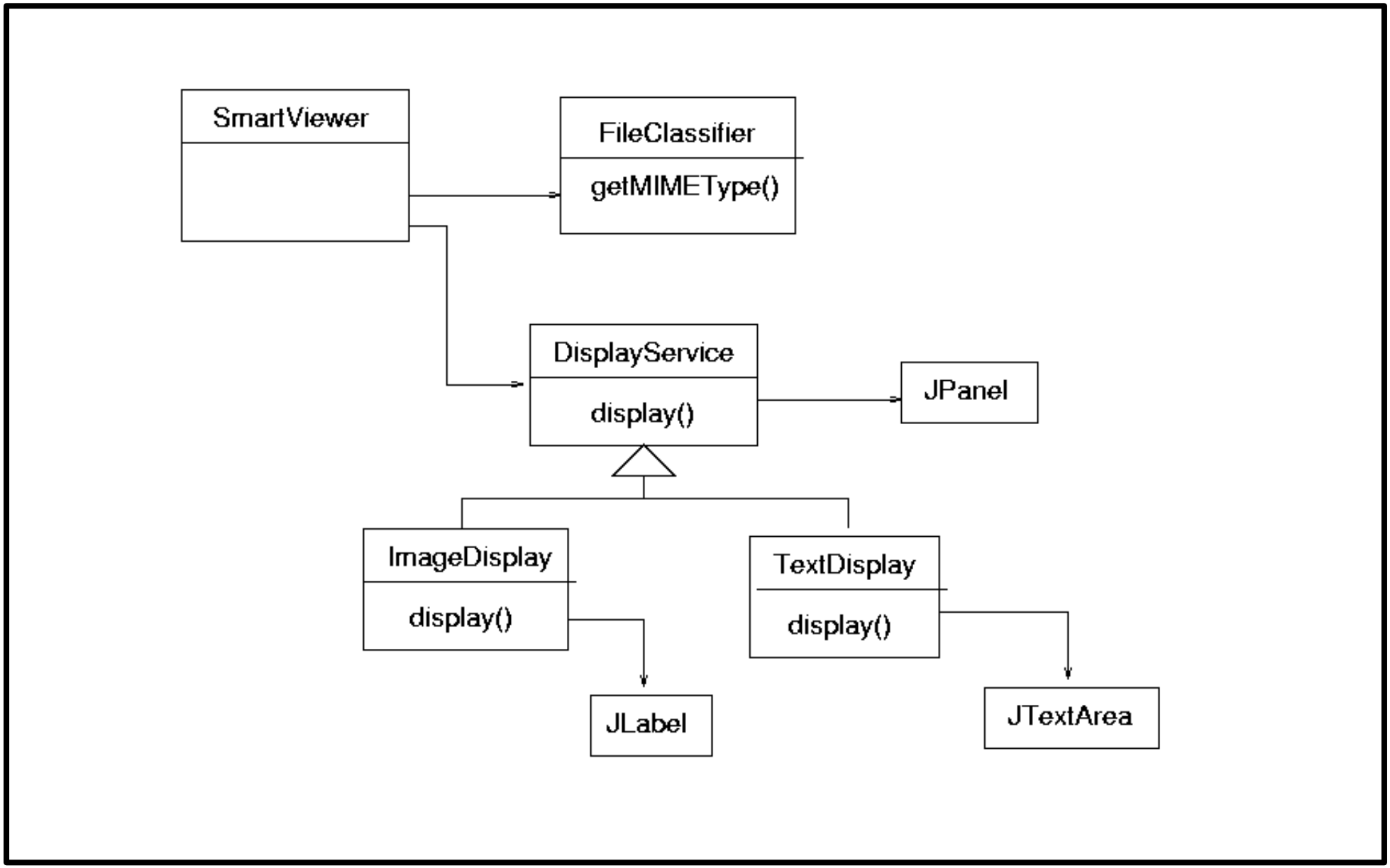
- A client asks for a service as an instance of a class, typically of an interface
- A client may ask for additional Entry attributes
- Matching on fields is *exact*
- You can ask for a printer with a certain speed, but not for one *greater than* a certain speed
- Additional services may be used if complex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

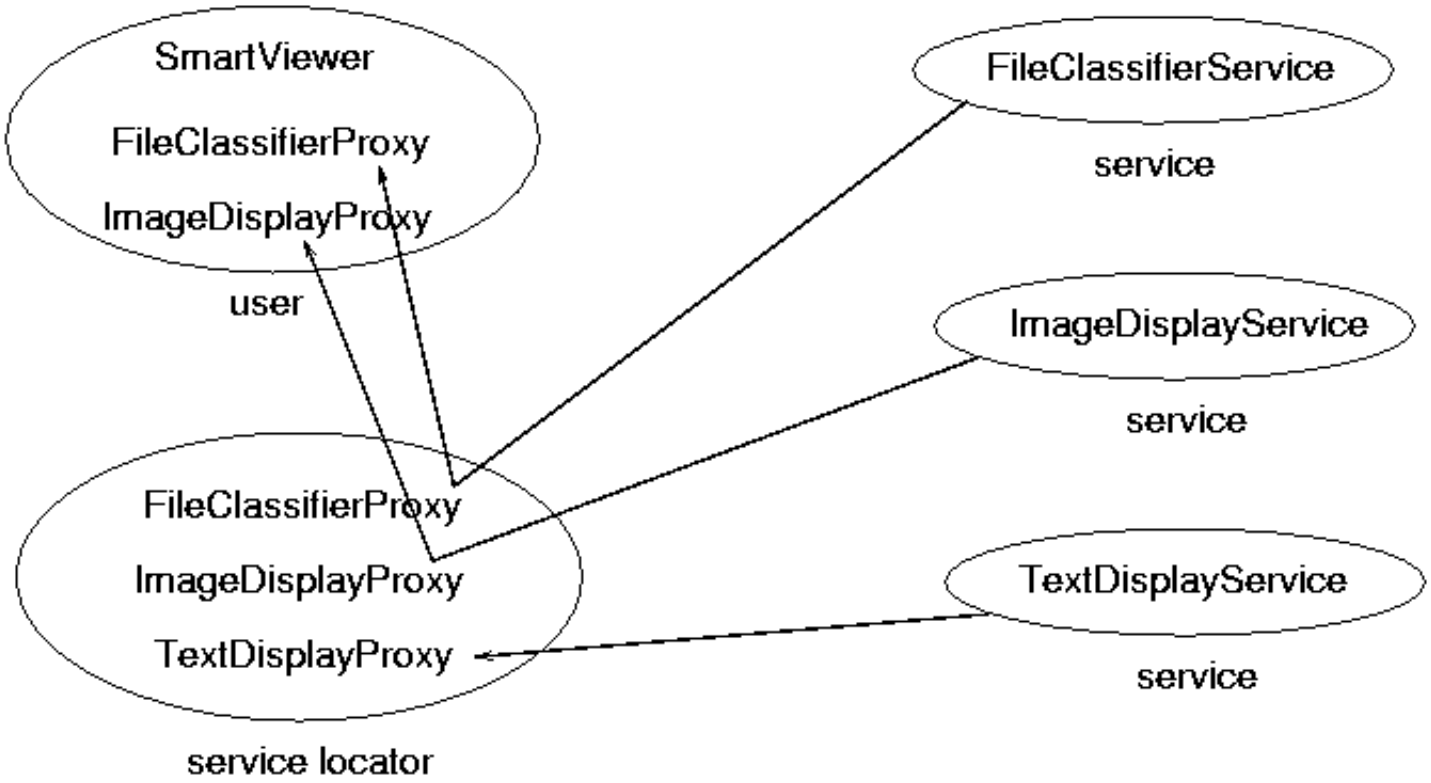
 © Jan Newmarch

Non-distributed Application



 © Jan Newmarch

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
```

The following code is simplified from the real case. It attempts to find a `FileClassifier` service, calling the method `getMIMEType()` on this service.

 © Jan Newmarch

```
public class TestUnicastFileClassifier {

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

    public TestUnicastFileClassifier() {
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;
        FileClassifier classifier = null;

        // Prepare for discovery
        lookup = new LookupLocator("jini://www.all_about_files.com");

        // Discover a lookup service
        // This uses the synchronous unicast protocol
        registrar = lookup.getRegistrar();

        // Prepare a template for lookup search
        Class[] classes = new Class[] {FileClassifier.class};
        ServiceTemplate template = new ServiceTemplate(null, classes, null);

        // Lookup a service
```

```
        classifier = (FileClassifier) registrar.lookup(template);

        // Call the service
        MIMEType type;
        type = classifier.getMIMEType("file1.txt");
        System.out.println("Type is " + type.toString());
    }
} // TestUnicastFileClassifier
```

 © Jan Newmarch

Server Structure

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

The following code is simplified from the real case. It exports an implementation of a file classifier service, as a `FileClassifierImpl` object.

```
public class FileClassifierServer implements DiscoveryListener {

    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
        Thread.currentThread().sleep(Lease.FOREVER);
    }

    public FileClassifierServer() {
        LookupDiscovery discover = null;

        // Prepare for discovery - empty here

        // Discover a lookup service
        // This uses the asynchronous multicast protocol,
        // which calls back into the discovered() method
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);

        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
```

 © Jan Newmarch

```
        ServiceRegistrar registrar = evt.getRegistrars()[0];
        // At this point we have discovered a lookup service

        // Create information about a service
        ServiceItem item = new ServiceItem(null,
                                           new FileClassifierImpl(),
                                           null);

        // Export a service
        ServiceRegistration reg = registrar.register(item, Lease.FOREVER);

        // Renew leasing
        leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
    }
} // FileClassifierServer
```

DISCOVERING A LOOKUP SERVICE

 © Jan Newmarch

Lookup Service

- Before a client or service can do anything, they find a lookup service
- Unicast lookup can be made to known locations
- Multicast search can be made for unknown locations
- Sun supply a lookup service called `reggie`

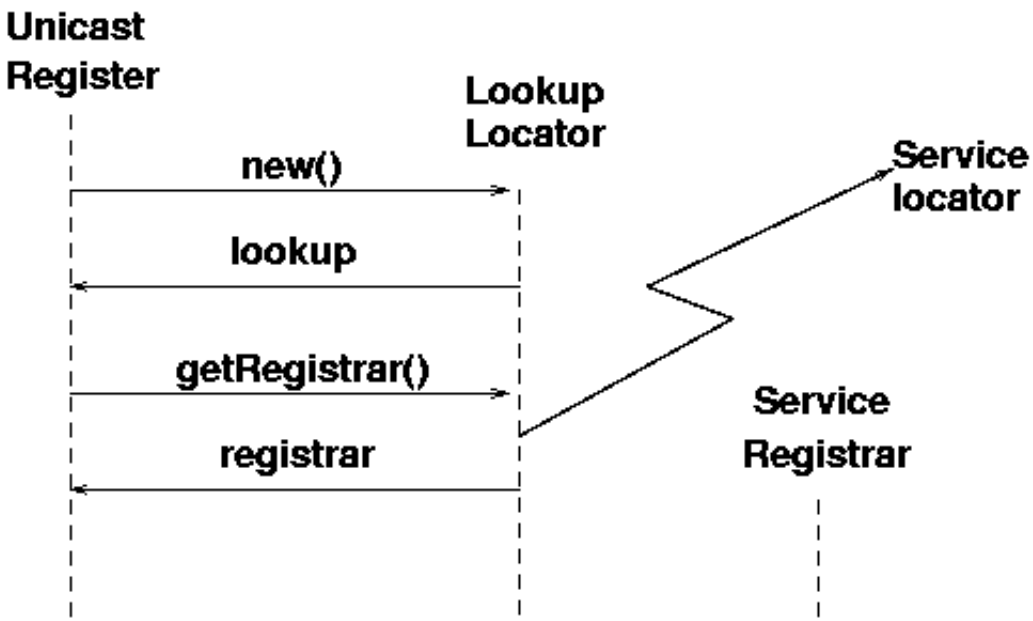
Unicast discovery

- The class `LookupLocator` handles unicast discovery

```
public class LookupLocator {
    public LookupLocator(String url);
    public LookupLocator(String host, int port);

    String getHost();
    int getPort();

    ServiceRegistrar getRegistrar() throws ...;
}
```

- `LookupDiscovery` specifies the url. Its `getRegistrar()` method returns a proxy for the lookup service

 © Jan Newmarch


Unicast
Register
Lookup
Locator
new()
lookup
getRegistrar()
registrar
Service
locator
Service
Registrar

Unicast discovery program

```
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;

public class UnicastRegister  {

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

    public UnicastRegister() {
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;

        try {
            lookup = new LookupLocator("jini://localhost");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }
```

 © Jan Newmarch

```
        try {
            registrar = lookup.getRegistrar();
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        }
        System.out.println("Registrar found");

        // the code takes separate routes from here for client or service
    }

} // UnicastRegister
```

Multicast discovery

- Broadcast discovery is done using the class `LookupDiscovery`

```
public class LookupDiscovery {
    public LookupDiscovery(String[] groups);

    addDiscoveryListener(LookupDiscoveryListener l);
}
```

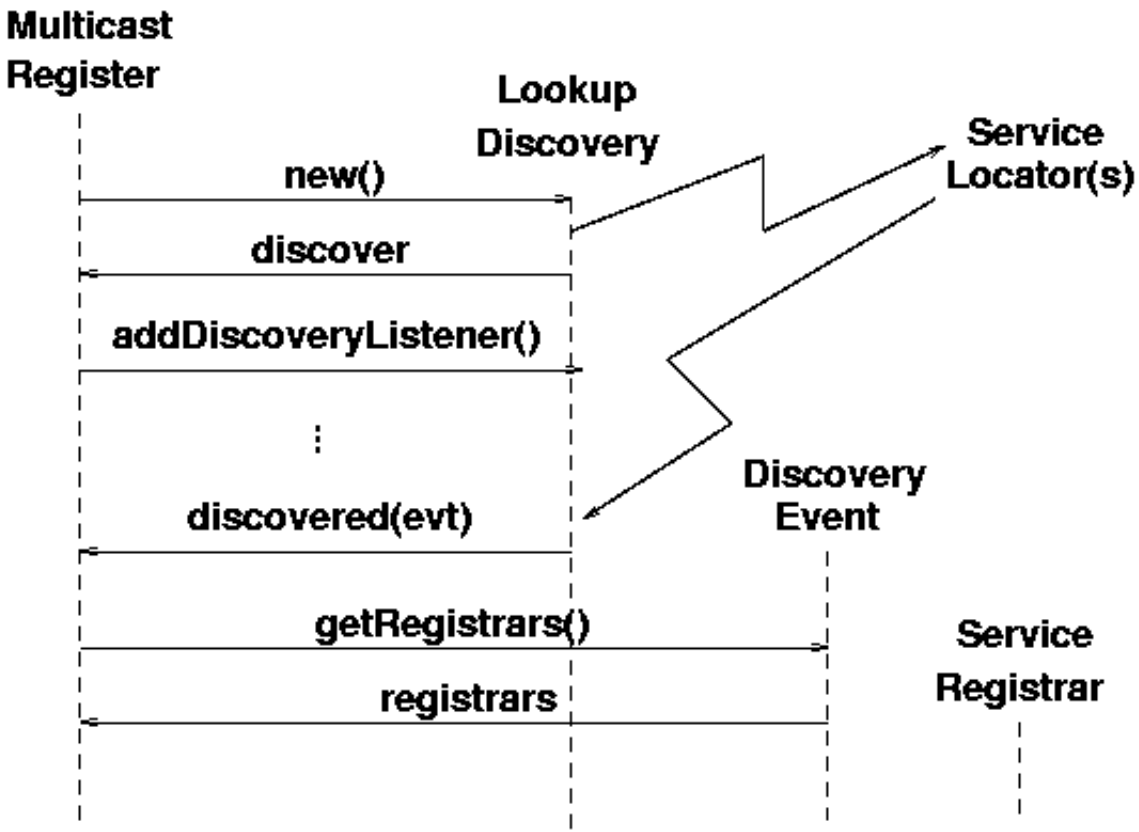
- Replies are handled asynchronously by a listener

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

 © Jan Newmarch

- The event contains registrar information

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



 © Jan Newmarch

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");
        }
    }
```

 © Jan Newmarch

```
    public void discarded(DiscoveryEvent evt) {

    }
} // MulticastRegister
```

ServiceRegistrar

- Both 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

 © Jan Newmarch

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`

 © Jan Newmarch

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

 © Jan Newmarch

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 contentTpe, 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

 © Jan Newmarch

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

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

Unicast client

Use this pattern when you do know where the lookup locator is:

```
package client;

import common.FileClassifier;
import common.MIMEType;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import java.rmi.RMISecurityManager;
import net.jini.core.lookup.ServiceTemplate;

public class TestUnicastFileClassifier {

    public static void main(String argv[]) {
        new TestUnicastFileClassifier();
    }
```

 © Jan Newmarch

```
public TestUnicastFileClassifier() {
    LookupLocator lookup = null;
    ServiceRegistrar registrar = null;
    FileClassifier classifier = null;

    try {
        lookup = new LookupLocator("jini://www.all_about_files.com");
    } catch(java.net.MalformedURLException e) {
        System.err.println("Lookup failed: " + e.toString());
        System.exit(1);
    }

    System.setSecurityManager(new RMISecurityManager());

    try {
        registrar = lookup.getRegistrar();
    } catch (java.io.IOException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    } catch (java.lang.ClassNotFoundException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
```

```
}

Class[] classes = new Class[] {FileClassifier.class};
ServiceTemplate template = new ServiceTemplate(null, classes, null);
try {
    classifier = (FileClassifier) registrar.lookup(template);
} catch(java.rmi.RemoteException e) {
    e.printStackTrace();
    System.exit(1);
}

if (classifier == null) {
    System.out.println("Classifier null");
    System.exit(2);
}
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());
```

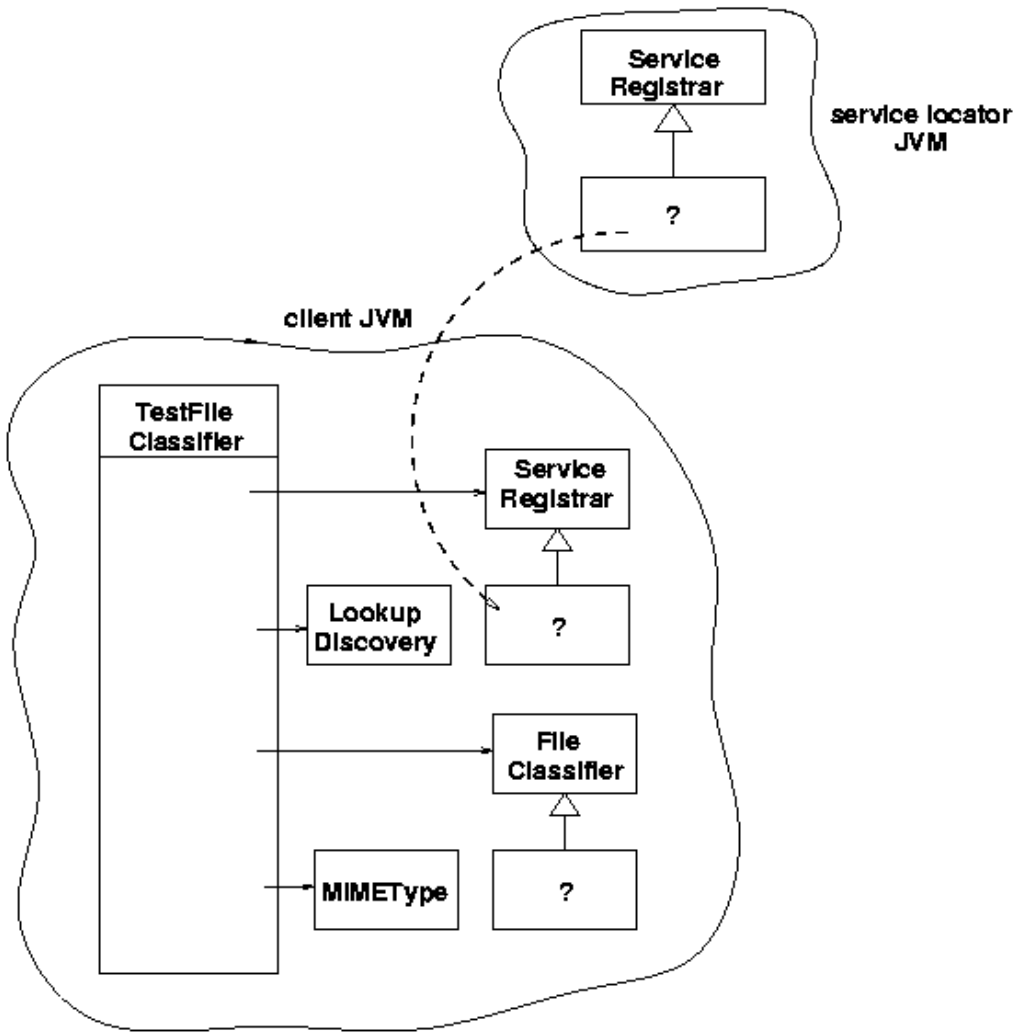
 © Jan Newmarch

```
        }
        // System.exit(0);
    }
} // TestUnicastFileClassifier
```

Client and service locator JVM's

The two JVM's look like

 © Jan Newmarch


Service Registrar
service locator JVM
?
client JVM
TestFile Classifier
Service Registrar
Lookup Discovery
?
File Classifier
MIMEType
?

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();
```

 © Jan Newmarch

```
        // 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;
```

 © Jan Newmarch

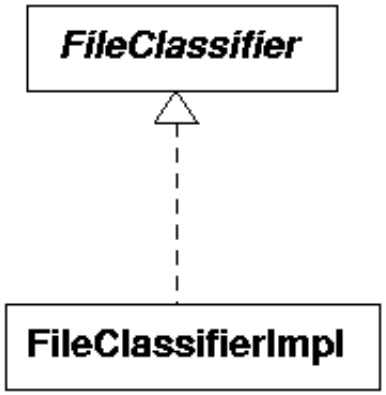
```
            }
            MIMEType type;
            try {
                type = classifier.getMIMEType("file1.txt");
                System.out.println("Type is " + type.toString());
            } catch(java.rmi.RemoteException e) {
                System.err.println(e.toString());
            }
            // System.exit(0);
        }
    }

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

PROXY IMPLEMENTATION CHOICES

 © Jan Newmarch

Proxy is the Service

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



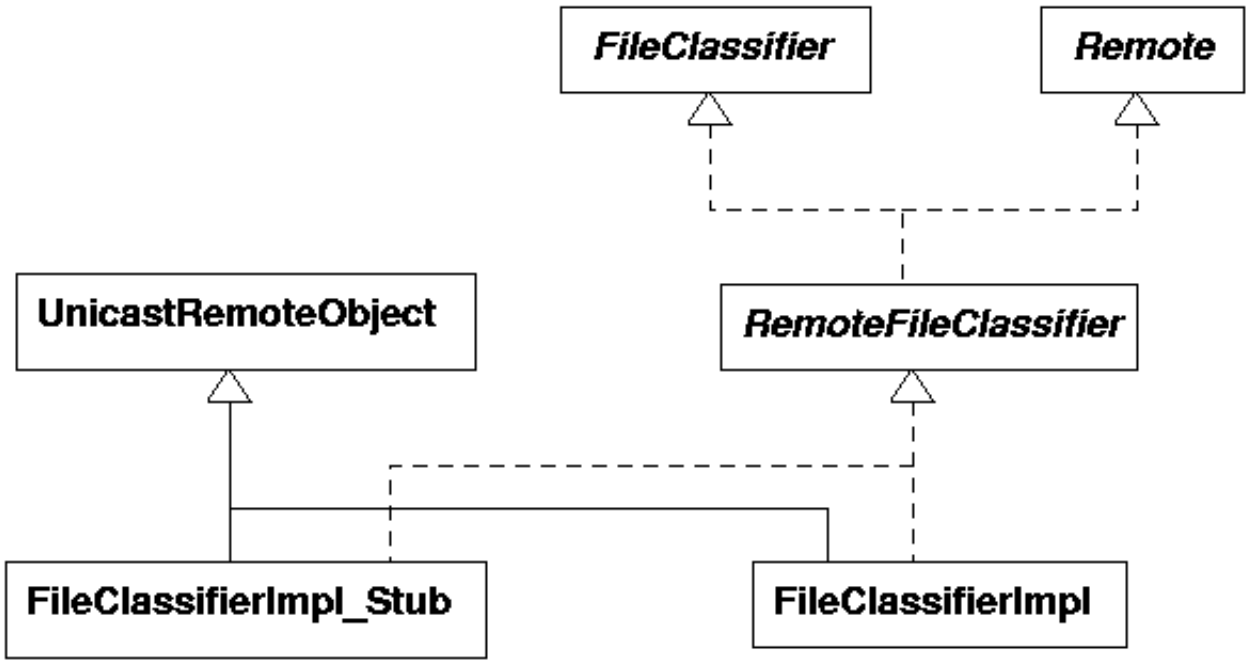
- The client uses `FileClassifier`
- The server creates a `FileClassifierImpl`
- The server exports the `FileClassifierImpl`
- This is a “fat” proxy model, with no processing being done on the server side
- This is suitable for software services that do not need to maintain state on

the server side

 © Jan Newmarch

RMI Proxies

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



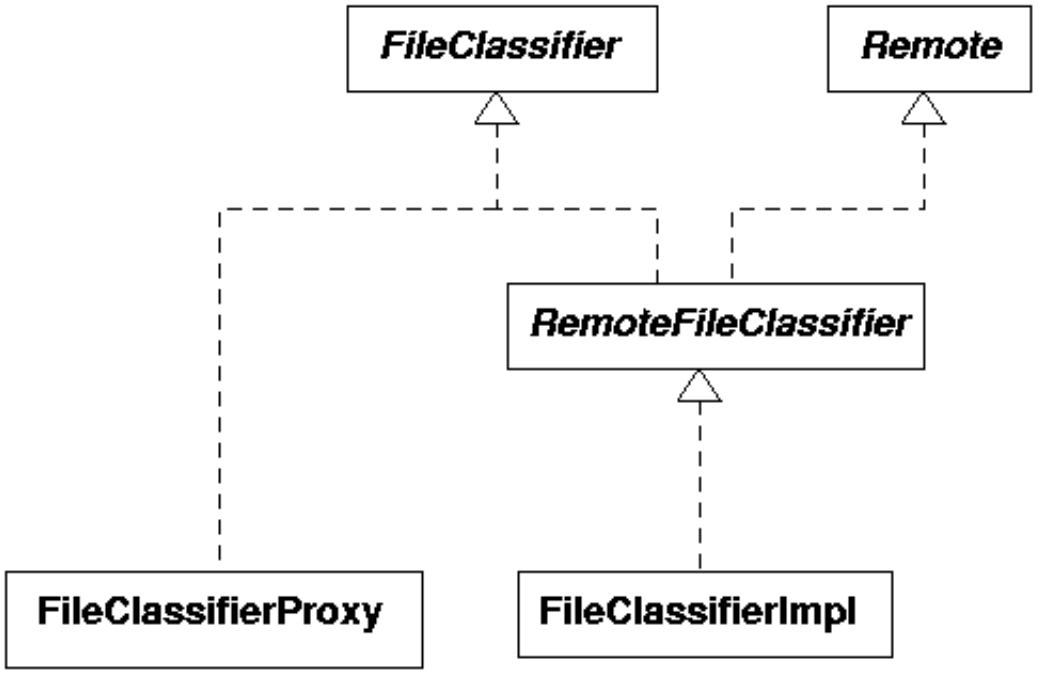
- The client uses `FileClassifier`
- The server creates a `FileClassifierImpl`

- 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

 © Jan Newmarch

Non-RMI Proxy

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



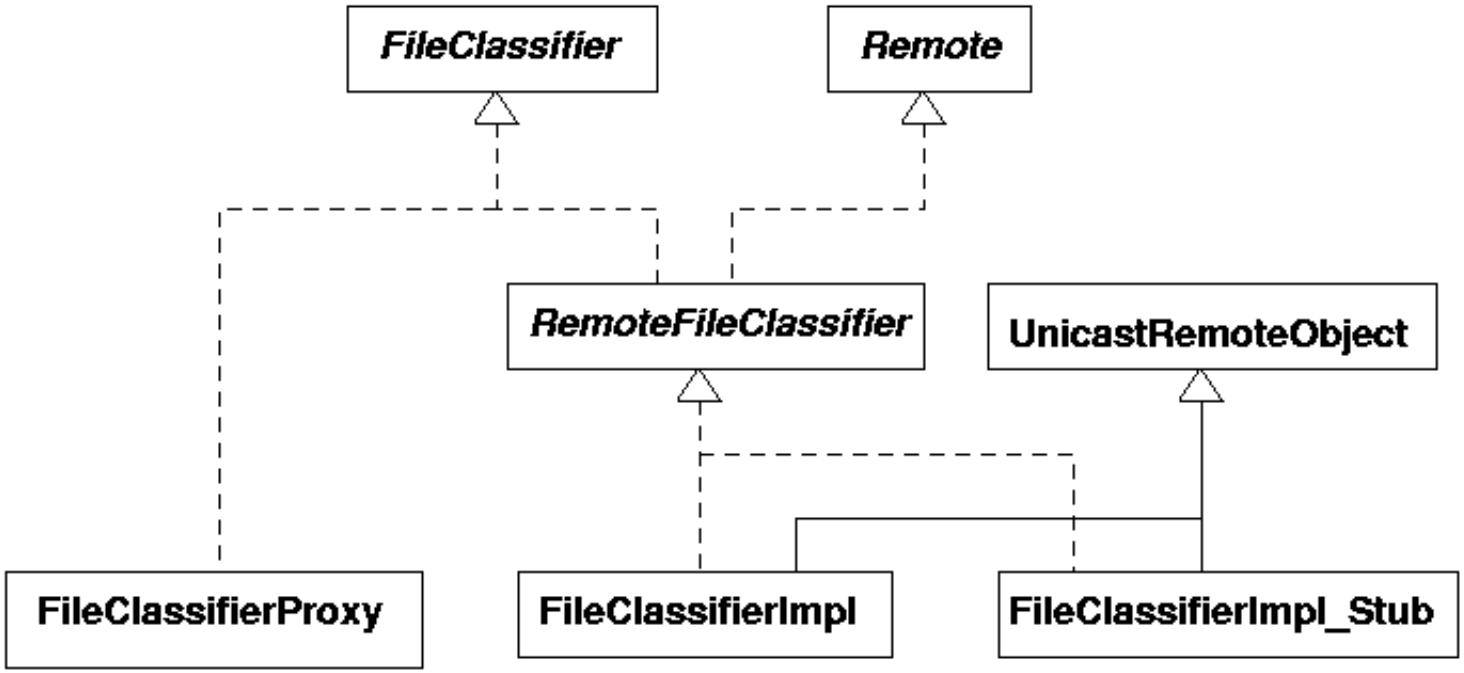
- The client uses `FileClassifier` (like before - it doesn't care what the proxy is)

- 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

 © Jan Newmarch

RMI and Non-RMI Proxies

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



- The client uses `FileClassifier`
- The server creates a `FileClassifierImpl` and a

`FileClassifierProxy`

- The server exports a `FileClassifierProxy` and a `FileClassifierImpl_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

 © Jan Newmarch

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

jan@ise.canberra.edu
Last modified: Sat Sep 4 00:14:09 EST 1999
Copyright ©Jan Newmarch