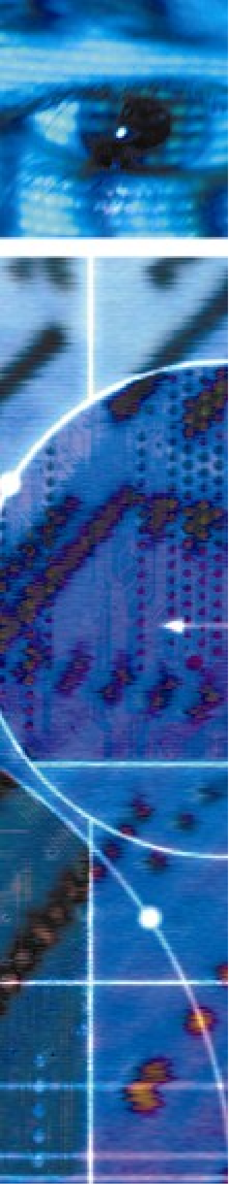
RMI (Remote Method Invocation)

Sang Shin
www.JPassion.com

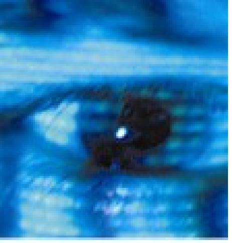
"Learn with JPassion!"

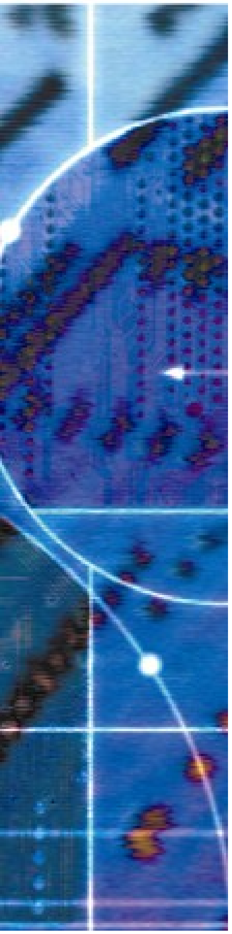


Topics

- What is RMI? Why RMI?
- Architectural components
- Serialization
- Writing RMI Server and Client
- Dynamic class loading
- Code movement
- Codebase
- ClassLoader delegation
- Activation
- RMI Security
- HTTP Tunneling

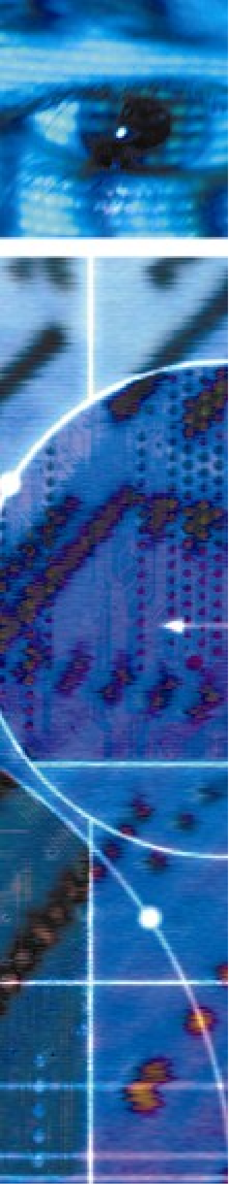
What is RMI?





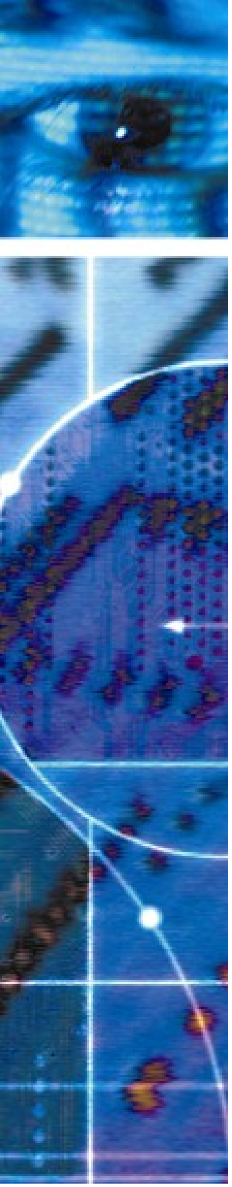
What is RMI?

- RPC (Remote Procedure Call) between Java Objects
- General RPC behavior
  - Invoke remote methods
  - Pass arguments into methods
  - Receive results from methods
- RPC Evolution
  - Non-object-oriented RPC
  - CORBA (Object-oriented)
  - RMI (Object-based – Java only)



What is RMI?

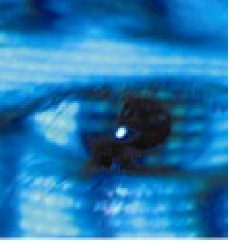
- Differences from other RPC’s
  - RMI is Java-based
  - RMI supports code movement
  - RMI has built-in security mechanism
  - RMI exposure of network failures to application programmers through *RemoteException*

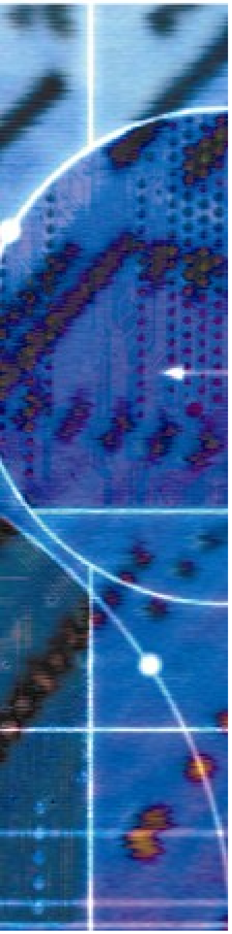


Why RMI?

- Capitalizes on the Java object model
- Minimizes complexity of distributed programming
- Uses pure Java interfaces
  - no new interface definition language (IDL)
- Preserves safety of Java runtime
- Recognizes differences of remote call from local call
  - partial failure
  - latency
  - no global knowledge on system state

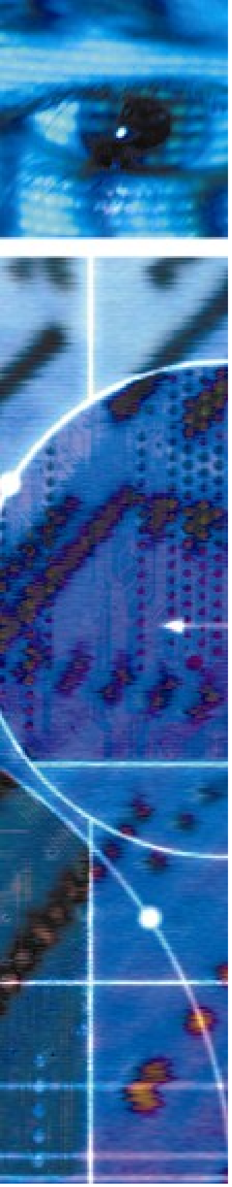
RMI Architectural Components





RMI Architectural Components

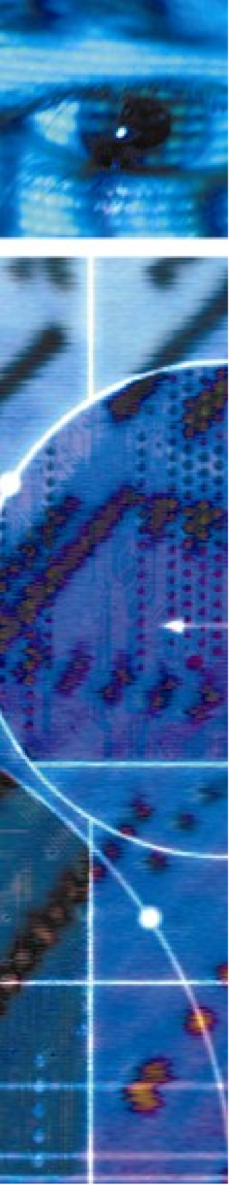
- Remote interface
- Stub and Skeleton (generated through “rmic”
- Remote object



Remote Interface

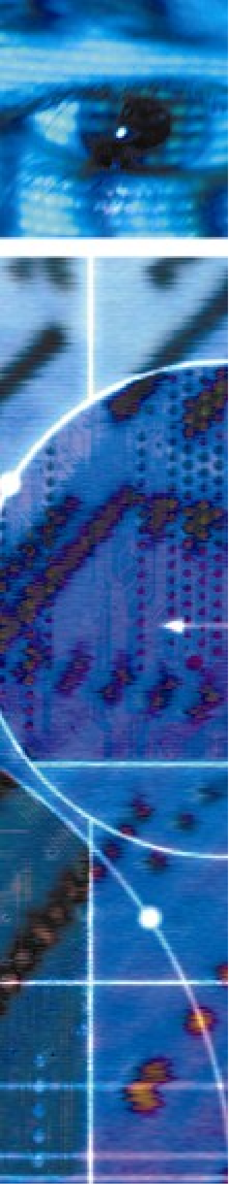
- Java interface
  - Specify remotely accessible methods
- Implemented by a class, an instance of which becomes a remote object
- Contract between caller of the remote method (RMI client) and remote object (RMI server)
- Extends *java.rmi.Remote* interface
  - Markup interface

Stub & Skeleton



Stub and Skeleton

- A tool (rmic) creates
  - RMI stub
  - (Optionally) RMI skeleton
- Gets created from RMI server implementation (not from RMI interface)

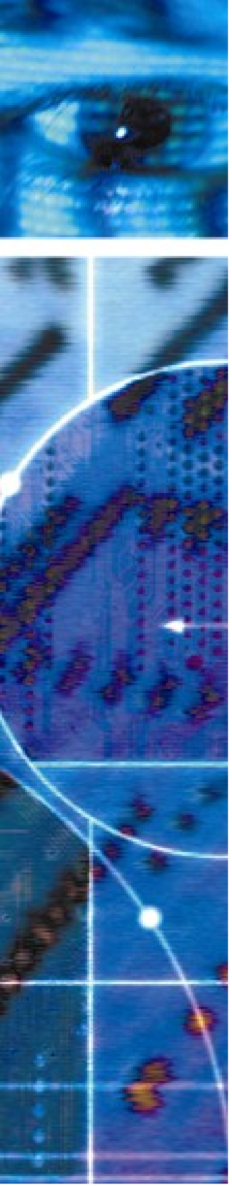


Stub and Skeleton

- RMI Stub
  - Resides in client’s local address space
  - Represents remote object to client
    - Plays the role of proxy of remote object
    - Implementation of Remote interface
    - Caller invokes methods of RMI Stub locally
  - Connects to the remote object
  - Sends arguments to and receive results from remote object
    - Performs marshaling and unmarshaling

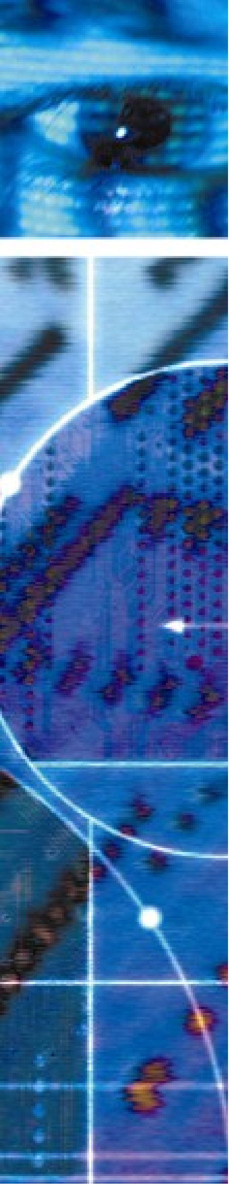
Stub and Skeleton

- RMI Skeleton
  - ◆ Resides in server’s address space
  - ◆ Receives arguments from caller (RMI Client's Stub) and send results back to caller
    - ■ Performs marshaling and unmarshaling
  - ◆ Figures out which method of remote object to be called
  - ◆ From JDK 1.3, RMI Skeleton gets created automatically via reflection



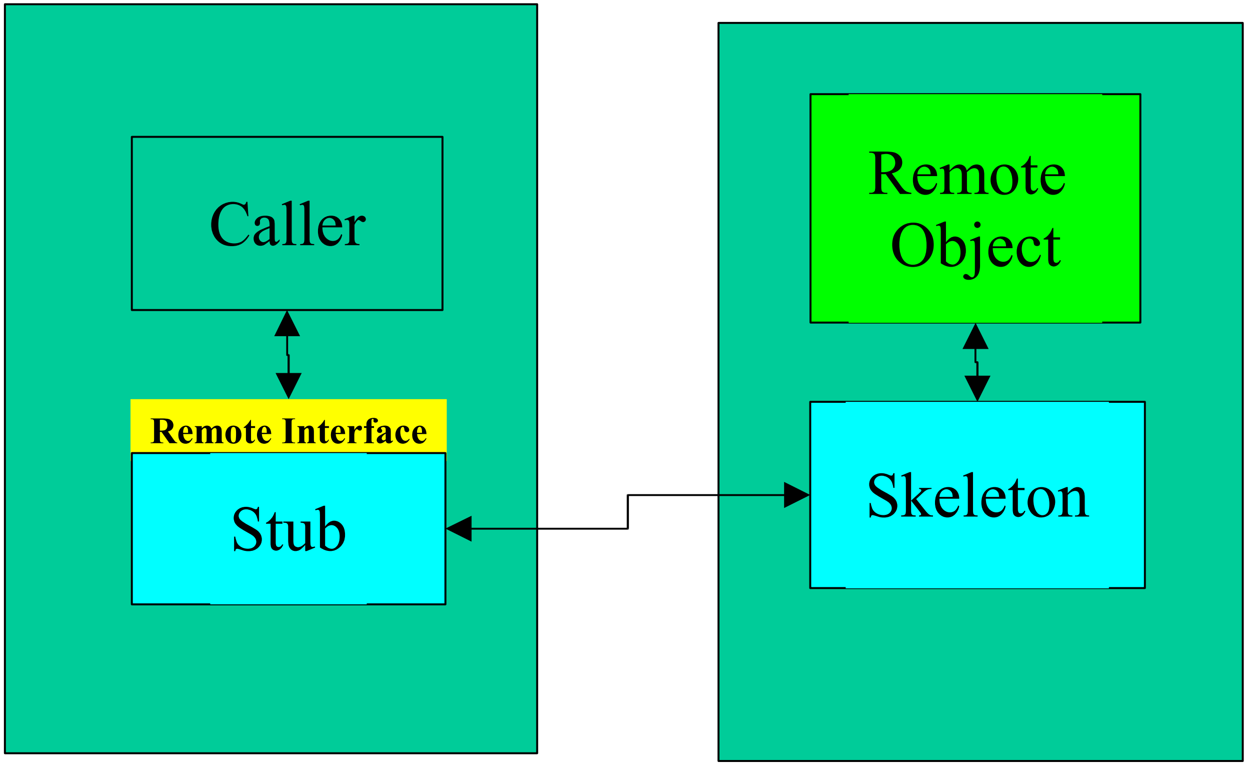
Remote Object

- Implementation of remote interface
- Needs to be exported
  - ◆ In order to be ready to receive calls from caller
- Can be exported in two types
  - ◆ Non-activatable (extends *java.rmi.server.UnicastRemoteObject*)
  - ◆ Activatable (extends *java.rmi.server.Activatable*)

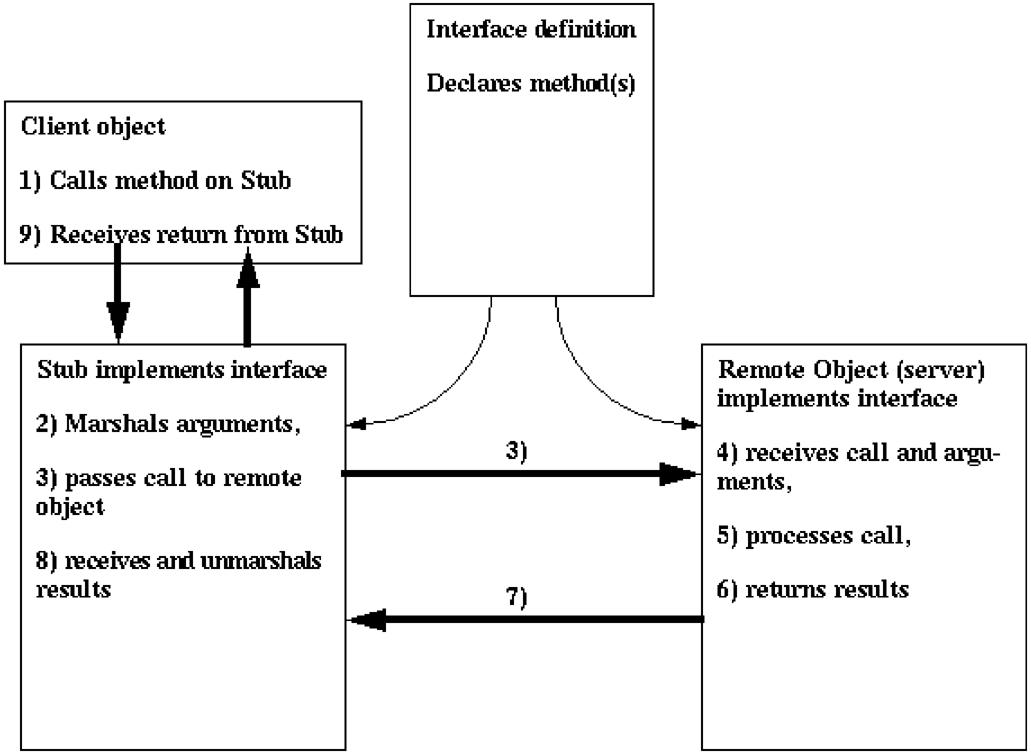


RMI Communication Model

RMI Communication Model



RMI Control Flow



Interface definition

Declares method(s)

Client object

1) Calls method on Stub

9) Receives return from Stub

Stub implements interface

2) Marshals arguments,

3) passes call to remote object

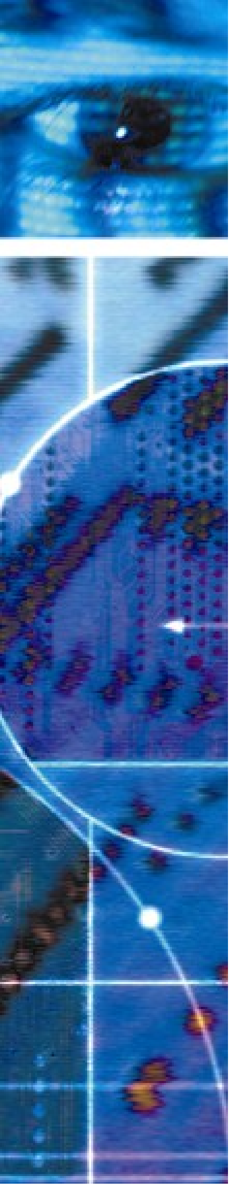
8) receives and unmarshals results

Remote Object (server) implements interface

4) receives call and arguments,

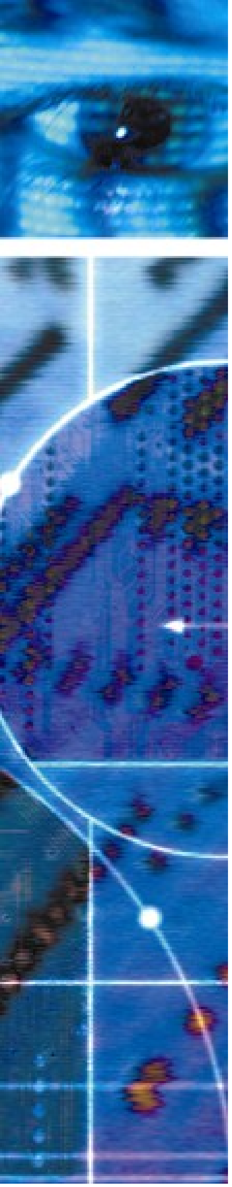
5) processes call,

6) returns results



RMI Control Flow

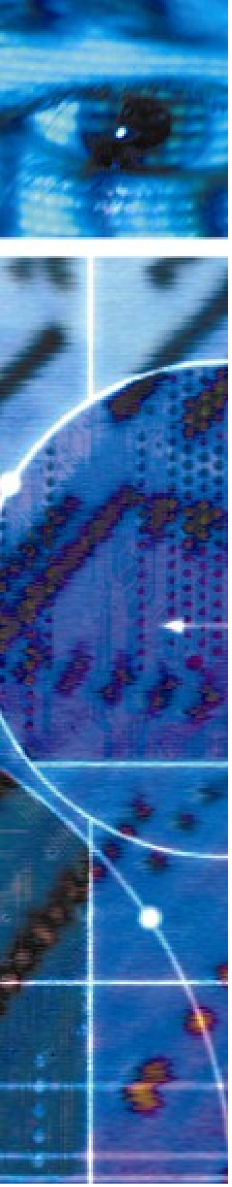
- Caller (Client)
    1. invokes a method of a remote object
- Stub of the remote object
    1. intercepts the method call
    2. marshals the arguments
    3. makes calls to remote object



RMI Control Flow

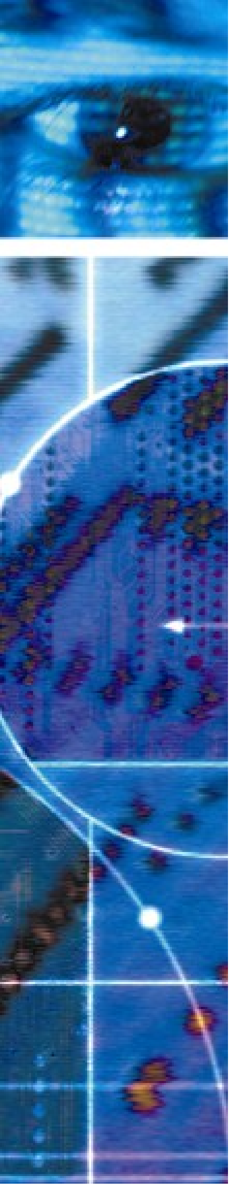
- Remote object
  1. Receives the calls via Skeleton
  2. Unmarshals the arguments
  3. Performs the call locally
  4. Marshals the result
  5. Send the result to client
- Stub
  1. Receives result
  2. Unmarshal result
  3. Return result to client

Serialization in RMI



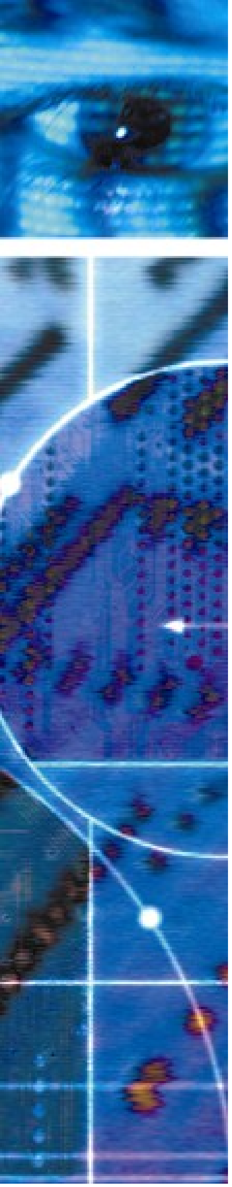
Marshaling and Unmarshaling

- Marshaling is a process of encoding objects to put them on the wire
- Unmarshaling is the process of decoding from the wire and placing object in the address space
- RMI uses Java programming lanaguage's serialization and deserialization to perform marshaling and unmarshaling
  - These terms are used interchangeably



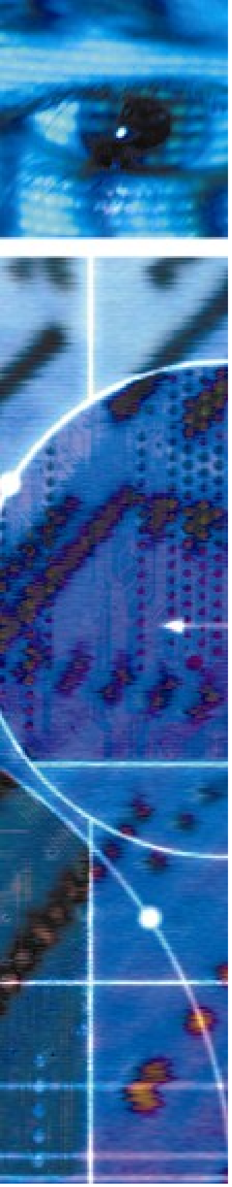
Serialization in RMI

- Arguments/Results get serialized before being transported by sender
- Arguments/Results get deserialized after being transported by receiver
- Arguments/Results in RMI can be one of the following two
    - Remote object
    - Non-remote object



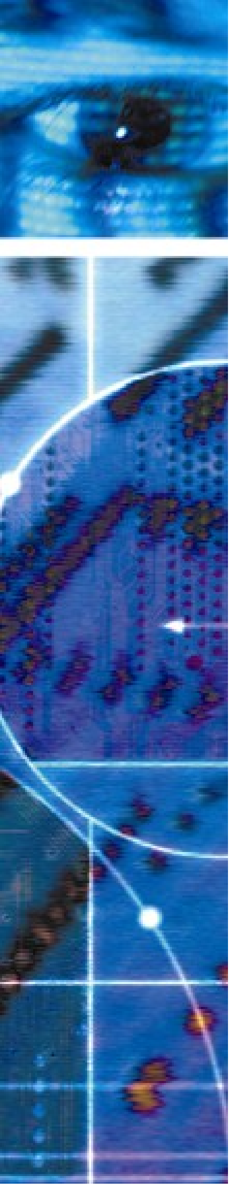
Serialization in RMI

- For remote object
  - Object which is *Remote* interface type
  - Stub gets serialized (instead of remote object itself)
  - “Pass by reference” semantics
    - Stub is kind of a reference to remote object
- For non-remote object
  - Object which is not *Remote* interface type
  - Normal serialized copy of the object
  - Should be type of *java.io.Serializable*
  - “Pass by Value” semantics



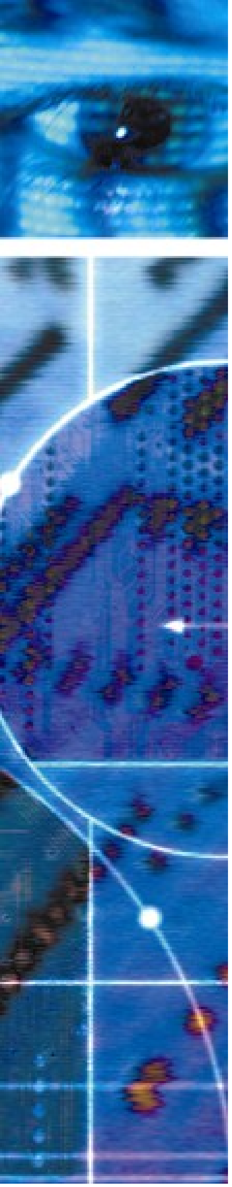
Example

```
// Arguments and Returns are non-remote objects
public interface SayHelloStringRemote extends Remote {
    public String SayHelloString (String message)
                                          throws  RemoteException;
}

// Arguments has both non-remote and remote objects
public interface SayHelloObjectRemote extends Remote {
    public String SayHelloObject (String messsage,
                                  SayHelloStringRemote  someName)
                                          throws  RemoteException;
}
```



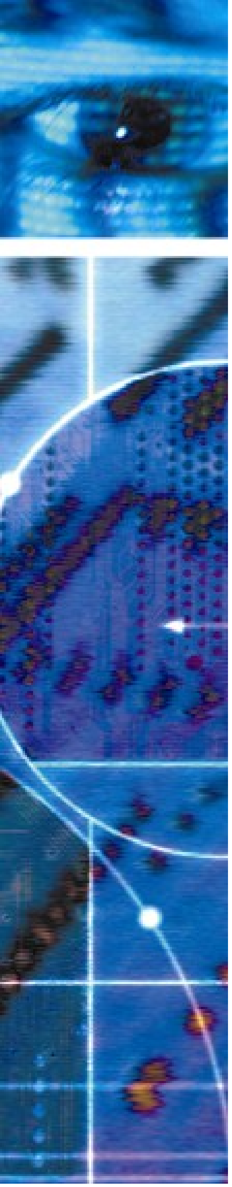
Serialization

- Serialized copy of an object
  - Stream of bytes
  - Persistently maintains state of an object
    - State of non-static and non-transient variables of the object
  - Does NOT contain class bytecodes (*.class files)
    - Instead maintains information on “where to get the class bytecodes”
      - codebase annotation
      - Who performs the codebase annotation?
    - If the class is unknown to the recipient, it will be downloaded automatically



Serialization

- Serialized copy defines state
- Class files define behavior
- Both can be moved around over the network
  - Collectively this is called "Code movement"



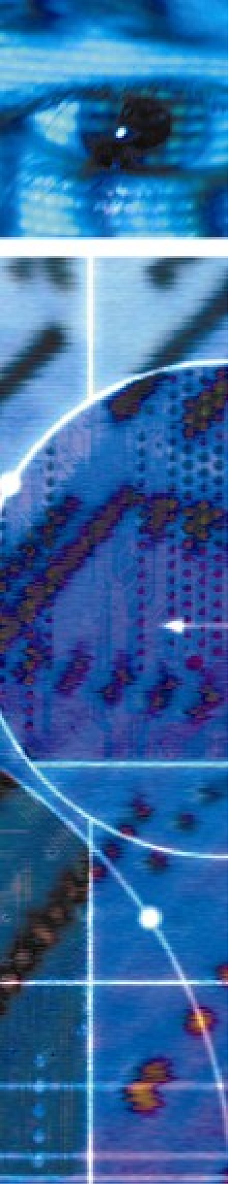
Writing RMI Server

Steps of Writing RMI Server

- #1: Define remote interface
- #2: Write and compile server implementation
- #3: Generate stub class from server implementation class
- #4: Write startup class

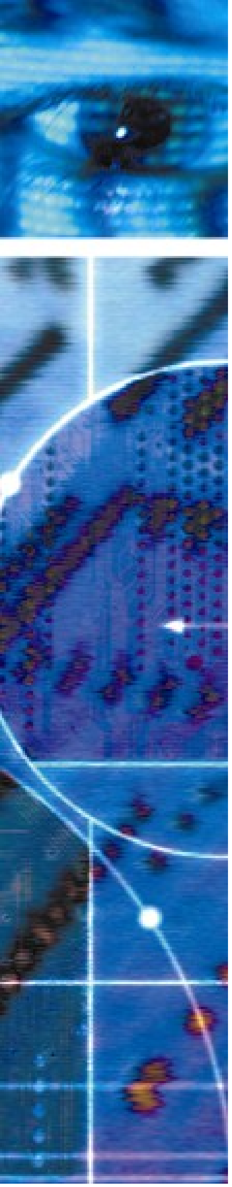
#1: Define Remote Interface

- Defines methods that are called remotely
- Must be declared as *public*
- Extends the *java.rmi.Remote* interface
- Each method must declare *java.rmi.RemoteException*
- The data type of any remote object that is passed as an argument or return value (either directly or embedded within a local object) must be declared as the Remote interface type (for example, *Hello*) not the implementation class (*HelloImpl*).



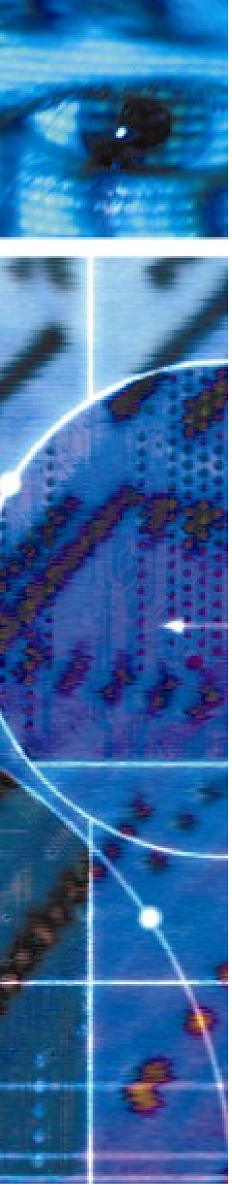
#1: Remote Interface Example

```
import java.rmi.*;

/**
 * Remote Interface
 */
public interface HelloInterface extends Remote {

    public String sayHello(String name)
                          throws RemoteException;
}
```



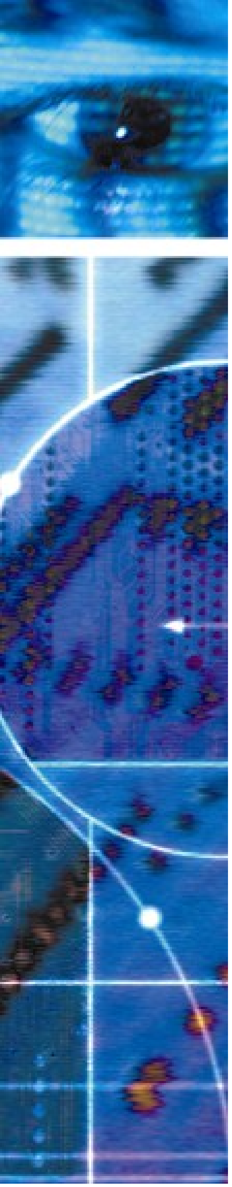
#2: Write Server implementation

- Implement the remote interface
- Extend one of the two remote classes
  - *java.rmi.server.UnicastRemoteObject*
  - *java.rmi.activation.Activatable*
- Write constructor for the remote object
  - By extending one of the two remote classes above, they are automatically exported
    - You can manually export it as well
  - Throw RemoteException
  - Register remote objects with RMI registry



#2: Server Implementation Example

```
import java.rmi.*;
import java.rmi.server.*;

/**
 * Remote implementation class.  Because it extends the
 * UnicastRemoteObject, it is automatically exported.
 */
public class HelloImpl extends UnicastRemoteObject
        implements HelloInterface {

    public HelloImpl() throws RemoteException {
    }

    public String sayHello(String name) throws RemoteException {
        return "Hello " + name + "!";
    }
}
```



#3: Generate Stub class

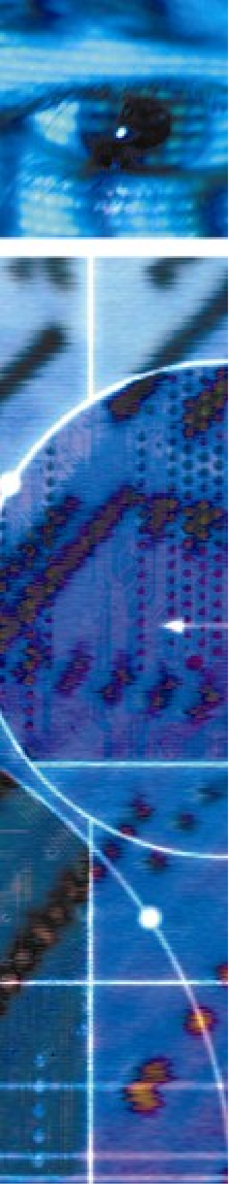
```
C:\myprojects\RMI_app>rmic HelloImpl
C:\myprojects\RMI_app>dir
 Volume in drive C has no label.
 Volume Serial Number is F090-5679

 Directory of C:\myprojects\RMI_app

10/16/2010  07:39 AM    <DIR>          .
10/16/2010  07:39 AM    <DIR>          ..
10/16/2010  07:37 AM               454 HelloImpl.class
10/16/2010  07:37 AM               757 HelloImpl.java
10/16/2010  07:39 AM             1,639 HelloImpl_Stub.class
10/16/2010  07:37 AM               222 HelloInterface.class
10/16/2010  07:35 AM               357 HelloInterface.java
               5 File(s)          3,429 bytes
               2 Dir(s)  22,326,777,856 bytes free
```

#4: Write Startup code

- Contains main() method
- Create and export remote object
- Register remote object with RMI registry

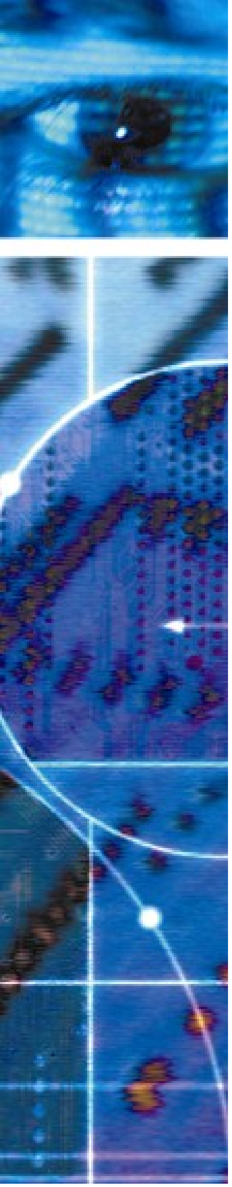
Startup code example

```
import java.rmi.*;

public class HelloServer {

    public static void main(String[] argv) {
        try {
            // Create remote object and register with rmiregistry
            Naming.rebind("Hello", new HelloImpl());
            System.out.println("Hello Server is ready.");
        } catch (Exception e) {
            System.out.println("Hello Server failed: " + e);
        }
    }
}
```

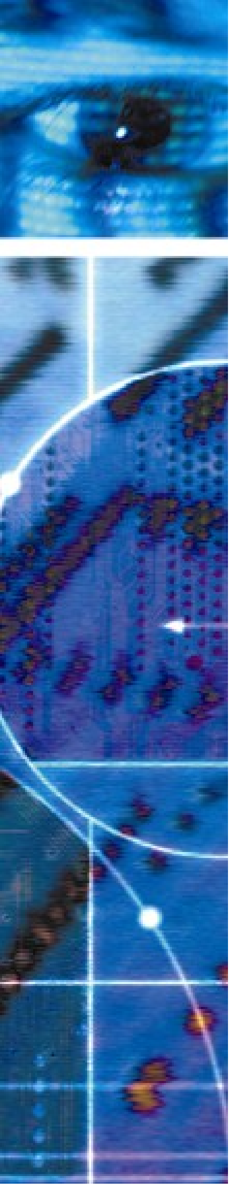


RMI Registry

- RMI Registry is a simple naming service
  - Bootstrap mechanism
  - Typically is used by caller to get the remote reference of the first remote object
- Client gets reference to remote object - actually reference to stub object of the remote object

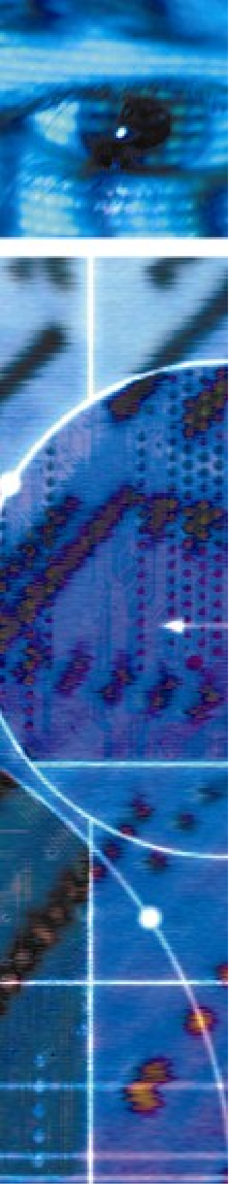


Writing RMI Client



Steps of Writing RMI Client

- Get a reference to the remote object implementation
    - ◆ The registry returns the Stub instance of the remote object bound to that name
- Invoke remote methods

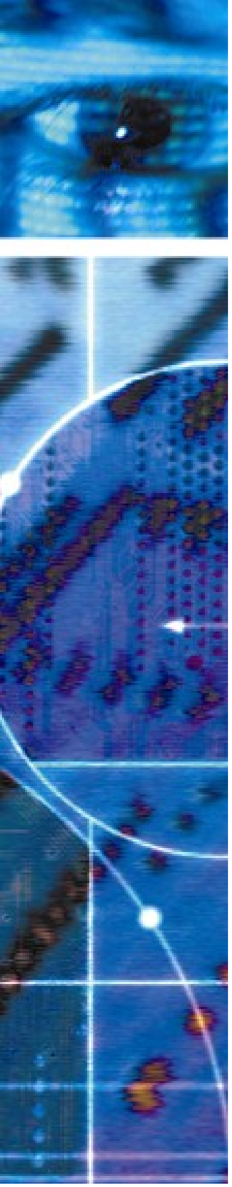


Client Example

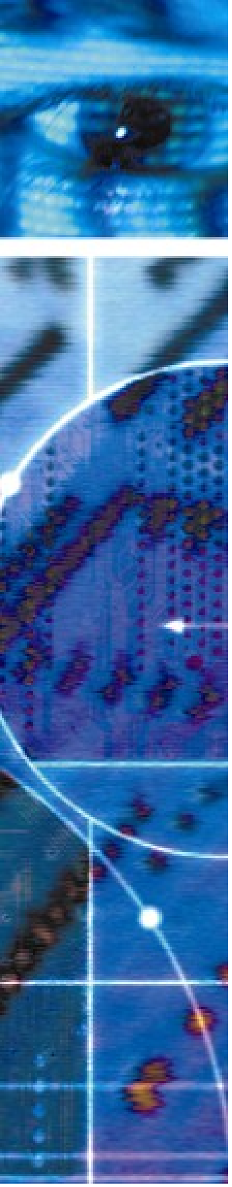
```
import java.rmi.*;

public class HelloClient {

    /**
     * Client program for the "Hello, world!" example.
     * @param argv The command line arguments which are ignored.
     */
    public static void main(String[] argv) {
        try {
            HelloInterface hello =
                    (HelloInterface) Naming.lookup("Hello");
            String result = hello.sayHello("Sang Shin");
            System.out.println(result);
        } catch (Exception e) {
            System.out.println("HelloClient exception: " + e);
        }
    }
}
```

Demo:

Exercise 1: “Hello World” RMI Server and Client 1602_javase_rmi.zip

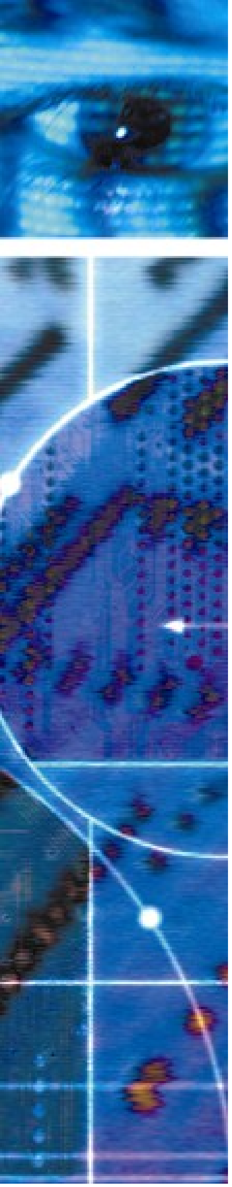


Dynamic Class Loading



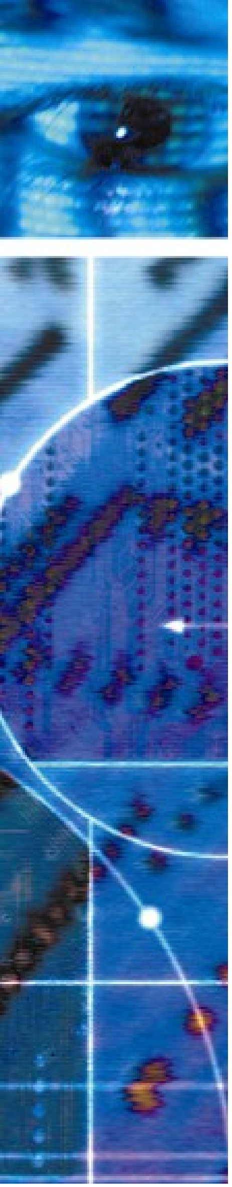
Dynamic Class Loading

- Class bytecodes (Class file) get downloaded during runtime
  - When caller does not have the class bytecodes in local classpath
    - RMI Stub needs to be downloaded to RMI Caller’s address space from somewhere
  - Serialized copy of an object contains “where to get class bytecodes” information
    - Codebase annotation



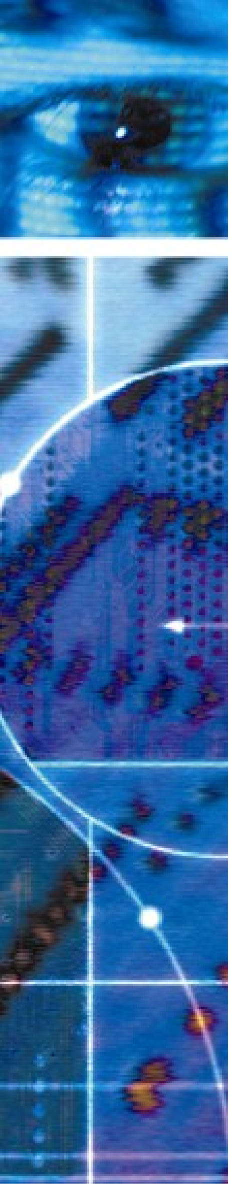
Who Does Provide Codebase Annotation Information?

- By the exporter of the class
- Via Export codebase (RMI codebase) property
  - *java.rmi.server.codebase*
  - Typically via HTTP URL



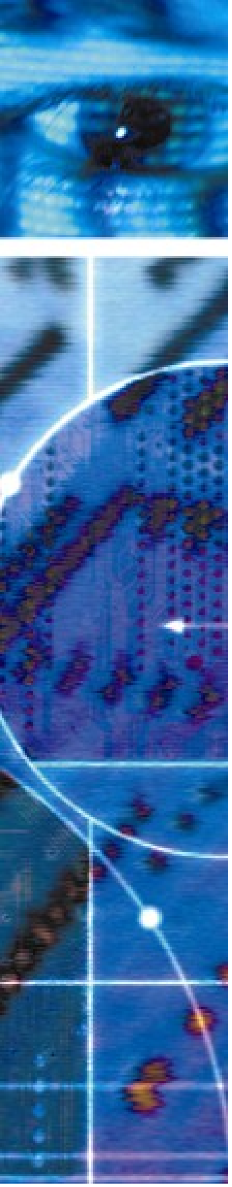
When Does the Codebase Annotation occurs?

- Whenever an object gets serialized
- For remote object
  - ◆ Codebase information of Stub class
- For non-remote object
  - ◆ Codebase information of normal class



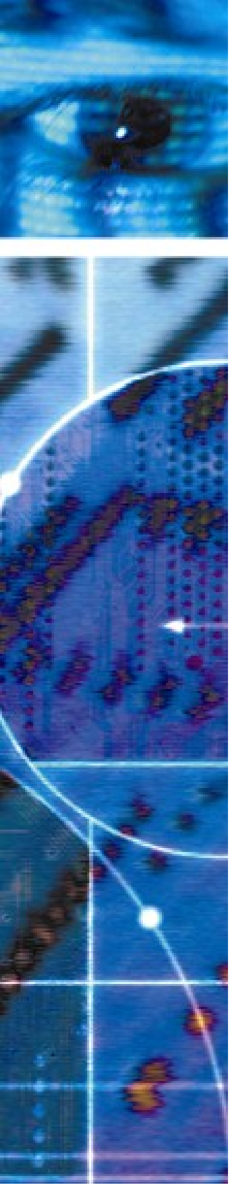
RMI Server and Client Deployment Scenario

- Both client and server have RMI Remote interface class in their local classpaths
- Server has *HelloWorld_Stub* class in its local classpath
- Client does not have *HelloWorld_Stub* class in its localpath
  - He could, but is diminishes the whole purpose of class downloading
- Server exports *HelloWorld_Stub* class via HTTP server

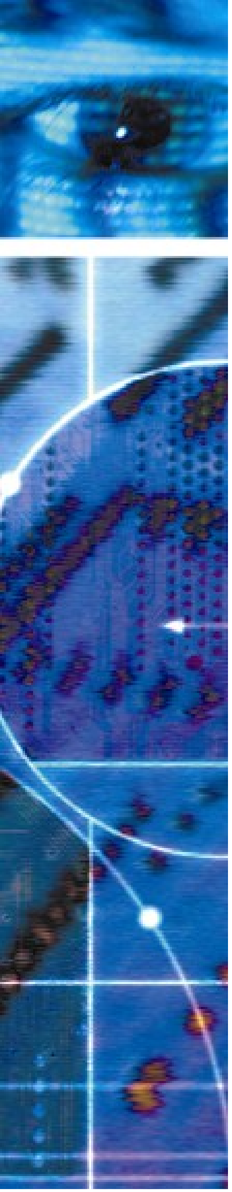


RMI Server and Client Deployment Scenario (Continued)

- Client gets *HelloWorld_Stub* serialized object from Registry
  - ◆ Client typically does not have *HelloWorld_Stub* class in its local classpath
  - ◆ So it will read the RMI codebase annotation (from the serialized stub object) and will try to download the *HelloWorld_Stub* class from the location specified in codebase annotation

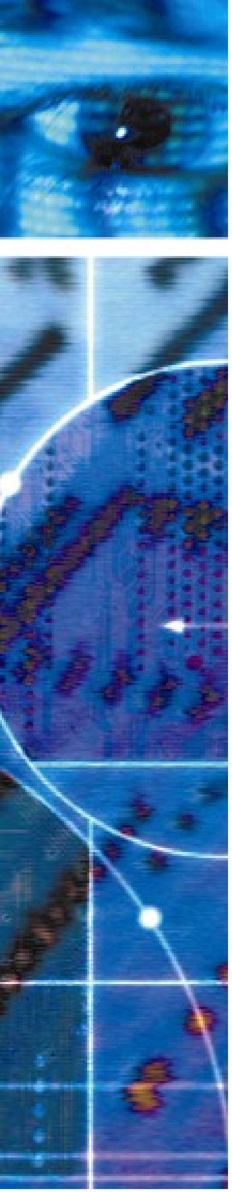


Code (and Data) Movement



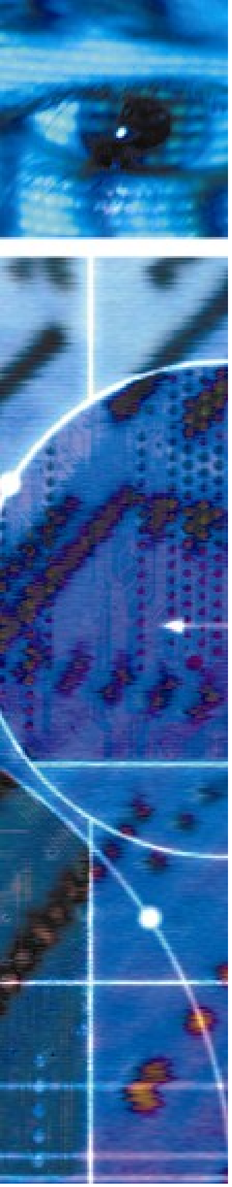
Code (and Data) Movement

- Performed in two phases
  1. Serialized object (Marshalled Object) gets moved
  2. Class files get downloaded
- Code
  - Represented by class files
- Data
  - Represented by state captured in serialized object



Serialized Object

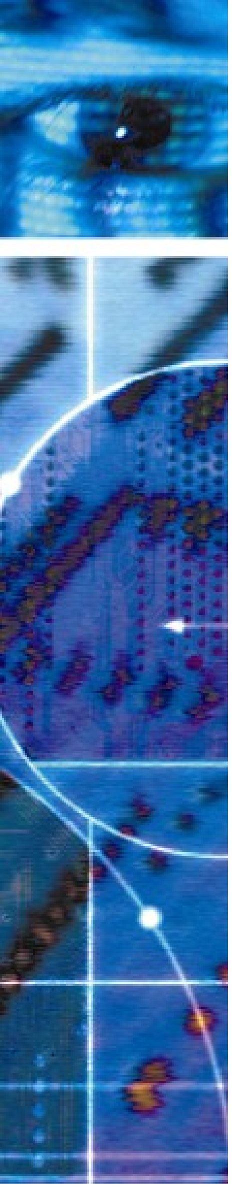
- Contains
  - Values of the fields of the object
  - Name of the class
  - Location of the class
    - Via codebase annotation performed by the exporter of the class
    - RMI codebase property



Codebase

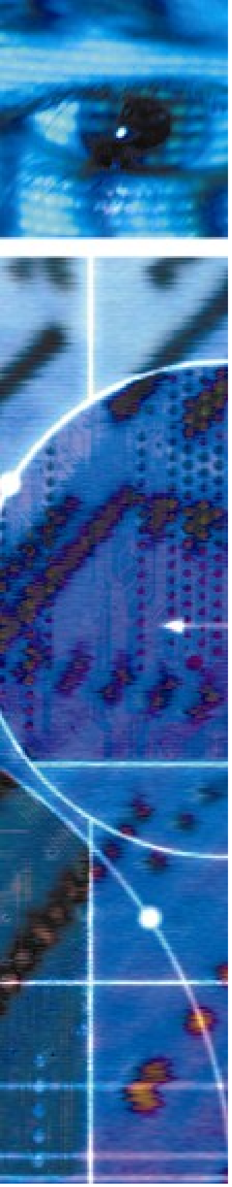
What is Codebase?

- Location where class bytecodes (Class files) reside



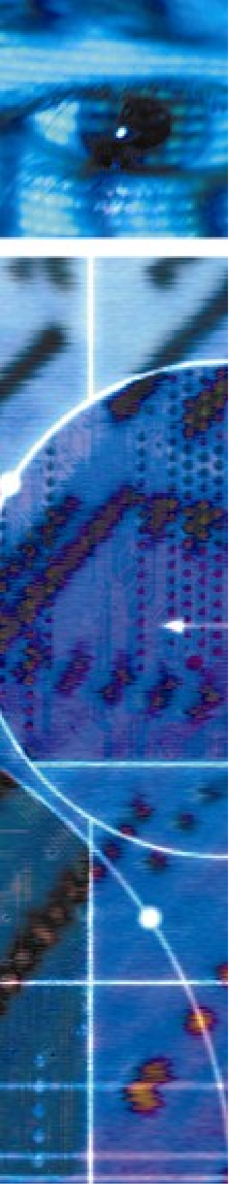
Two types of Codebase

- Import codebase
  - codebase your local VM uses to load classes it needs
  - specified via *CLASSPATH* or *-cp* option
- Export codebase (RMI codebase)
  - codebase remote VMs use to obtain the class files "exported" from your local VM
  - specified via *java.rmi.server.codebase* property
    - Codebase annotation



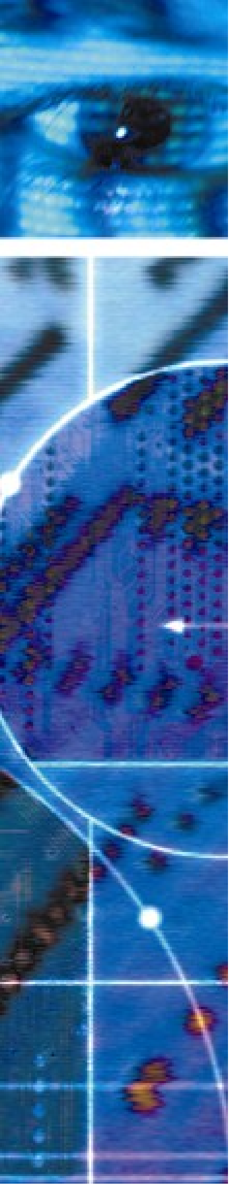
Behind the Scene Activities

- Any objects marshaled by a server will be annotated with RMI codebase
  - For remote object, the stub object gets marshaled and annotated
- When a client instantiates the object, the bytecodes of the class will be downloaded by RMIClassloader from the location specified as RMI codebase



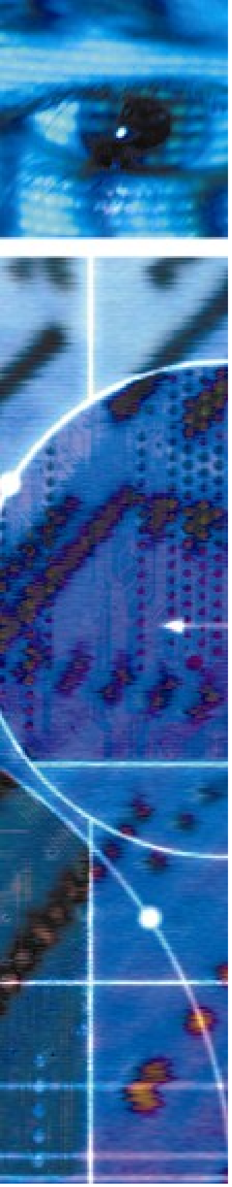
RMI codebase forms

- Could be in any URI form
  - HTTP (Recommended)
  - FTP
  - FILE (Not recommended)
- Classes can be accessible via
  - JAR
  - Directory path
    - Trailing slash required



RMI codebase

- RMI server
  - Export classes that are needed by its client
    - Stub classes for remote objects
    - Interface classes of remote objects
      - If client has the classes in its local classpath, no downloading occurs
    - Any classes that are needed by the interface and stub classes
- RMI client
  - Export classes that are needed by the server
    - Same as above

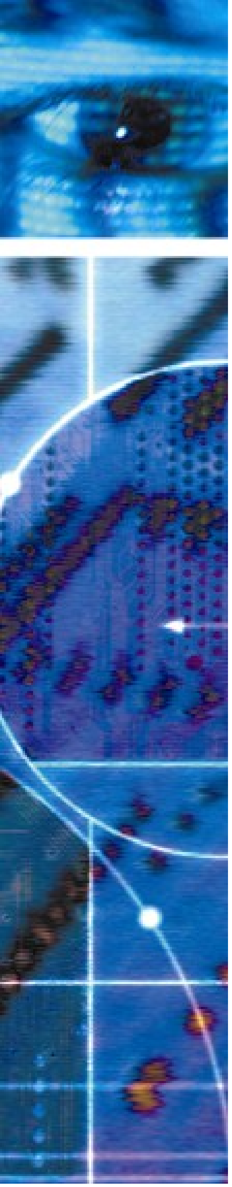


RMI codebase examples

- Directories need a trailing slash
  - -Djava.rmi.server.codebase="file:/export/home/btm/classes/”
  - -Djava.rmi.server.codebase= "http://daydreamer:8080/export/home/btm/root/dir/”
- Jar files do not need a trailing slash
  - -Djava.rmi.server.codebase= "file:/export/home/btm/jars/examples-dl.jar”
  - -Djava.rmi.server.codebase= "http://daydreamer:8080/export/home/btm/jars/examples-dl.jar”
- You can specify multiple locations
  - -Djava.rmi.server.codebase= "http://daydreamer:8080/export/home/btm/jars/examples-dl.jar http://daydreamer:8080/export/home/btm/root/dir/"

Demo:

Exercise 2: “Hello World” RMI Server and Client Using Export Codebase 1602_javase_rmi.zip



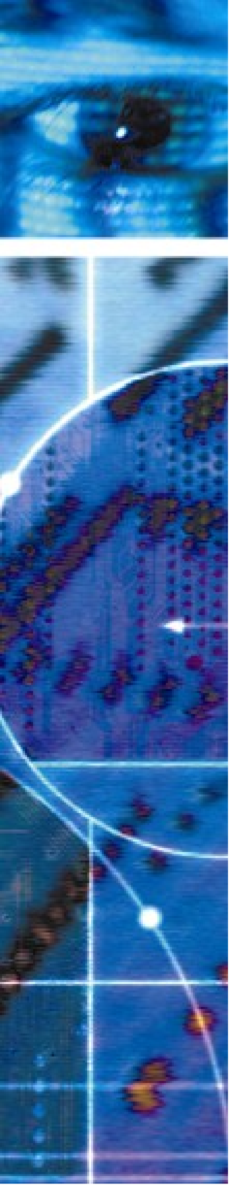
Typical Causes of Problems

- The *java.rmi.server.codebase* (RMI codebase) property was not set at all
  - Do not use “localhost”
- RMI codebase was set, but HTTP server is not running
- RMI codebase was set, HTTP server is running, but the class is not present under the proper path in HTTP server
- The port number on which HTTP server is listening is not the same as the port number in the RMI codebase
- The name of the host on which HTTP server is running is not the same as the hostname in the RMI codebase
- If a non-jar URL is being used in the RMI codebase, there is no trailing slash (if class file location is in a jar file, no trailing slash is required)

Typical RMI codebase Symptom

java.rmi.UnmarshalException: error unmarshalling return; nested exception is: java.lang.ClassNotFoundException: example.testService_Stub

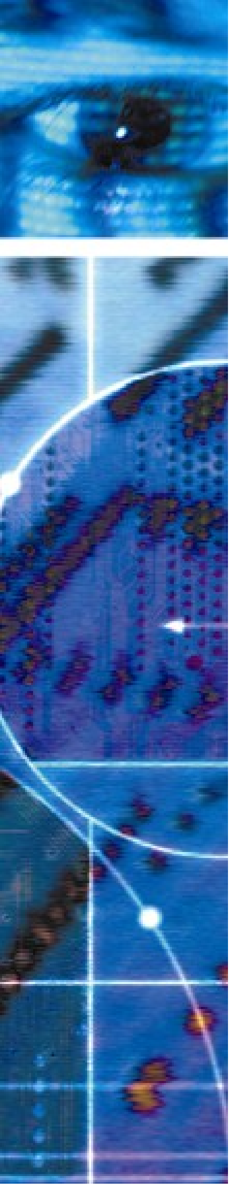
- Client could not download the stub class from the server



Typical RMI codebase Symptom

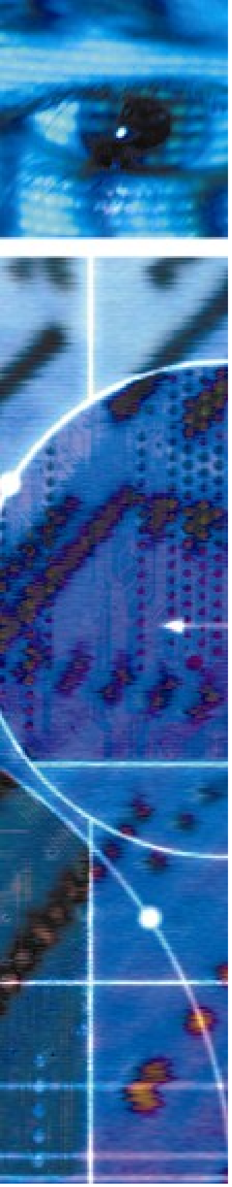
```
RemoteException occurred in server thread;  nested
   exception is:
   java.rmi.UnmarshalExceptionException: error
   unmarshalling arguments; nested exception is:

   java.lang.ClassNotFoundException:test.TestClient$
   ServiceListener_Stub
```

- Server could not download the remote event listener stub class from the client
  - See if stub was generated correctly (via RMIC)
  - See if listener object was exported (via .exportObject() method)
  - See if RMI codebase is set correctly by the client



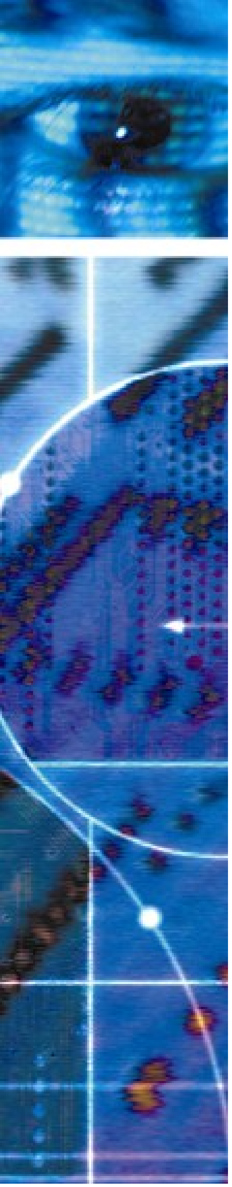
Typical RMI codebase Symptom

- Things are working fine but when client and server are on different machines, I get ClassNotFoundException
  - Very likely due to the fact that the class files are not available anymore
    - Do not use CLASSPATH for downloadable files
      - Do **use** RMI codebase
    - Do not use “localhost”



Implementation Guideline

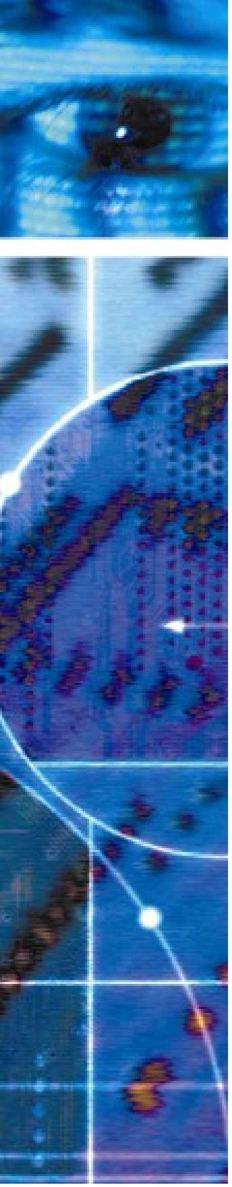
- Client has remote interface class file in its local classpath (unless it uses reflection)
- The classes that are needed for implementation should be downloadable from the server
  - Stub classes
  - Interface classes
    - Needed when client does not have interface classes in its local path
  - Any other classes that the stub and interface refers to
- Make jar file in the form of xxx-dl.jar



Example

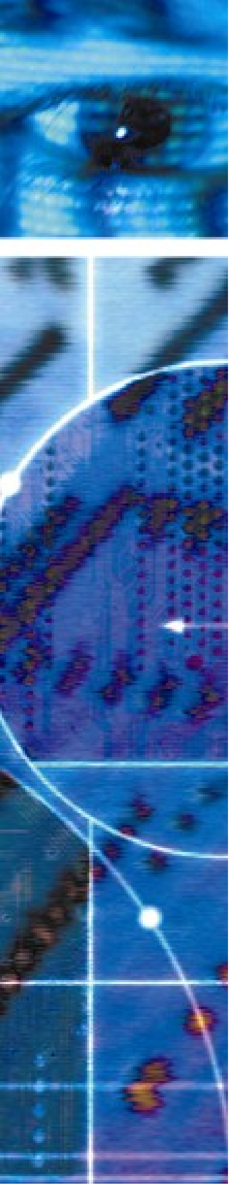
- eventg/buildEventGenerator & eventg/runEventGenerator

```
[daydreamer] java -Djava.security.policy=/home/sang/src/examples/lease/policyEventGenerator
-Djava.rmi.server.codebase=http://daydreamer:8081/EventGenerator-srvc-dl.jar
http://daydreamer:8081/EventGenerator-attr-dl.jar -jar
/home/sang/jars/EventGenerator.jar daydreamer

[daydreamer] jar -tvf EventGenerator-srvc-dl.jar
     0 Mon Mar 22 13:04:56 EST 1999 META-INF/
    66 Mon Mar 22 13:04:56 EST 1999 META-INF/MANIFEST.MF
   982 Mon Mar 22 13:04:04 EST 1999 examples/eventg/EventGenerator.class
  7933 Mon Mar 22 13:04:20 EST 1999
examples/eventg/EventGeneratorImpl_Stub.class
  1532 Mon Mar 22 13:03:52 EST 1999 examples/eventg/TestLease.class
   911 Mon Mar 22 13:03:52 EST 1999 examples/eventg/TestLeaseMap.class
  1554 Mon Mar 22 13:04:00 EST 1999 examples/eventg/TestEventLease.class
   967 Mon Mar 22 13:04:00 EST 1999 examples/eventg/TestEventLeaseMap.class
   410 Mon Mar 22 13:03:56 EST 1999 examples/eventg/TestEvent.class

[daydreamer] jar -tvf EventGenerator-attr-dl.jar
     0 Mon Mar 22 13:05:14 EST 1999 META-INF/
    66 Mon Mar 22 13:05:14 EST 1999 META-INF/MANIFEST.MF
   752 Mon Mar 22 13:05:10 EST 1999 net/jini/lookup/entry/ServiceInfo.class
  1764 Mon Mar 22 13:05:12 EST 1999
com/sun/jini/lookup/entry/BasicServiceType.class
```



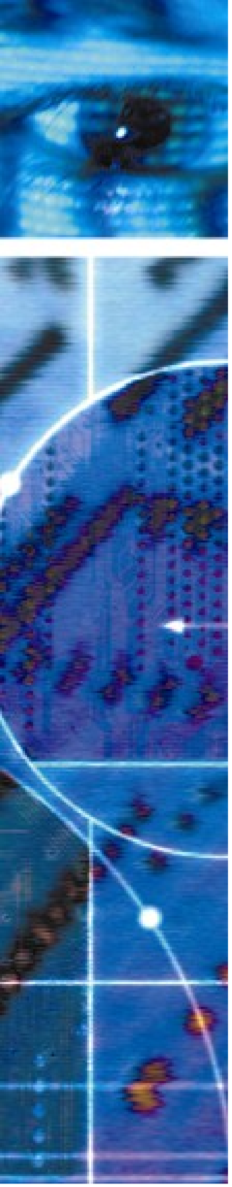
Trouble-shooting methods

- Run HTTP server in verbose mode (Example next slide)
  - Will display all the jar or class files being downloaded
- Set “-Djava.rmi.loader.logLevel=VERBOSE” on RMI client (Example next slide)
  - Will tell which class file is being downloaded from which location
- Try “javap -classpath <pathlist or jar files> <classname>” on command line (Example next slide)
  - Will tell what is really missing
- See if you can access the jar file using a browser
  - “Save as” dialog box pops up if the file is accessible
- Try FTP URL notation (instead of HTTP)
  - If it works, HTTP has a problem



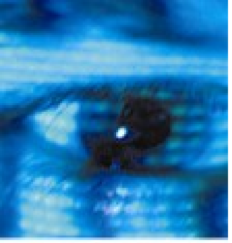
Running HTTP server in verbose mode

```
[daydreamer] java -cp /files/jini1_0/lib/tools.jar com.sun.jini.tool.ClassServer
    -port 8081   -dir /home/sang/jars -verbose

java -cp /home/sang/files/jini1_0/lib/tools.jar com.sun.jini.tool.ClassServe
ort 8081 -dir /home/sang/jars -verbose
RegRemoteAndProvideLease-srvc-dl.jar from daydreamer:65296
RegRemoteAndProvideLease-srvc-dl.jar from daydreamer:33431
RegRemoteAndProvideLease-srvc-dl.jar from daydreamer:33797
DiscoveryByGroup-srvc-dl.jar from daydreamer:37616
DiscoveryByGroup-srvc-dl.jar from daydreamer:37617
DiscoveryByGroup-attr-dl.jar from daydreamer:37620
DiscoveryByGroup-attr-dl.jar from daydreamer:37621
DiscoveryByLocator-srvc-dl.jar from daydreamer:37886
DiscoveryByLocator-srvc-dl.jar from daydreamer:37887
```

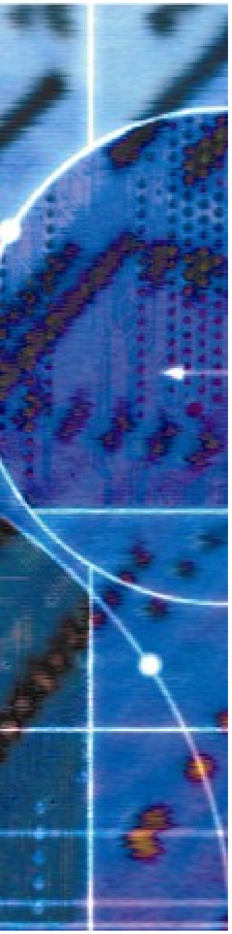


-Djava.rmi.loader.logLevel=VERBOSE

```
[daydreamer] java
        -Djava.security.policy=/home/sang/src/examples/client/policyLookupSrvcAndInvoke                -Dsun.rmi.loader.logLevel=VERBOSE
                                                   -jar /home/sang/jars/LookupSrvcAndInvoke.jar daydreamer

groupsWanted[0] = daydreamer
Waiting For Discovery to Complete

Wed Mar 17 07:43:01 EST 1999:loader:unicast discovery:LoaderHandler.loadClass: loading class "com.sun.jini.reggie.RegistrarProxy" from
        [http://daydreamer:8080/reggie-dl.jar]
.Wed Mar 17 07:43:02 EST 1999:loader:unicast discovery:LoaderHandler.loadClass: loading class "com.sun.jini.reggie.RegistrarImpl_Stub"
        from [http://daydreamer:8080/reggie-dl.jar]
LookupDiscoveryListener:  discovered...
  Lookup on host jini://daydreamer/:
    regGroups[0] belongs to Group: myGroup
    regGroups[1] belongs to Group: daydreamer
...........
Discovery of Available Lookups Complete.
Query each Lookup for known Services, the Invoke ...
Lookup Service on Host: jini://daydreamer/
  Belongs to Group: daydreamer
Wed Mar 17 07:43:13 EST 1999:loader:main:LoaderHandler.loadClass: loading class "com.sun.jini.lookup.entry.BasicServiceType" from
        [http://daydreamer:8080/reggie-dl.jar]
Wed Mar 17 07:43:13 EST 1999:loader:main:LoaderHandler.loadClass: loading class "net.jini.lookup.entry.ServiceInfo" from
        [http://daydreamer:8080/reggie-dl.jar]
Wed Mar 17 07:43:13 EST 1999:loader:main:LoaderHandler.loadClass: loading class "com.sun.jini.lookup.entry.BasicServiceType" from
        [http://daydreamer:8080/sun-util.jar, http://daydreamer:8081/RegRemoteAndProvideLease-srvc-dl.jar,
        http://daydreamer:8081/RegRemoteAndProvideLease-attr-dl.jar]
Wed Mar 17 07:43:13 EST 1999:loader:main:LoaderHandler.loadClass: loading class "net.jini.lookup.entry.ServiceInfo" from
        [http://daydreamer:8080/sun-util.jar, http://daydreamer:8081/RegRemoteAndProvideLease-srvc-dl.jar,
        http://daydreamer:8081/RegRemoteAndProvideLease-attr-dl.jar]
```



javap

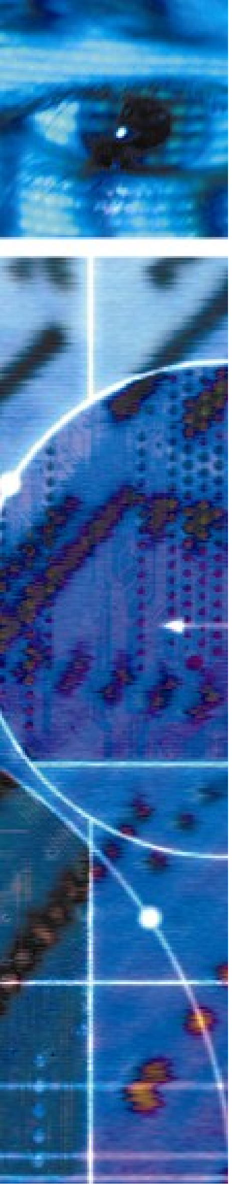


```
[daydreamer:291] javap -classpath LookupSrvcAndInvoke.jar examples/lease/TestLease
Class 'examples/lease/TestLease' not found

[daydreamer:289] javap -classpath RegRemoteAndProvideLease-srvc-dl.jar examples/lease/TestLease
Error: No binary file 'AbstractLease’

[daydreamer:326] javap -classpath RegRemoteAndProvideLease.jar:sun-util.jar examples/lease/TestLease
Error: No binary file 'Lease'

[daydreamer:332] javap -classpath RegRemoteAndProvideLease.jar:sun-util.jar:jini-core.jar
        examples/lease/TestLease
Compiled from TestLease.java
public class examples/lease/TestLease extends com.sun.jini.lease.AbstractLease {
   protected final examples.lease.RegRemoteAndProvideLease server;
   protected final java.lang.String leaseID;
   protected examples/lease/TestLease(examples.lease.RegRemoteAndProvideLease,java.lang.String,long);
   public boolean canBatch(net.jini.core.lease.Lease);
   public void cancel() throws net.jini.core.lease.UnknownLeaseException, java.rmi.RemoteException;
   public net.jini.core.lease.LeaseMap createLeaseMap(long);
   public long doRenew(long) throws net.jini.core.lease.UnknownLeaseException, java.rmi.RemoteException;
   java.lang.String getLeaseID();
   examples.lease.RegRemoteAndProvideLease getRegRemoteAndProvideLease();
   void setExpiration(long);
}
```

javap

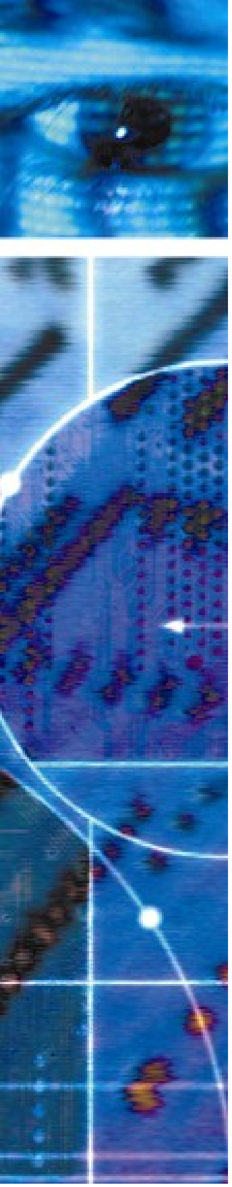
- admin/AdminServer registers with a lookup service without including OurOwnAdmin class file in its downloadable jar
  - You will see unknown service on the Lookup browser

```
[daydreamer:230] cd ~sang/jars
[daydreamer:232] ls -lat Admin*
-rw-rw----   1 sang     jinieast    8035 Mar 22 21:19 AdminClient.jar
-rw-rw----   1 sang     jinieast    2083 Mar 21 23:44 AdminServer-attr-dl.jar
-rw-rw----   1 sang     jinieast    4953 Mar 21 23:44 AdminServer-srvc-dl.jar
-rw-rw----   1 sang     jinieast   13560 Mar 21 23:44 AdminServer.jar

[daydreamer:229] !226
javap -classpath AdminServer-srvc-dl.jar examples/admin/AdminServerImpl_Stub
Error: No binary file 'Administrable'

[daydreamer:229] javap -classpath AdminServer-srvc-dl.jar:jini-ext.jar examples/admin/AdminServerImpl_Stub
Error: No binary file 'DestroyAdmin'

[daydreamer:229] javap -classpath AdminServer-srvc-dl.jar:jini-ext.jar:sun-util.jar examples/admin/AdminServerImpl_Stub
Error: No binary file 'OurOwnAdmin'
```

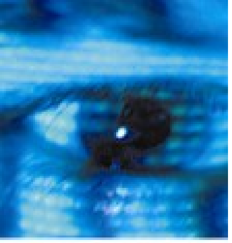


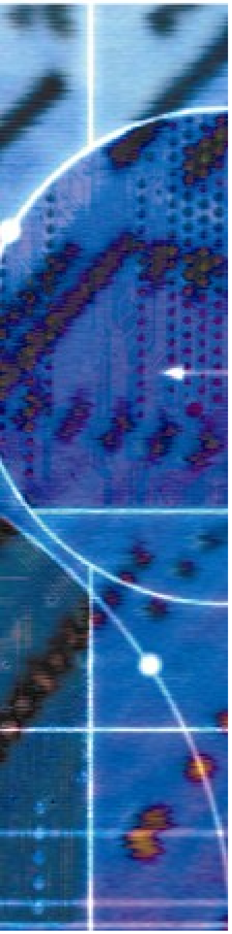
Review Points

- RMI codebase
  - Used for exporting class files
    - Serialized object has codebase annotation
  - Set via *java.rmi.server.codebase* property
  - Cause of most of *ClassNotFoundException* problems



ClassLoader Delegation

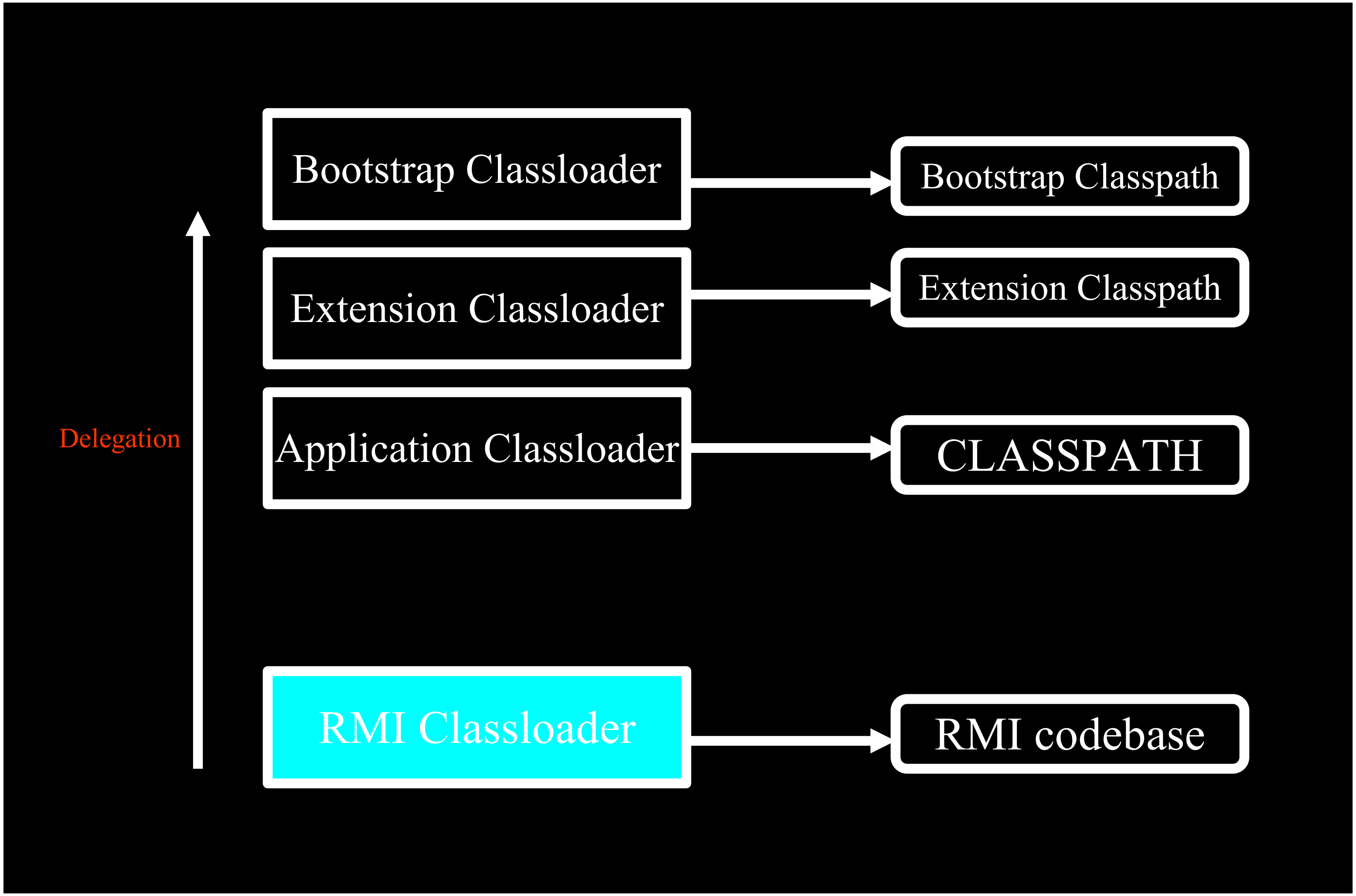




ClassLoader Delegation

- Introduced in JDK 1.2
  - Class files are searched based on classloader hierarchy
    - Bootstrap classloader
    - Extension classloader
    - Application classloader
    - RMI classloader
  - Ask parent classloader first
    - Reason why a class file in local CLASSPATH gets picked up first before the same class file gets downloaded from remote location

Classloader Hierarchy



Bootstrap Classloader

Bootstrap Classpath

Extension Classloader

Extension Classpath

Application Classloader

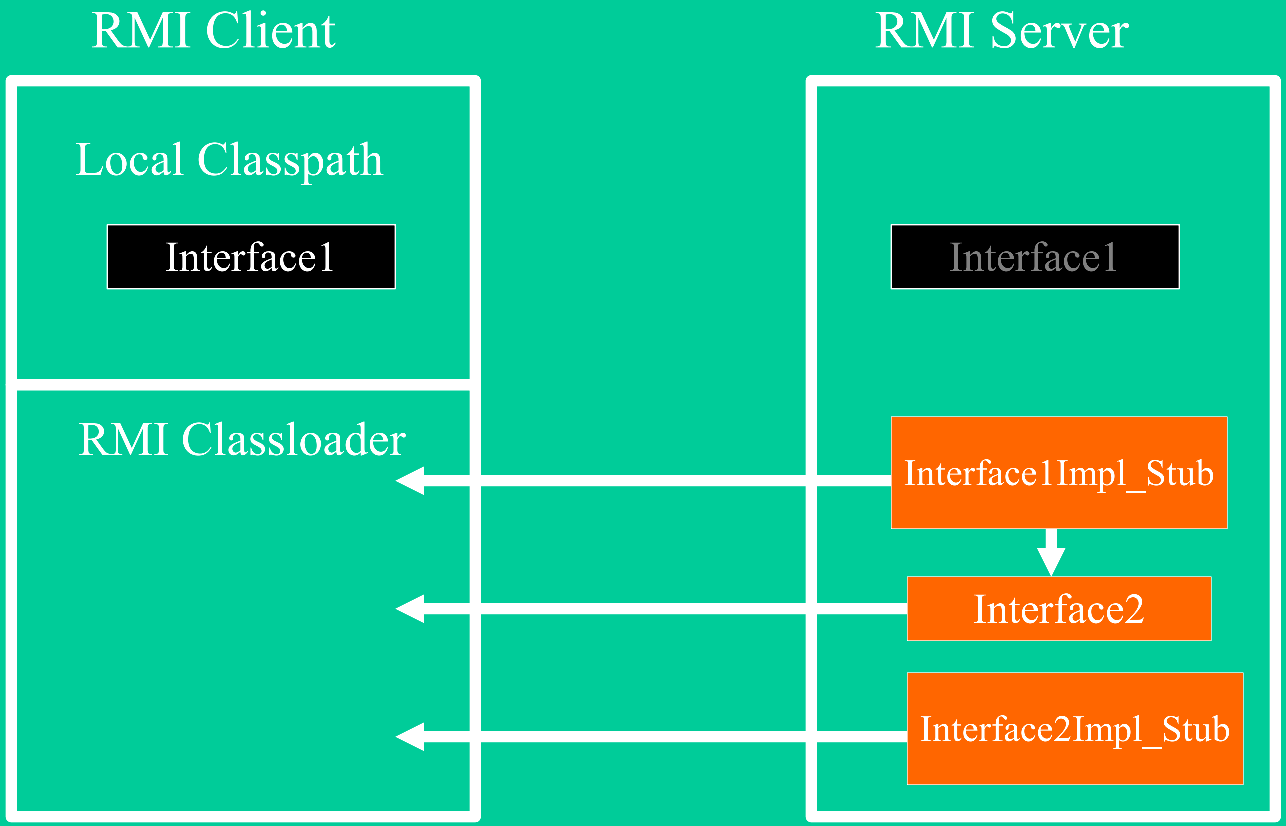
CLASSPATH

RMI Classloader

RMI codebase

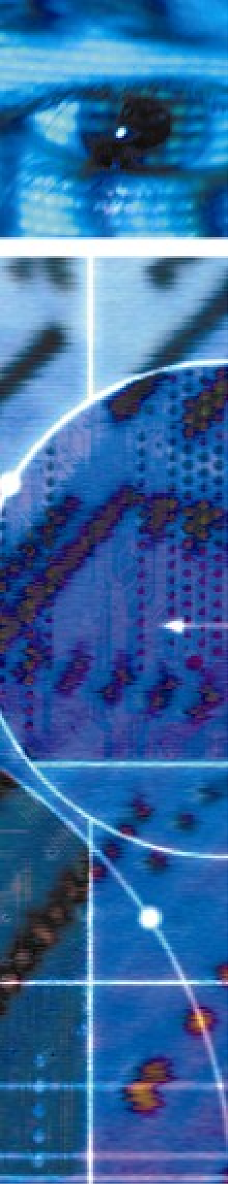
Delegation

Example


RMI Client
RMI Server
Local Classpath
Interface1
RMI Classloader
Interface1
Interface1Impl_Stub
Interface2
Interface2Impl_Stub

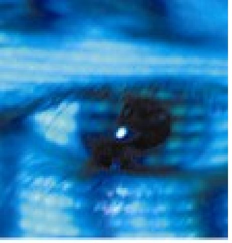
Activation

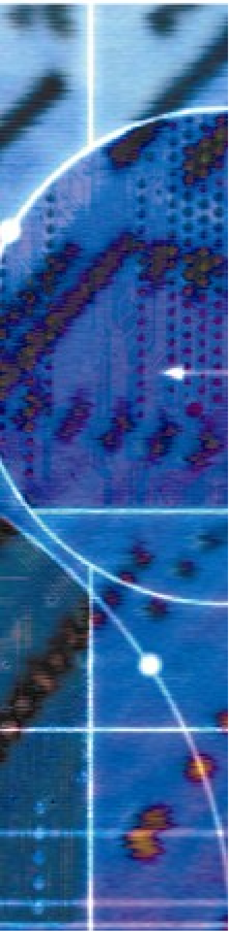
JINI



Activation

- Why activatable objects?
  - Service could be shut down inadvertently or intentionally
  - Activatable service gets restarted automatically when system boots up or on-demand basis
    - Activatable service needs to be started (registered with RMID) only once
- Activation system components
  - RMID (Activation system daemon)
  - RMID log file
    - Persistently stores all activatable objects
    - Default is <Directory where RMID gets started>/log directory
  - Activatable services
    - They are run as child processes of RMID





Control Flow of Activation

[A new activatable service with running RMID]

(5) Client, via lookup operation, retrieves the proxy object, which contains the RMI reference

(6) Client Stub talks to the service directly and gets an exception since the service (as an RMI server) is inactive

(7) Client Stub then talks to RMID

(9) Client now can talk directly with the service

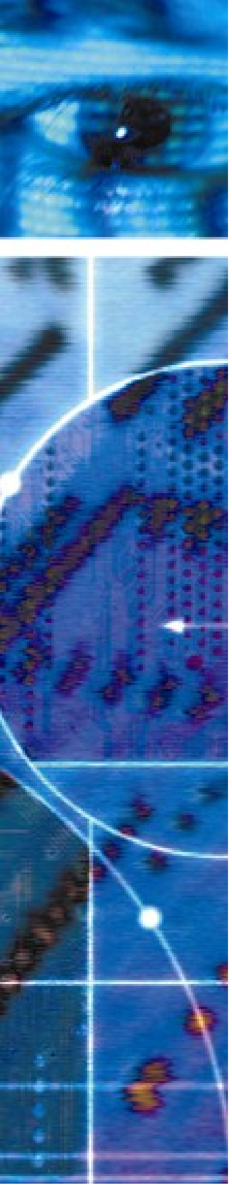
(1) RMID running

(2) A new service registers with RMID and gets a special RMI reference -RMID logs the information in persistent storage

(3) The service (actually the proxy object) registers with the lookup service - the proxy object contains the RMI reference

(4) The service goes inactive (intentionally or inadvertently)

(8) RMID restarts the service if necessary in a new VM



Control Flow of Activation

[RMID crash and reboot]

(5) Client, via lookup operation, retrieves the proxy object, which contains the RMI reference

(6) Client talks to the service directly .

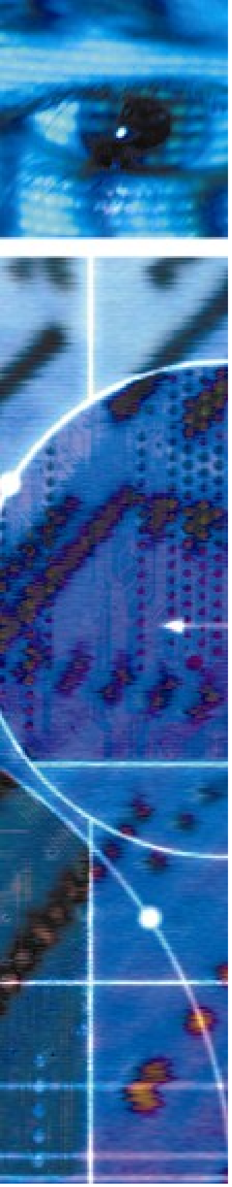
(1) A service is registered with RMID

(2) RMID crashes and reboots

(3) RMID reads the log file and restarted the services (the ones which set the RESTART flag during the registration with RMID)

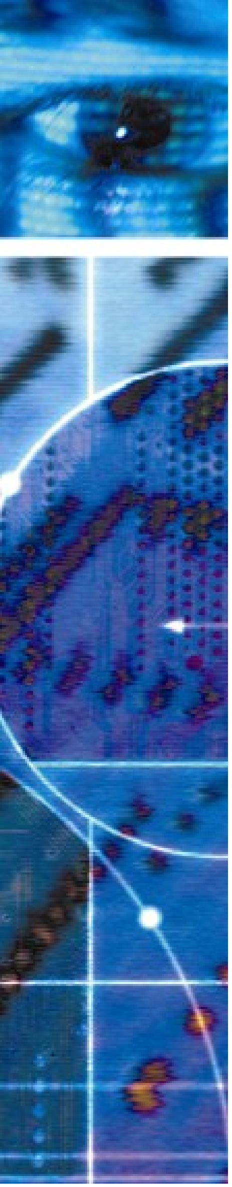
.

.



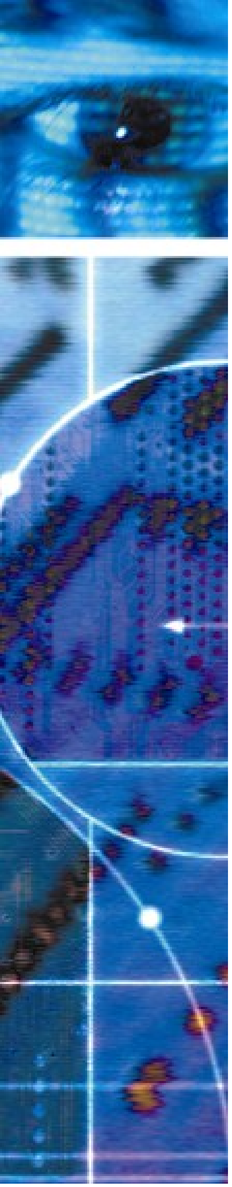
RMID

- As long as RMID is running and RMID log file is persistent, a service can get started on “as needed” basis
- Methods of destroying a service
  - Kill RMID and remove RMID log file
    - Other services will be destroyed as well
    - Sledge hammer approach
  - Use com.sun.jini.admin.DestroyAdmin interface’s destroy() method if the service supports it
    - Recommended approach



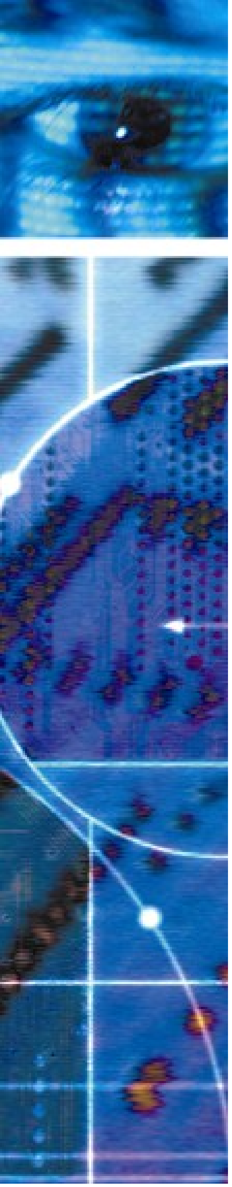
Activation Trouble-shooting

- java.rmi.activation.ActivationException: ActivationSystem not running
    - Possibly DNS lookup problem
    - Try CTRL-\ (Solaris) and CTRL-BREAK (Win32) for stack trace
- Start RMID with
    - -J-Dsun.rmi.server.activation.debugExec=true
- For any RMI properties you want to set for activatable services (child processes of RMID), start RMID with “-C-Dproperty=value”
    - -C-Djava.rmi.server.logCalls=true



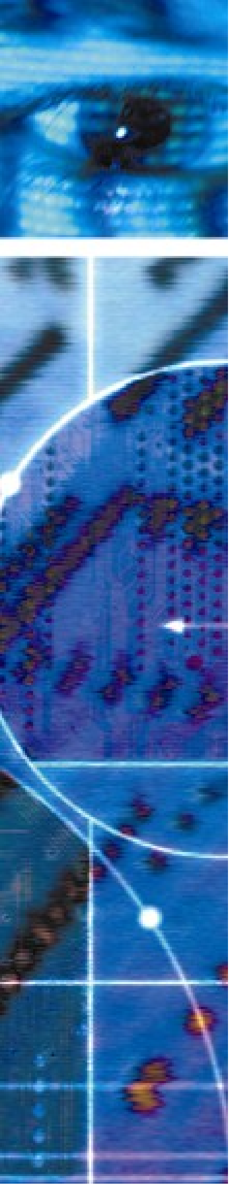
RMI Tunneling

JINI



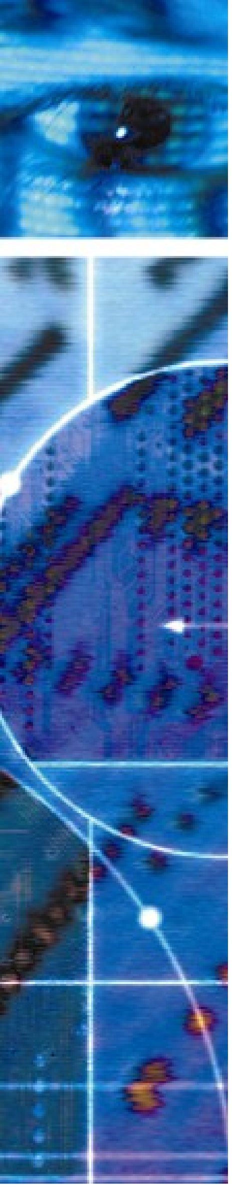
RMI Tunneling

- Features
  - Protocol runs over HTTP protocol
  - Allows RMI client within a firewall to talk to an RMI server outside of the firewall
- Limitation
  - RMI server cannot talk back to the RMI client
- Implications to Jini
  - No multicast discovery
    - Have to use Unicast
  - No event notification from RMI server to RMI client



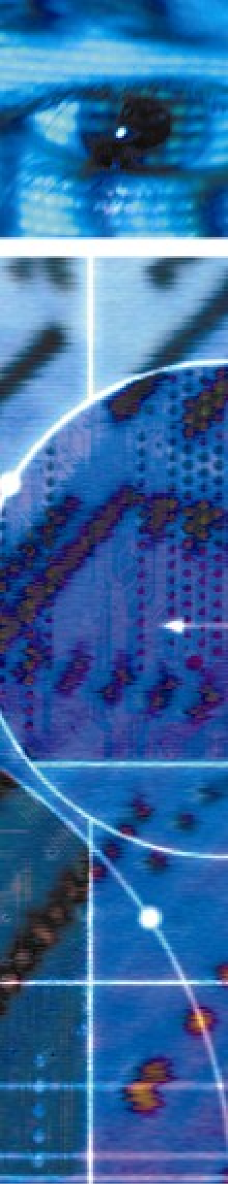
RMI Security

JINI



Java Security

- In Java, SecurityManager handles security control
  - Based on security policy file
  - Security policy define “permission control” based on
    - Where the code came from
    - Who signed the code
    - Examples
      - All code signed by Dave can write to a particular directory
      - Any code downloaded from a particular HTTP server site has no filesystem access

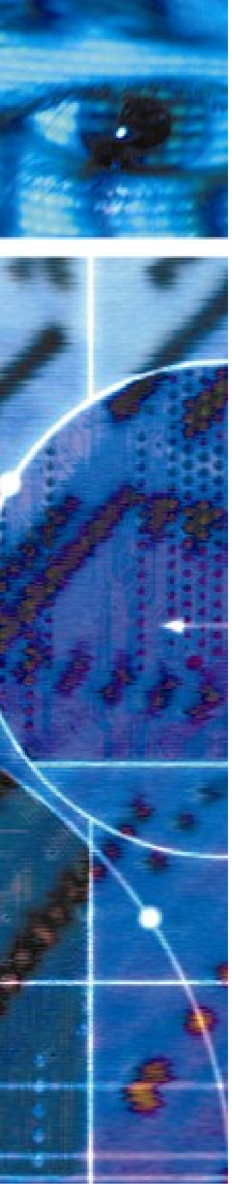


Security Policy Example

- Give all all permission to any code

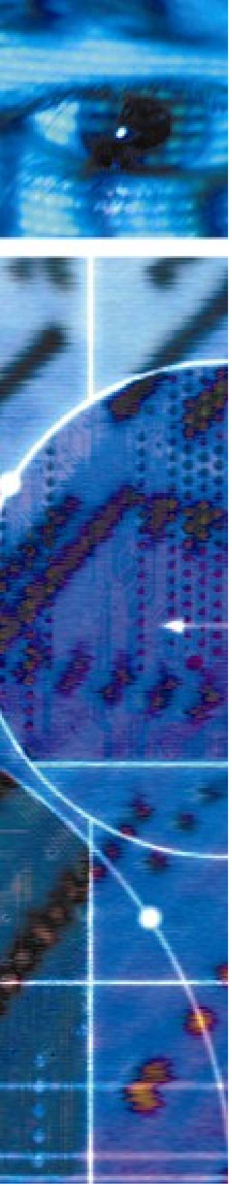
```
grant {
    permission java.security.AllPermission "", "";
};
```

- Use the above “all permission to all” only during testing
  - Never use it in production environment



Security Policy Example

```
grant codebase "file:${java.class.path}" {
   // file system dependent permissions for unix file system
   permission java.io.FilePermission "./*", "read,write,execute,delete";
   permission java.io.FilePermission "/tmp", "read,write,execute,delete";
   permission java.io.FilePermission "/tmp/-", "read,write,execute,delete";
   permission java.io.FilePermission "/var/tmp", "read,write,execute,delete";
   permission java.io.FilePermission "/var/tmp/-", "read,write,execute,delete";
   // uncomment this one if you need lookup to accept file: codebases
   // permission java.io.FilePermission "<<ALL FILES>>", "read";
   permission java.lang.RuntimePermission "modifyThreadGroup";
   permission java.lang.RuntimePermission "modifyThread";
   permission java.net.SocketPermission "*:1024-", "connect,accept";
   // for http: codebases
   permission java.net.SocketPermission "*:80", "connect";
   permission java.net.SocketPermission "224.0.1.84", "connect,accept";
   permission java.net.SocketPermission "224.0.1.85", "connect,accept";
   permission java.util.PropertyPermission "java.rmi.server.hostname", "read";
   permission java.util.PropertyPermission "com.sun.jini.reggie.*", "read";
   permission java.util.PropertyPermission "net.jini.discovery.*", "read";
   permission net.jini.discovery.DiscoveryPermission "*";
   // file system dependent permissions for windows file system
   permission java.io.FilePermission ".\\*", "read,write,execute,delete";
   permission java.io.FilePermission "c:\\temp", "read,write,execute,delete";
   permission java.io.FilePermission "c:\\temp\\-", "read,write,execute,delete";
   permission java.io.FilePermission "c:\\windows\\temp", "read,write,execute,delete";
   permission java.io.FilePermission "c:\\windows\\temp\\-", "read,write,execute,delete";
   // Deleted the rest
};
```

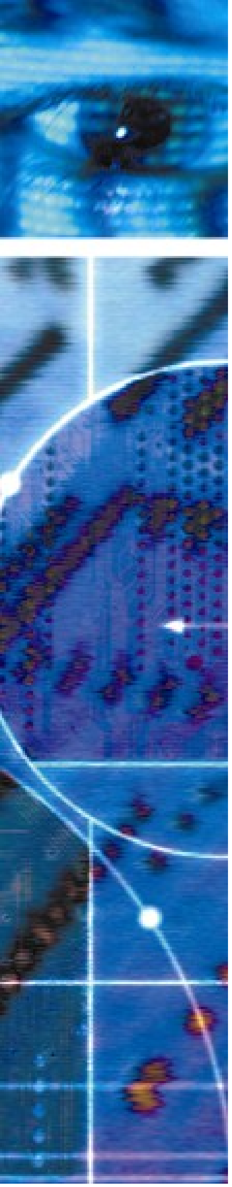


RMI Security

- Security is a serious concern since executable code is being downloaded from remote location
- In RMI, *SecurityManager* has to be installed in order to be able to download any code from remote location
  - ◆ Without its installation, RMI will search for class files only from local classpath
- The security policy file further specifies the “permission control”

RMI Security

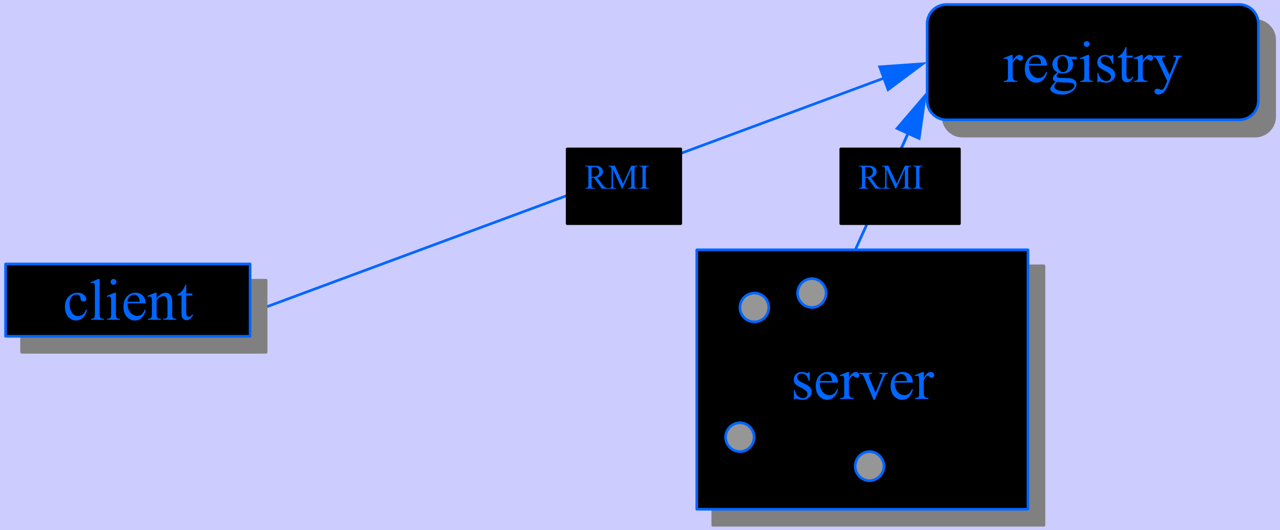
- RMI client needs to install security manager because it needs to download Stub file of RMI object
- A simple RMI server might not need to install security manager if it does not need to download class files from remote location
  - It is still good practice to install it anyway



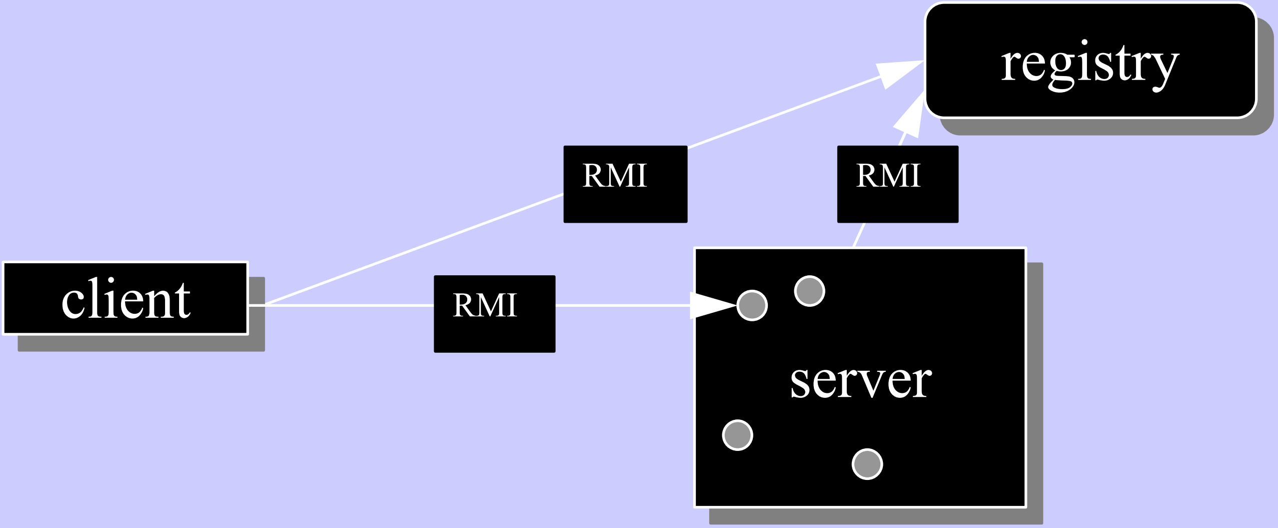
Review Points

JINI™

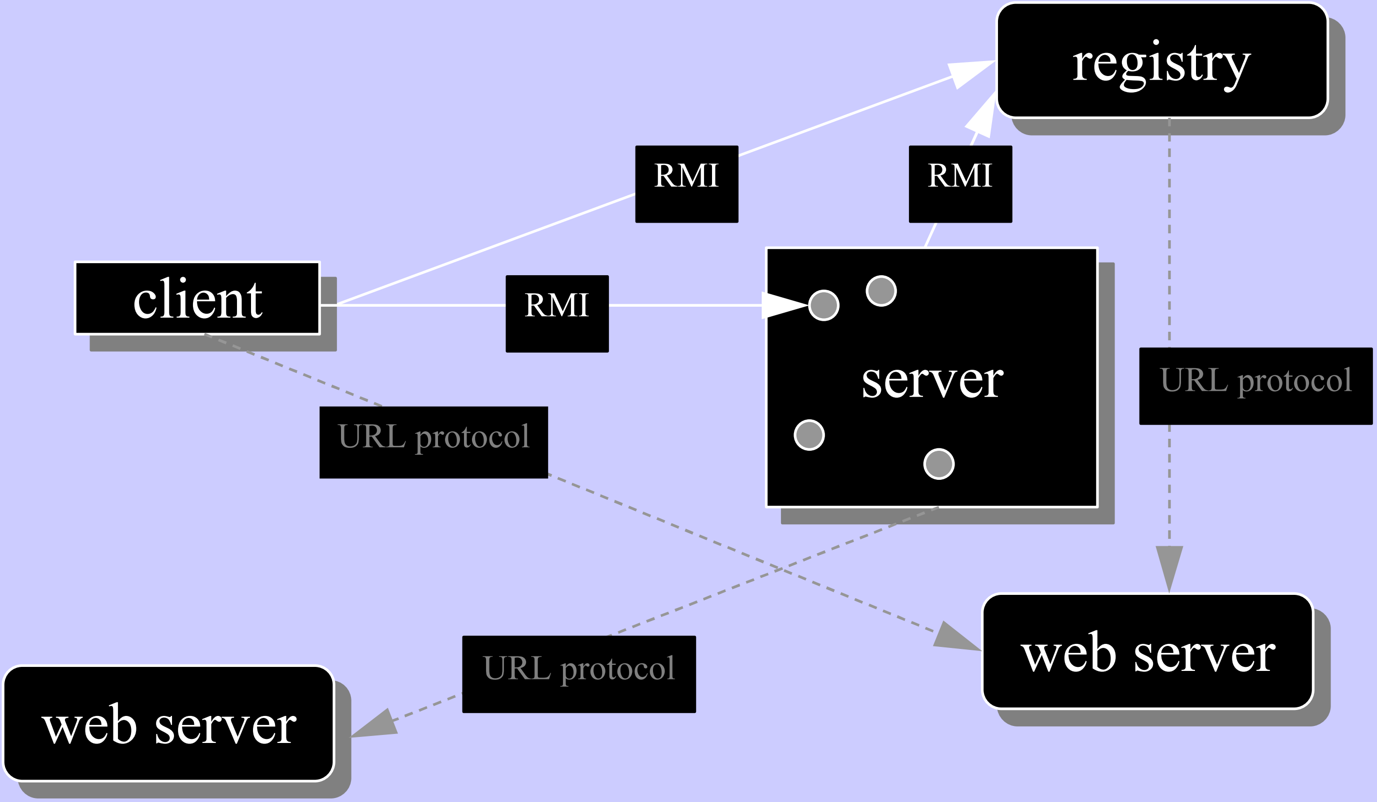
Locating Remote Objects



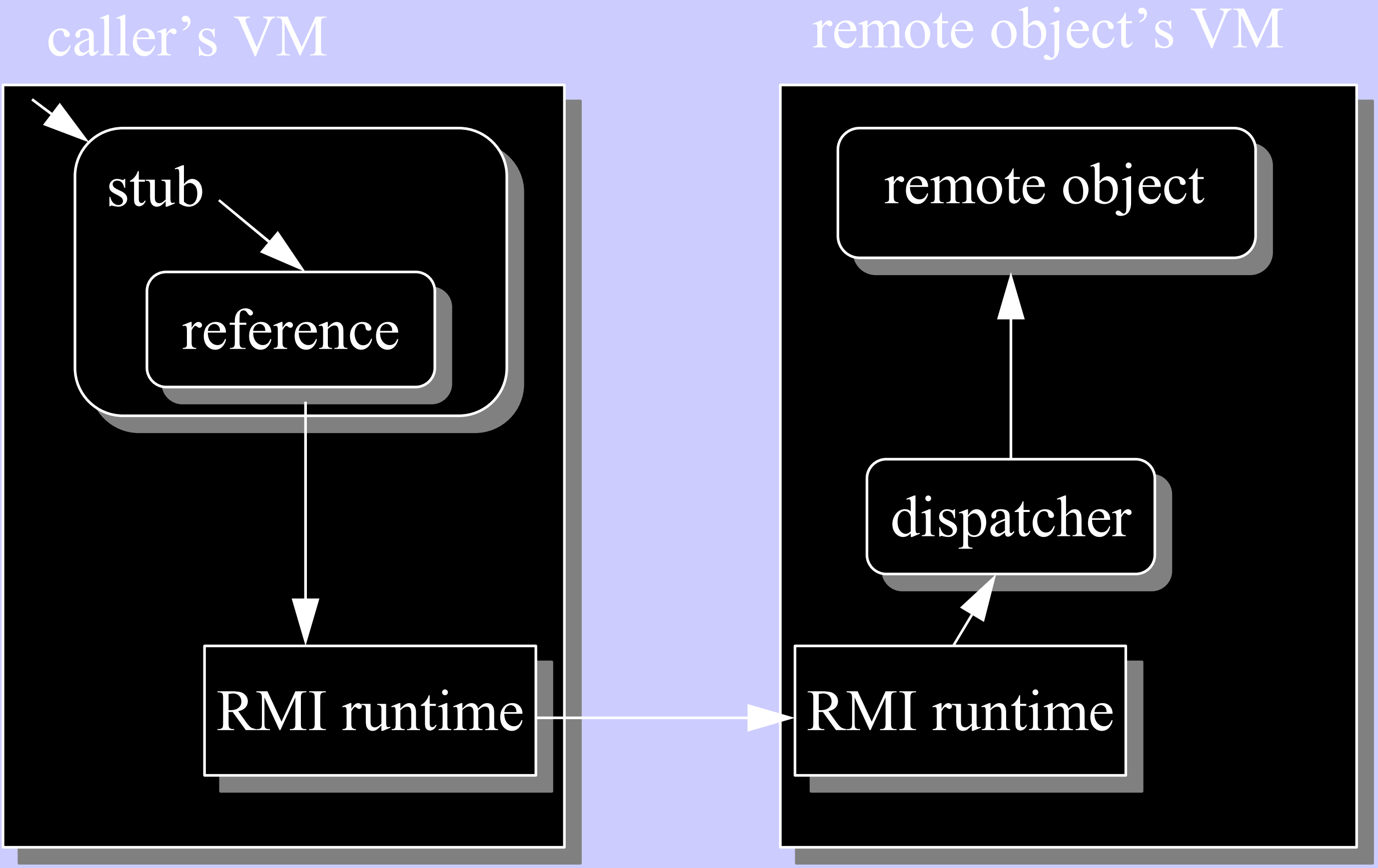
Remote Communication

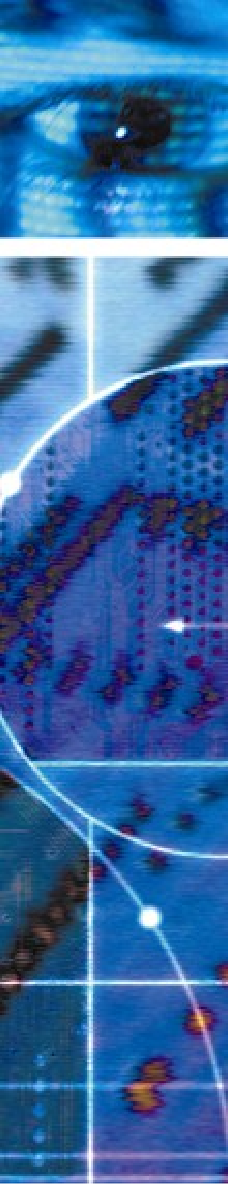


Loading Classes



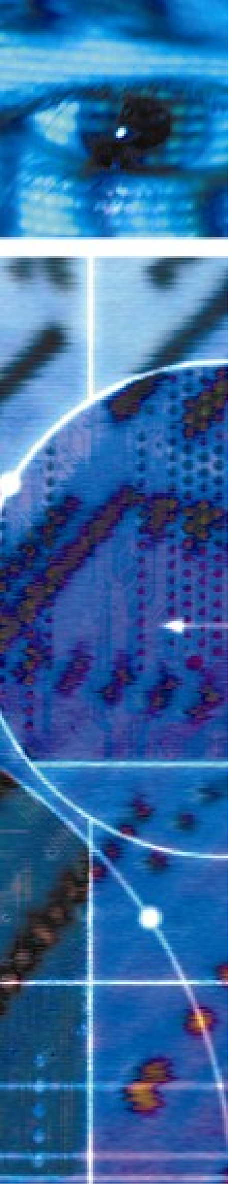
Method Invocation





RMI Limitation

- Client and server paradigm
  - Client has to know about the server
    - where the server is
    - how to reach the server
    - what the server can do
  - If the server becomes unavailable, the client generally fails too



Summary

- RMI is for invoking methods of remote Java object
- Enables the movement of data and code
  - Data (State of object) movement via serialized object
  - Code movement via class downloading

Thank you!

Sang Shin

Michèle Garoche

http://www.javapassion.com

“Learn with Passion!”