Distributed Systems

RMI

Prof. Agostino Poggi

What Is RMI?

- Its acronym means Remote Method Invocation
- Supports communication between **different** Java **virtual machines**, potentially across the network
- Allows an object running in one Java **virtual machine** to invoke methods on an object running in **another** Java **virtual machine**
- Uses the same syntax and semantics used for non-distributed programs

Main Features

- Locate remote objects
  - Their **references** can be obtained by a **naming facility** called **RMI registry**
  - Their **references** can be passed or returned through **remote invocations**
- Hide details of communication with remote objects
  - Details of communication handled by RMI
  - To the programmer, remote communication looks similar to regular Java method invocations
- Load class definitions for remote objects
  - RMI provides mechanisms for **loading class definitions** as well as for **transmitting class instances**

Remote Interface

- RMI does not have a separate IDL
  - RMI is used between Java virtual machines
  - Java already includes the concept of interfaces
- An Interface to be a remote interface needs to extend the interface **java.rmi.Remote**
- All methods in a remote interface must be declared to throw the **java.rmi.RemoteException** exception

Remote Class

- A class to be a **remote class** needs to **implement** a **remote interface**
- A remote class also extends the library class **java.rmi.server.UnicastRemoteObjec**t
  - This class includes a constructor that **exports** the object to the **RMI system** when it is created, thus making the object visible to the outside world
- A common convention is to name the remote class **appending *Impl*** to the **name** of the corresponding **remote interface**

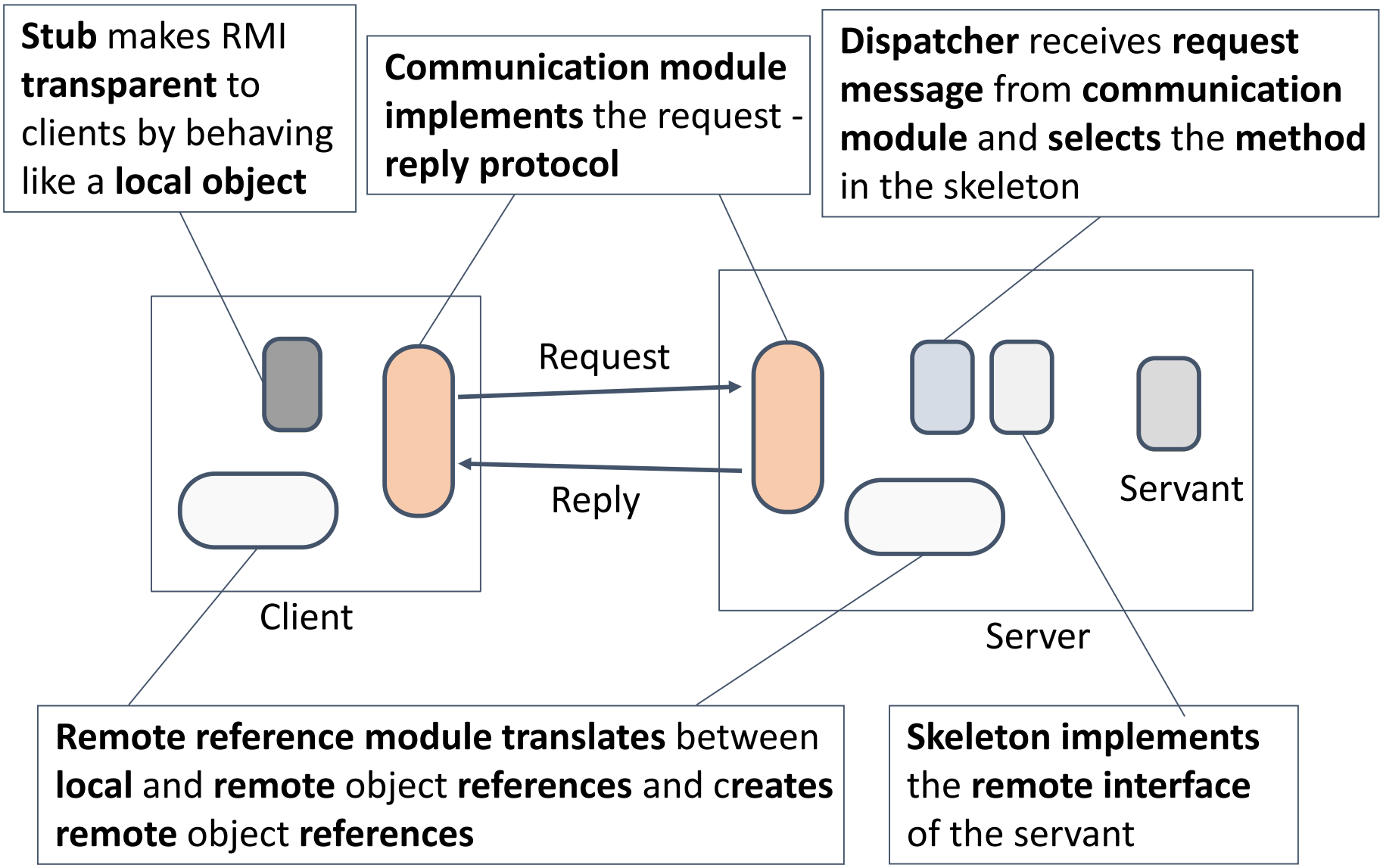
Remote Method

- A client can **request** the execution of all those methods that are declared inside a **remote interface**
- A client request can contain **some parameters**
- Parameters can be of three types
  - **Primitive types**
  - **Objects**
  - **Remote objects**
- **Remote call** execution is **based** on **reflection**

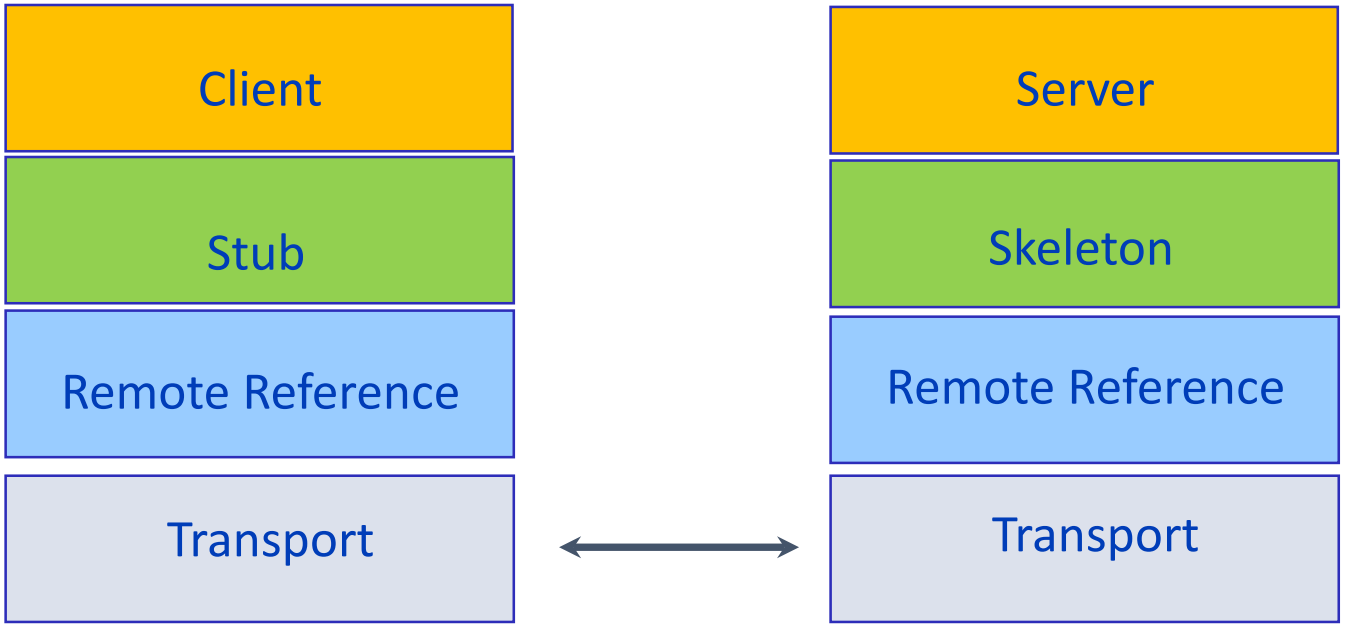
Parameter Passing

	Primitive Type	Local Object	Remote Object
Local Call	Call by Value	Call by Reference	Call by Reference
Remote Call	Call by Value	 Call by Value	Call by Reference

Application Components



RMI Stack


Client
Stub
Remote Reference
Transport
Server
Skeleton
Remote Reference
Transport

Remote Call Execution

- Stub has to marshal information about a method and its arguments into a request message
  - An object of class **Method**
  - An **array of objects** for the method's arguments
- Dispatcher and skeleton
  - Obtains the **remote** object **reference**
  - Unmarshals the **Method** object and its arguments
  - Calls the **invoke** method on the object reference and on the array of arguments values
  - Marshals the **result** or any **exceptions** into the **reply** message

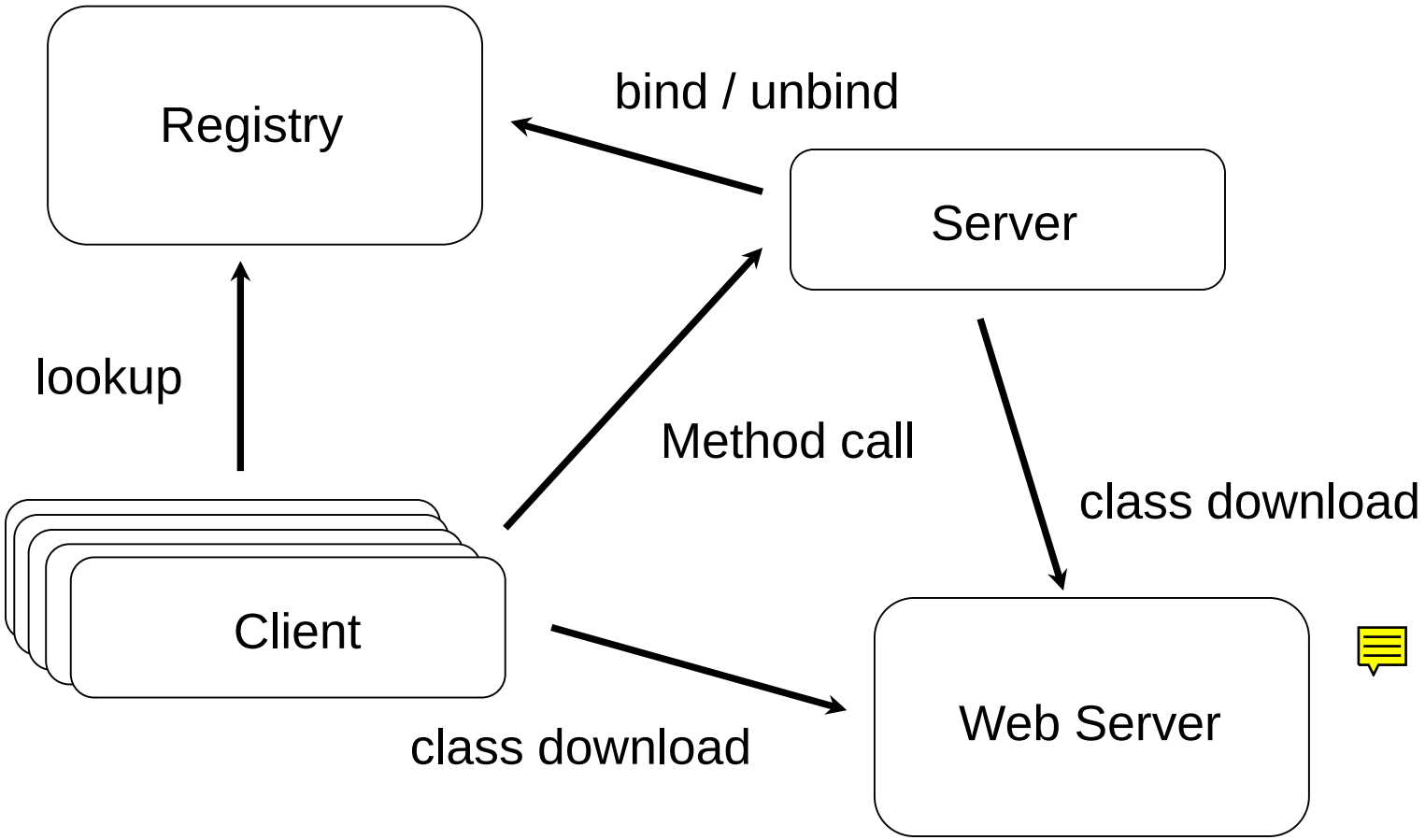
Remote Reference Layer

- Client Side
  - Knows if the **remote object (**still) exists
  - Knows **where** to **locate server** holding the remote object
- Server Side
  - Knows if the **local object** exists
  - Knows where to **locate** the **local implementation**

Transport Layer

- **Manages connection** between Java virtual machines
- Used over the **network** and on the **local host**
- There is a **messaging protocol** implemented **over TCP/IP**

Application Components



RMI Registry

- Runs on **each machine**, which **hosts remote objects** and **accepts requests** for **services**
- Allows the **registration** of remote interfaces and the **retrieval** of **remote objects**
  - **Registration** operations are only possible through servers running on the **same machine** of the **registry**

- Its default TCP/IP port is 1099

java.rmi.registry.Registry

Modifier and Type	Field	Description
static int	REGISTRY_PORT	Well known port for registry.

Modifier and Type	Method	Description
void	bind(String name, Remote obj)	Binds a remote reference to the specified name in this registry.
String[]	list()	Returns an array of the names bound in this registry.
Remote	lookup(String name)	Returns the remote reference bound to the specified name in this registry.
void	rebind(String name, Remote obj)	Replaces the binding for the specified name in this registry with the supplied remote reference.
void	unbind(String name)	Removes the binding for the specified name in this registry.

java.rmi.registry.LocateRegistry

Modifier and Type	Method	Description
static Registry	createRegistry(int port)	Creates and exports a Registry instance on the local host that accepts requests on the specified port.
static Registry	createRegistry(int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)	Creates and exports a Registry instance on the local host that uses custom socket factories for communication with that instance.
static Registry	getRegistry()	Returns a reference to the remote object Registry for the local host on the default registry port of 1099.
static Registry	getRegistry(int port)	Returns a reference to the remote object Registry for the local host on the specified port.
static Registry	getRegistry(String host)	Returns a reference to the remote object Registry on the specified host on the default registry port of 1099.
static Registry	getRegistry(String host, int port)	Returns a reference to the remote object Registry on the specified host and port.
static Registry	getRegistry(String host, int port, RMIClientSocketFactory csf)	Returns a locally created remote reference to the remote object Registry on the specified host and port.

Getting Remote Objects

- Explicit by **using** registry **methods**

  - Server binds/rebinds name with the registry

  - Client looks up name with the registry

- Implicit by **receiving** a remote object **reference**

TemperatureReader Interface & Class

```
public interface TemperatureReader extends Remote {
   int getTemperature() throws RemoteException;
}

public class TemperatureReaderImpl
  extends UnicastRemoteObject implements TemperatureReader
{
  private static final long serialVersionUID = 1L;

  private static final int MAX = 100;
  private static final int MIN = -100;

  private Random random;

  public TemperatureReaderImpl() throws RemoteException {
    this.random = new Random();
  }

  public int getTemperature() throws RemoteException {
    return this.random.nextInt(MAX - MIN) + MIN;
  }
}
```

TemperatureServer Class

```
public class TemperatureServer {
  private static final int PORT = 1099;

  public static void main(final String [] args) throws Exception
  {
    Registry registry = LocateRegistry.createRegistry(PORT);
    TemperatureReader service = new TemperatureReaderImpl();

    registry.rebind("temperature", service);
  }
}
```

Locates the registry

Creates a remote object

Publishes a remote reference to that object with external name "*temperature*"

TemperatureClient Class

Looks up a reference to a remote object with external name "*temperature*"

Locates the registry

```
public class TemperatureClient {
  public static void main(String [] args) throws Exception
  {
    Registry registry = LocateRegistry.getRegistry();

    TemperatureReader service =
        (TemperatureReader) registry.lookup("temperature");

    System.out.println("Temperature is "
        + service.getTemperature());
  }
}
```

Invokes the remote method on the server

Distributed Garbage Collection

- In Java, unused objects are garbage collected
  - Java virtual machine automatically **destroys objects** that are **not referenced** by anyone
- RMI implements a distributed garbage collector
- The aims of a distributed garbage collector are
  - **Retain** the local and remote **objects** when it is still be **referenced**
  - **Collect** the **objects** when **none** holds **reference** to them
- RMI garbage collection algorithm is based on **reference counting**

Distributed Garbage Collection (2/2)

- Server
  - **Counts** the number of **remote references** to each **remote object** it exports
  - When there are **no** more **local** and **remote** references to the object, the object is **destroyed**
- Client should notify when it no longer uses remote references
  - Each **remote reference** has an associated **"lease" time**
  - Client **renews** the **lease** on the used references
  - When all leases expire, then the **object** can be **destroyed**

Explicit Notification (1/2)

- A remote object class can implement the **Unreferenced** interface
  - To receive notification when there are no more clients that reference that remote object
  - To release resources
- This interface defines an **unreferenced** method
- This method is called by the RMI runtime sometime after it determines that the list of clients, referencing the remote object, becomes empty

UnrefTemperatureReaderImpl Class

```
public class UnrefTemperatureReaderImpl
    extends UnicastRemoteObject
    implements TemperatureReader, Unreferenced
{
  private static final long serialVersionUID = 1L;

  private static final int MAX = 100;
  private static final int MIN = -100;

  private Random random;

  public UnrefTemperatureReaderImpl() throws RemoteException {
    this.random = new Random();

    // Opens logging file ...
  }

  public int getTemperature() throws RemoteException {
    int t = this.random.nextInt(MAX - MIN) + MIN;

    // Saves temperature ...

    return t;
  }

  public void unreferenced() {
    // Frees resources
  }
}
```

E.g., file, network and database connections can be released

Remote Class

- RMI can **exchange objects** through serialization, but **serialized objects** do **not contain** the code of the **class** they implement 
- In fact, **de-serialization** process can be c**ompleted** only if the client has **available** the code of the **class** to be de-serialized
- **Class** code can be **provided** by **manually copy** implementation class files to the client
- A more general approach is to **publish** implementation class files on a **Web server**

Dynamic Class Loading (1/2)

- Remote object's codebase is specified by setting the **java.rmi.server.codebase** property

```
java –Djava.rmi.server.codebase=http://www.rmi.com/classes/
   TemperatureClient
```

```
java –Djava.rmi.server.codebase=http://www.rmi.com/myStub.jar
   TemperatureClient
```

- Client **requests** a **reference** to a **remote object**
- Registry **returns** a **reference** (the stub instance) to the **requested class**

Dynamic Class Loading (2/2)

- If the **class definition** for the stub instance is **found** locally in the client's **CLASSPATH**
- Then it **loads** the **class definition**
- Else it **retrieves** the **class definition** from the **remote object codebase**

Security (1/2)

- In Java a **security manager checks** if an **untrusted** code may have **access** to some system resources

- A **policy** defines which **class has** the **right** to do what depending on the class identity (i.e., its URL and its certificates)

- A **protection domain** is a set of **permissions** established by the current **policy**

Security (2/2)

- A **security manager delegates** all **decisions** to an **access controller** that decides according to
  - The **permissions** of the protection domain
  - The **checking** of the **sequence** of calls for **resource grant/denial**

Security in RMI Systems (1/2)

- Security is a serious concern since executable code is being downloaded from a remote location
- RMI **normally allows** the **loading** of code only from the **local CLASSPATH**
- RMI **allows** the **loading** of code from a remote location only if a suitable **security manager** is **set** and an appropriate **security policy** is **defined**
- RMI **clients** usually **need** to install **security manager** because they need to **download stub files**
- RMI **servers** usually do **not need** it, **but** it is still **good practice** to install it anyway

Security in RMI Systems (2/2)

- A **security manager** can be **set** as follows

  System.setSecurityManager(new RMISecurityManager());

- A **security policy** can be **defined** in a plain text file

  grant codebase { permission java.security.AllPermission };

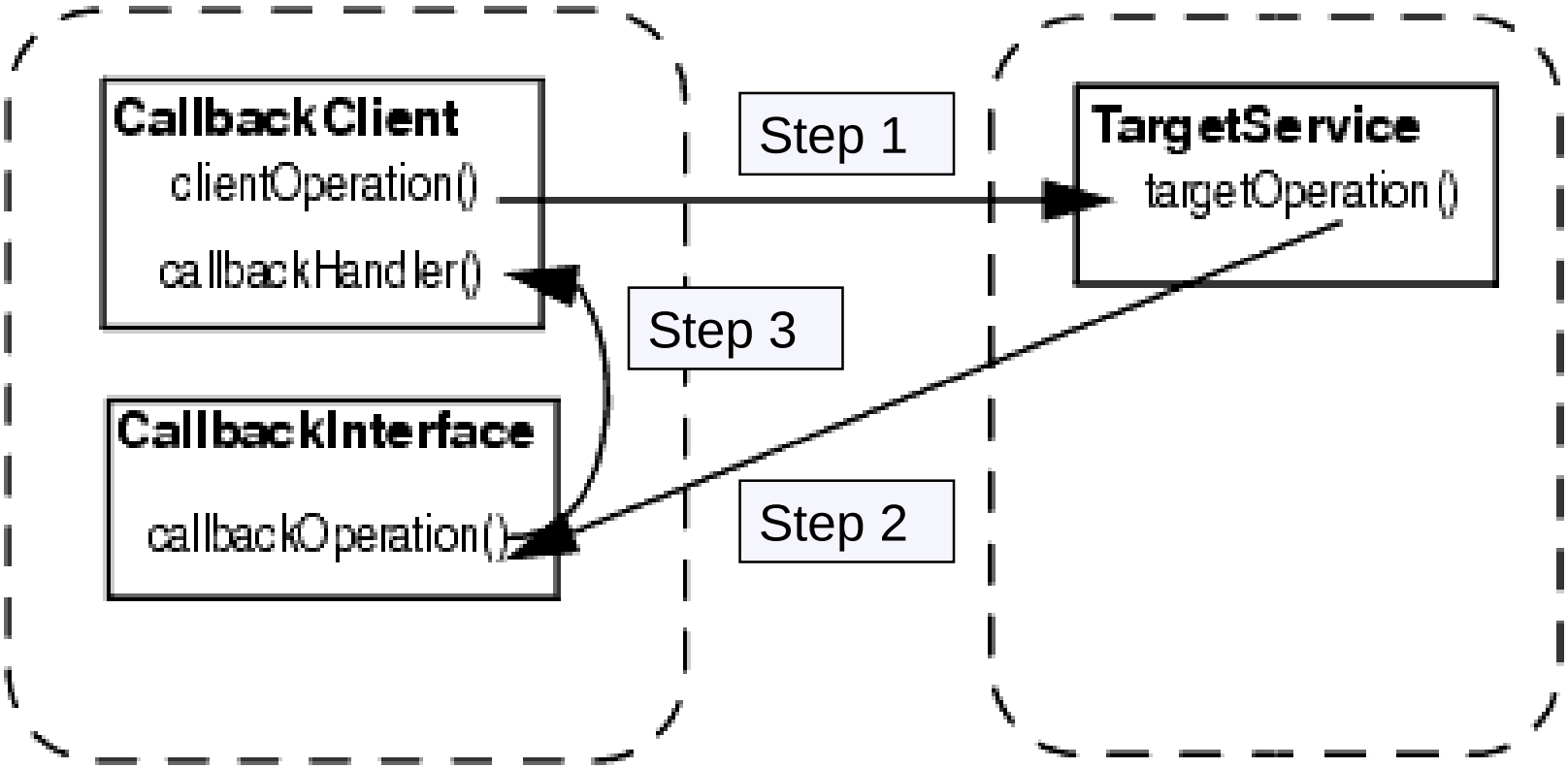
  grant codebase http://www.rmicode.com/classes/

  { permission java.security.AllPermission };

- A **security policy** can be **assigned** to the client as follows

  java –Djava.security.policy=rmi.policy TemperatureClient

Callback


CallbackClient
clientOperation()
callbackHandler()
CallbackInterface
callbackOperation()
TargetService
targetOperation()
Step 1
Step 3
Step 2

Callback Features

- Its actions **simplify** the **implementation** of the code that notify events
  - Improve performance by **avoid** constant **polling** 
  - **Delivery** information in a **timely manner**
- In a RMI based system, callbacks are implements as follows
  - Client **creates** a **remote object**
  - Client **passes** the **remote object reference** to the server
  - Whenever an event occurs, the server **calls** the **client** via the **remote object**

Subscribe Interface & Class

```
public interface Subscribe extends Remote {
  void subscribe(final TemperatureWriter w)
       throws RemoteException;
}

public class SubscribeImpl extends UnicastRemoteObject
                           implements Subscribe
{
  private static final long serialVersionUID = 1L;

  private Set<TemperatureWriter> writers;

  public SubscribeImpl(final Set<TemperatureWriter> s)
         throws RemoteException
  {
    this.writers = a;
  }

  public void subscribe(final TemperatureWriter w)
              throws RemoteException
  {
    this.writers.add(w);
  }
}
```

TemperatureWriter Interface & Class

```
public interface TemperatureWriter extends Remote
{
  void putTemperature(final int t) throws RemoteException;
}


public class TemperatureWriterImpl
       extends UnicastRemoteObject
       implements TemperatureWriter
{
  private static final long serialVersionUID = 1L;

  public TemperatureWriterImpl() throws RemoteException
  {
  }

  public void putTemperature(final int t) throws RemoteException
  {
    System.out.println(t);
  }
}
```

CallbackServer Class

```
public class CallbackServer {
  private static final int PORT = 1099;
  private static final int MAX = 100;
  private static final int MIN = -100;
  public static void main(final String[] args) throws Exception {
    Random random = new Random();
    Registry registry = LocateRegistry.createRegistry(PORT);
    Set<TemperatureWriter> writers = new CopyOnWriteArraySet<>();
    Subscribe service = new SubscribeImpl(writers);
    registry.rebind("subscribe", service);
    while (true) {
      int t = random.nextInt(MAX - MIN) + MIN;
      try {
        for (TemperatureWriter w : writers) {
          w.putTemperature(t);
        }
        Thread.sleep(1000);
      }
      catch (Exception e) {
        continue;
      }
    }
  }
}
```

Creates a remote object

Publishes a remote reference to that object with external name "*subscribe*"

Invokes the remote method on the client

CallbackClient Class

```
public class CallbackClient {
  public static void main(final String [] args)
                      throws Exception
  {
    Registry registry = LocateRegistry.getRegistry();

    TemperatureWriter w = new TemperatureWriterImpl();

    Subscribe service =
      (Subscribe) registry.lookup("subscribe");

    service.subscribe(w);
  }
}
```

Creates a remote object

Looks up a reference to a remote object with external name "*subscribe*"

Invokes the remote method on the server

Further Reading

- Plainfossé, David, and Marc Shapiro. "A survey of distributed garbage collection techniques." Memory Management. Springer, Berlin, Heidelberg, 1995. 211-249.