Remote Procedure Call and Remote Method Invocation

CMPE 273 Enterprise Distributed Systems

RPC

- 1984: Birrell & Nelson
- Mechanism to call procedures on other machines
- Remote Procedure Call

- Combine Socket programming and procedure call

Regular procedure calls

- Machine instructions for call & return but the compiler really makes the procedure call abstraction work:
  - Parameter passing
  - Local variables
  - Return data
- `x = f(a, "test", 5);`

RPC implementation

- Create **stub functions** to make it appear to the user that the call is local
- Stub function contains the function's interface
- Writing application is simplified
  - RPC hides all network code into stub functions details
  - Sockets, port numbers, byte ordering
- RPC: presentation layer in OSI model

RPC Parameter Passing

- Pass by value: copy data to network message
- Pass by reference: does not make sense without shared memory
- Copy items referenced to message buffer
  1. Send them over
  2. Unmarshal data at server
  3. Pass local pointer to server stub function
  4. Send results back

Complex data structures: copy & reconstruct

RMI

- Distribute objects across different machines to take advantage of hardware and software
- Developer builds network service and installs it on a machine
- User requests an instance of a class using URL syntax
- User uses object as if it were a local object

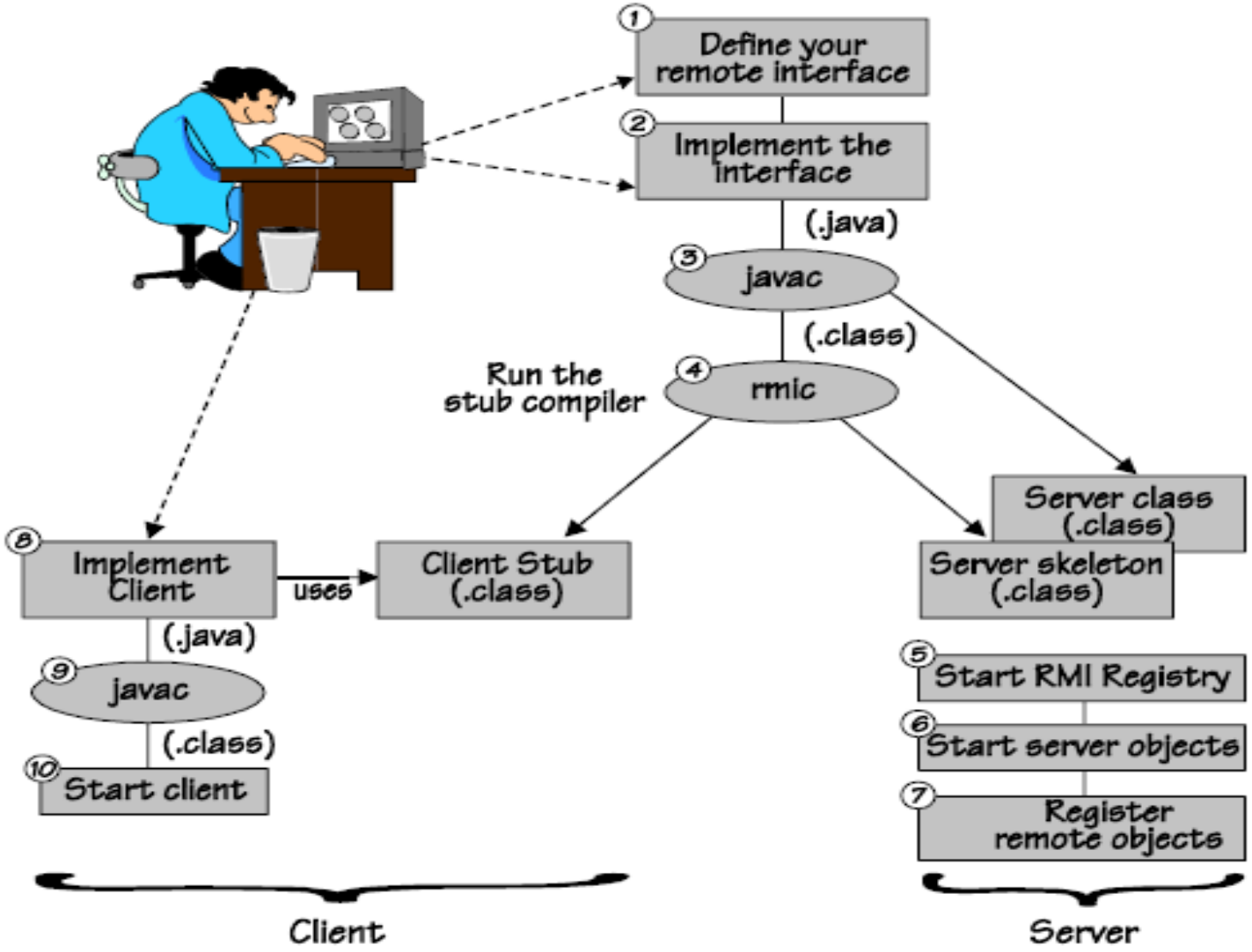
RMI

- **RMI is built for Java only!**
  - No goal of OS interoperability (as CORBA)
  - No language interoperability(goals of SUN, DCE, and CORBA)
  - No architecture interoperability
- **No need for external data representation**
  - All sides run a JVM
- **Benefit: simple and clean design**

RMI Operations

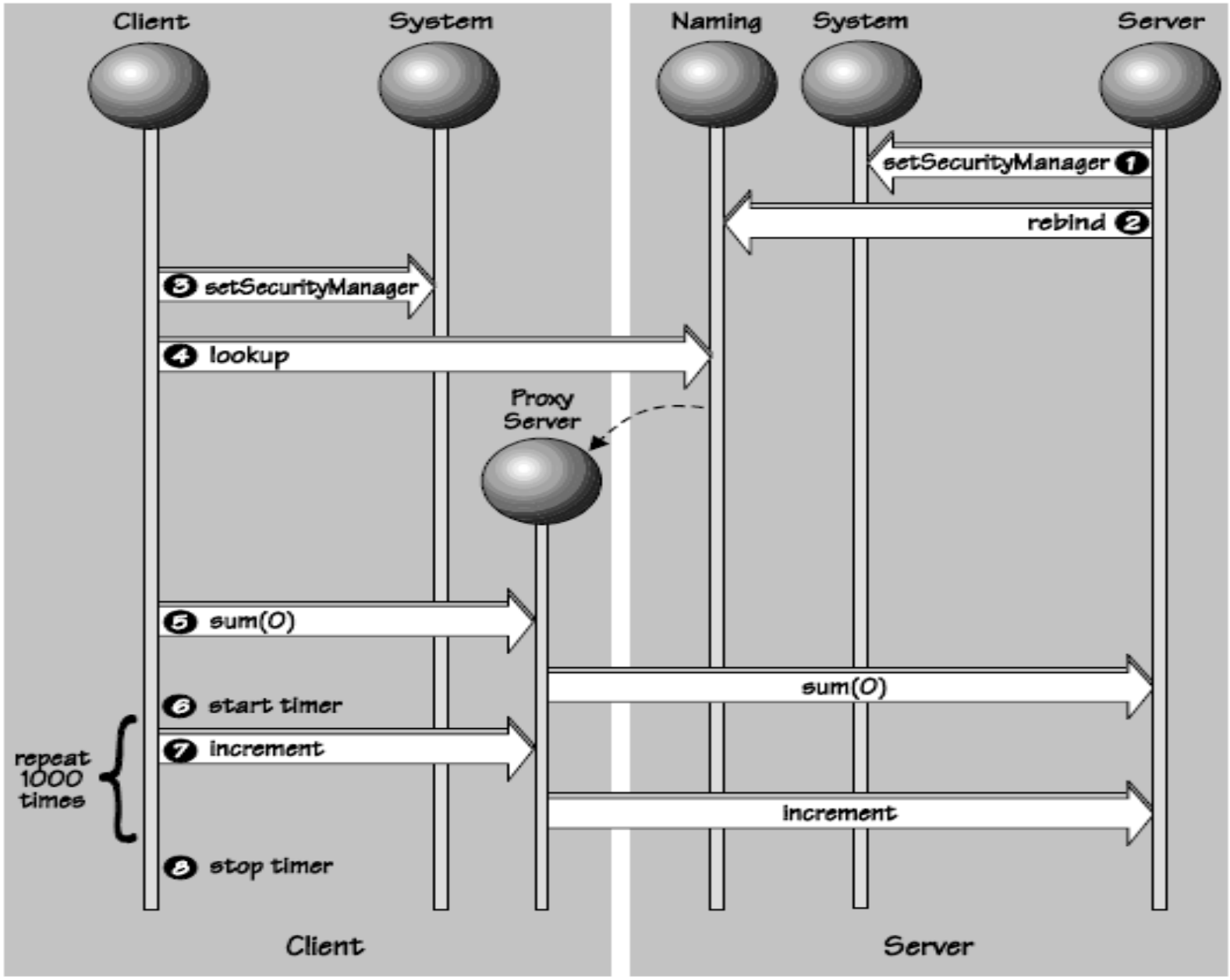
- Stub operation
  - Package identifier of remote object
  - Package method identifier
  - Marshal parameters
  - Send package to server skeleton
- Skeleton Operation
  - _____ parameters
  - Calls method or exception
  - Marshall method return
  - Send package to client stub

How to write an RMI Application



Naming service

- Object registry does this: **rmiregistry**
- Server: Register object(s) with Naming.bind("ObjectName", obj);
- Client: Contact rmiregistry to look up name
- MyInterface test = (MyInterface)Naming.lookup("rmi://www.sjsu.edi/ObjectName");
  rmiregistry returns a remote object reference.
- Lookup gives reference to local stub.
- Invoke remote method(s):test.func(1, 2, "hi");


Client
System
Naming
System
Server
setSecurityManager 1
rebind 2
3 setSecurityManager
4 lookup
Proxy
Server
5 sum(0)
sum(0)
6 start timer
repeat
1000
times
7 increment
increment
8 stop timer
Client
Server

RMI Garbage Collection

- Client JVM sends a dirty call to the server JVM when the object is in use
  - The dirty call is refreshed based on the lease time given by the server
  - Client JVM sends a clean call when there are no more local references to the object
- Unlike DCOM: no incrementing/decrementing of references

Simple RMI Example

- The interface for the Remote Object
  - The interface should extend java.rmi.Remote and all its methods should throw java.rmi.RemoteException

```
/* The RMI server will make a real remote object that
implements this, then register an instance of it with some
URL */
public interface countRMI extends java.rmi.Remote {
  int sum() throws java.rmi.RemoteException;
  void sum (int _val) throws java.rmi.RemoteException;
  public int increment() throws RemoteException;
}
```

```
  public int sum() throws RemoteException
  { return sum;
  }

  public  void sum(int val) throws RemoteException
  { sum = val;
  }

  public int increment() throws RemoteException
  { sum++;
    return sum;
  }
}
```

RMI Client

- Look up the object from the host using Naming.lookup cast it to the appropriate type and use it like local object

```
// CountRMIClient.java  RMI Count client

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class CountRMIClient
{ public static void main(String args[])
  { // Create and install the security manager
    System.setSecurityManager(new RMISecurityManager());

    try
    { CountRMI myCount = (CountRMI)Naming.lookup("rmi://"
                         + args[0] + "/" + "my CountRMI");

      // Set Sum to initial value of 0
      System.out.println("Setting Sum to 0");
      myCount.sum(0);
```

Shark.sjsu.edu

Local call

Remote Objet/Server

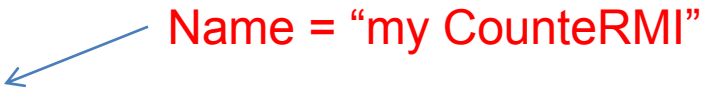
- Remote Object
  - This class must extend UnicastRemoteObject and implement the remote object interface defined earlier
  - The constructor should throw Remote Exception
- The RMI Server
  - The server builds an object and register it with a particular URL
  - Use Naming.rebind (replace any previous binding) or Naming.bind (throw AlreadyBoundException if a previous binding exists)

Remote Object Implementation

```
// CountRMIImpl.java, CountRMI implementation
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class CountRMIImpl extends UnicastRemoteObject
        implements CountRMI
{ private int sum;

  public CountRMIImpl(String name) throws RemoteException
  {
    super();
    try
    { Naming.rebind(name, this);
      sum = 0;
    } catch (Exception e)
    { System.out.println("Exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
```



Compiling/Running

Compile the Client/Server Programs

```
prompt> javac -d \CorbaJavaBook.2e\classes CountRMI.java
prompt> javac -d \CorbaJavaBook.2e\classes CountRMIImpl.java
prompt> javac -d \CorbaJavaBook.2e\classes CountRMIClient.java
prompt> javac -d \CorbaJavaBook.2e\classes CountRMIServer.java

prompt> rmic -d \CorbaJavaBook.2e\classes CountRMIImpl
```

Run the Client/Server Programs

```
prompt> start rmiregistry
prompt> start java CountRMIServer
prompt> java CountRMIClient <server-hostname>
```

If you're running it locally,
use localhost as the hostname.

The Output:

```
Setting Sum to 0
Incrementing
Avg Ping = 3.275 msec
Sum = 1000
```