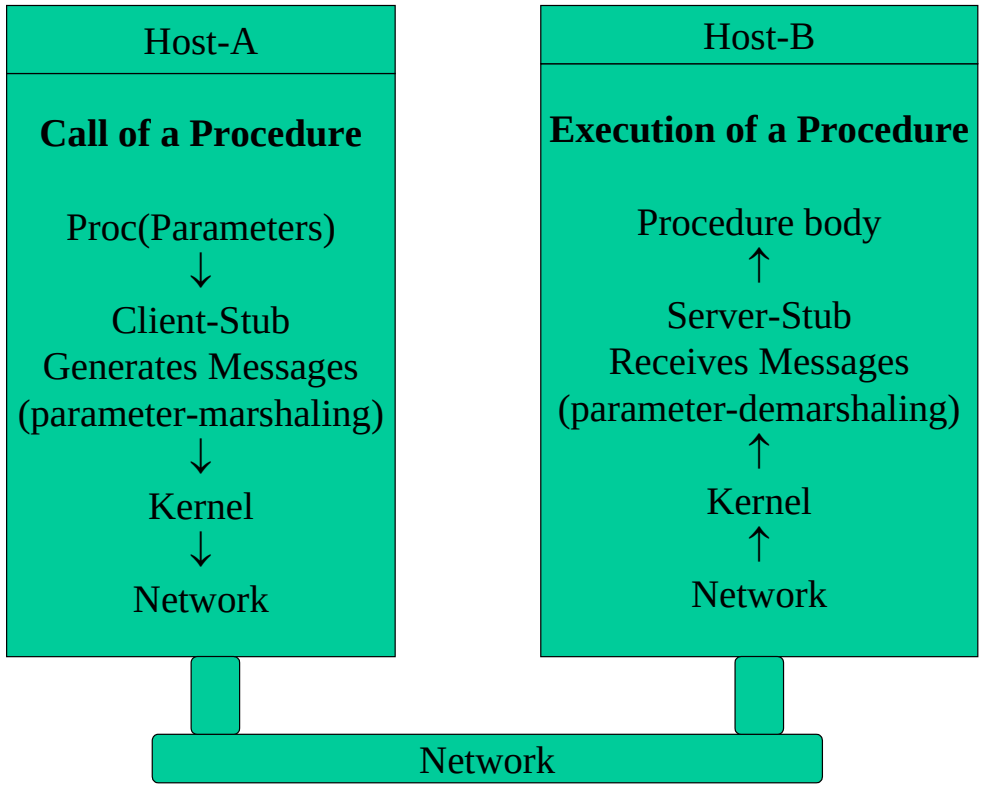
Distributed Systems

2. Remote Object Systems (Java-RMI)

Remote Procedure Call

- A procedure is called and executed in different address spaces. The body is remote to the caller
- Implements response-based transient synchr. communication
- Calling of a remote procedure should be the very same as that of a local procedure.
- Main gain: hides distribution



Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Semantic problems without failure (1)

- Stubs should be generated automatically
  - Error-prone, if programmed manually
- Access transparency (heterogeneous data repr.)
  - Numbering of bits maybe different on diff. CPUs
    - Most significant bit is on the left/right (big/little “endians”)
  - Code systems maybe different
    - ASCII, EBCDIC
  - Representation of numbers maybe different
    - 1- or 2-complement
- Distributed garbage collection is difficult
  - All address spaces must be checked for lack of reference

Semantic problems without failure (2)

- Access to global data is difficult (impossible)
- VAR parameters (call by reference) cannot be realized
- Copy/restore semantics
  - Copy input value at call and overwrite it on return
  - Slightly different semantics

```
VAR  a: INTEGER := 1;
    …
    PROCEDURE Foo (VAR x: INTEGER) =        (*pathologic, but legal example*)
    BEGIN
        x:= 2;
        a:= 0;
    END Foo;
…
Foo (a);

a = 0 if we use call by reference           (*operations on x and a are the same*)
a = 2 if we user copy/restore               (*last value of x overwrites a at return*)
```

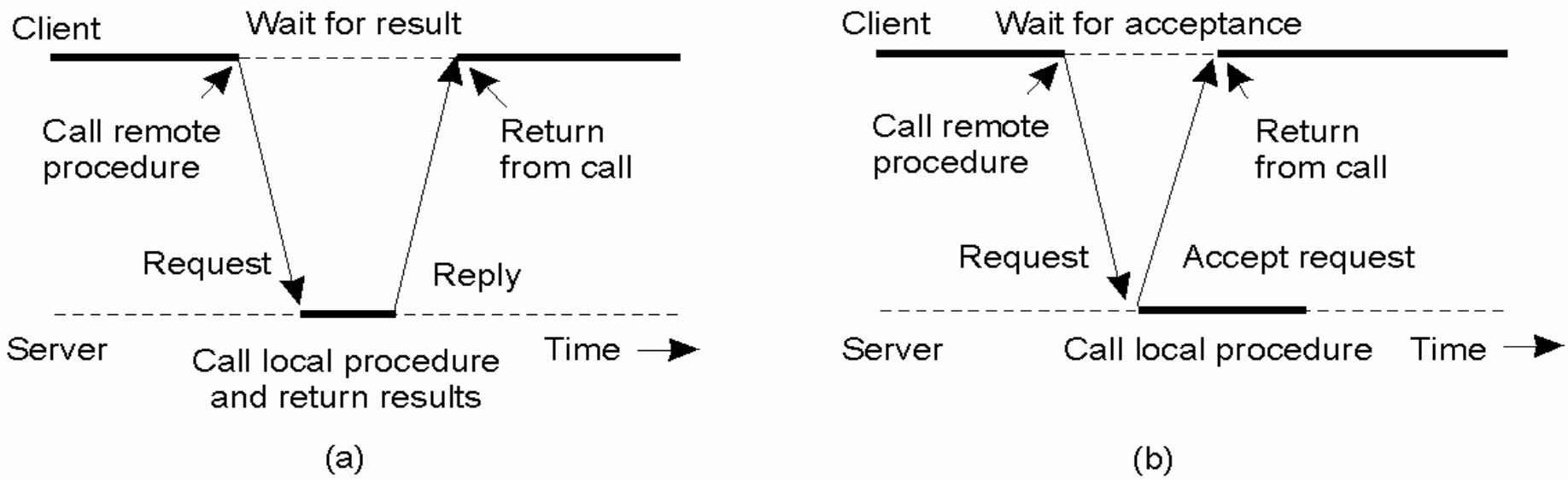
Semantic problems with failure (1)

- *In most cases we do not know, why an answer is missing*
- Server cannot be found
  - Easy, generate exception
- Lost Request Messages
  - Repeat the request after time-out
- Lost Reply Messages
  - Repeating the request is only valid if it is *idempotent* (repeatable)
    - `Read(Block b);` is *not* idempotent: returns *different* block on each call
    - `Read(Block b; int atPosition);` returns always the *same* block
  - We may apply sequence numbers, to filter duplicates
- Server Crashes (before or after the request?)
  - Exactly once semantics – no general solution
  - At least once semantics – for idempotent operations
  - At most once semantics – similar to atomicity

Semantic problems with failure (2)

- Client Crashes
  - The sever works on a job without client: called *orphan*
  - Extermination: Client logs tasks: can identify + kill orphans after reboot – expensive, may not work for chains
  - Reincarnation: Sequentially numbered epochs. At a new epoch (e.g. after reboot) all remote work is killed
  - Gentle reincarnation: At a new epoch servers try to identify their own clients, and kill only real orphans
  - Expiration: Every RPC is given a time quantum *T*. After *T* all remote computation is killed.
- None is really good. Killing an orphan may be disastrous
  - It may keep locks,
  - It might have changed persistent data etc.

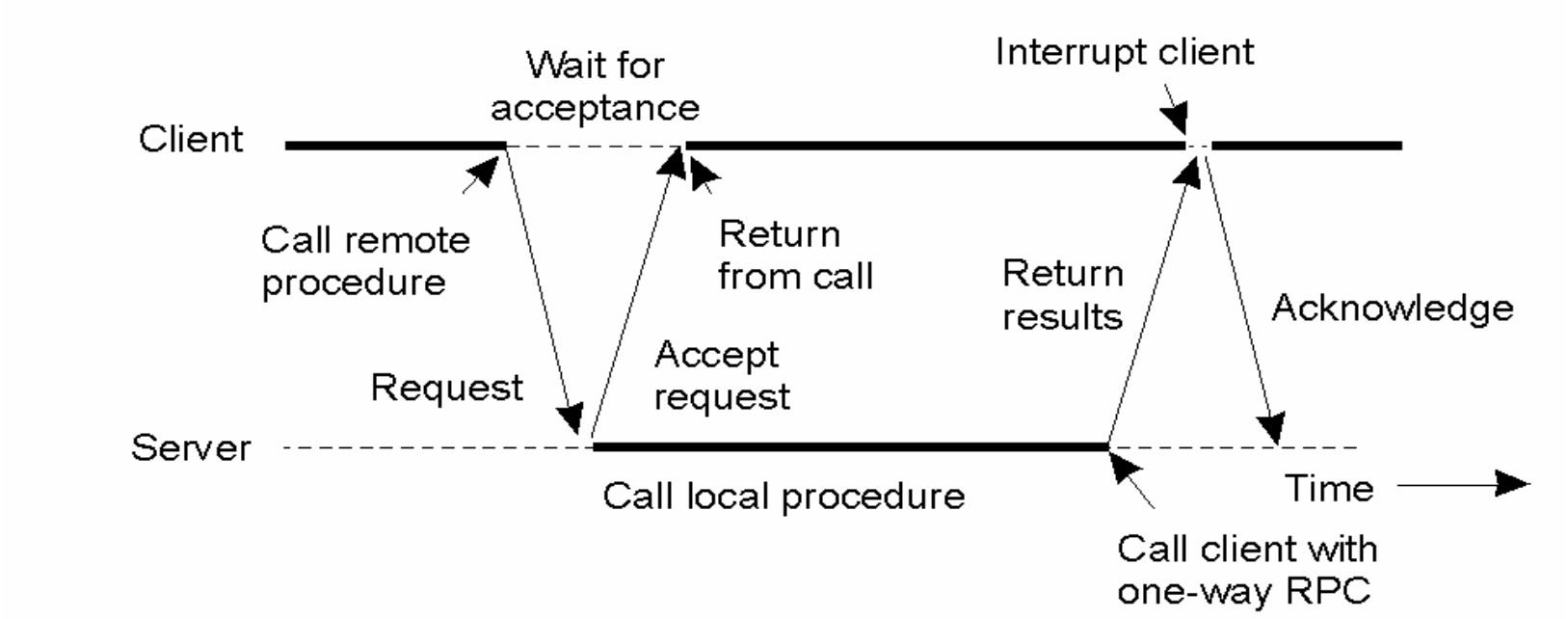
Asynchronous RPC (1)



a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC

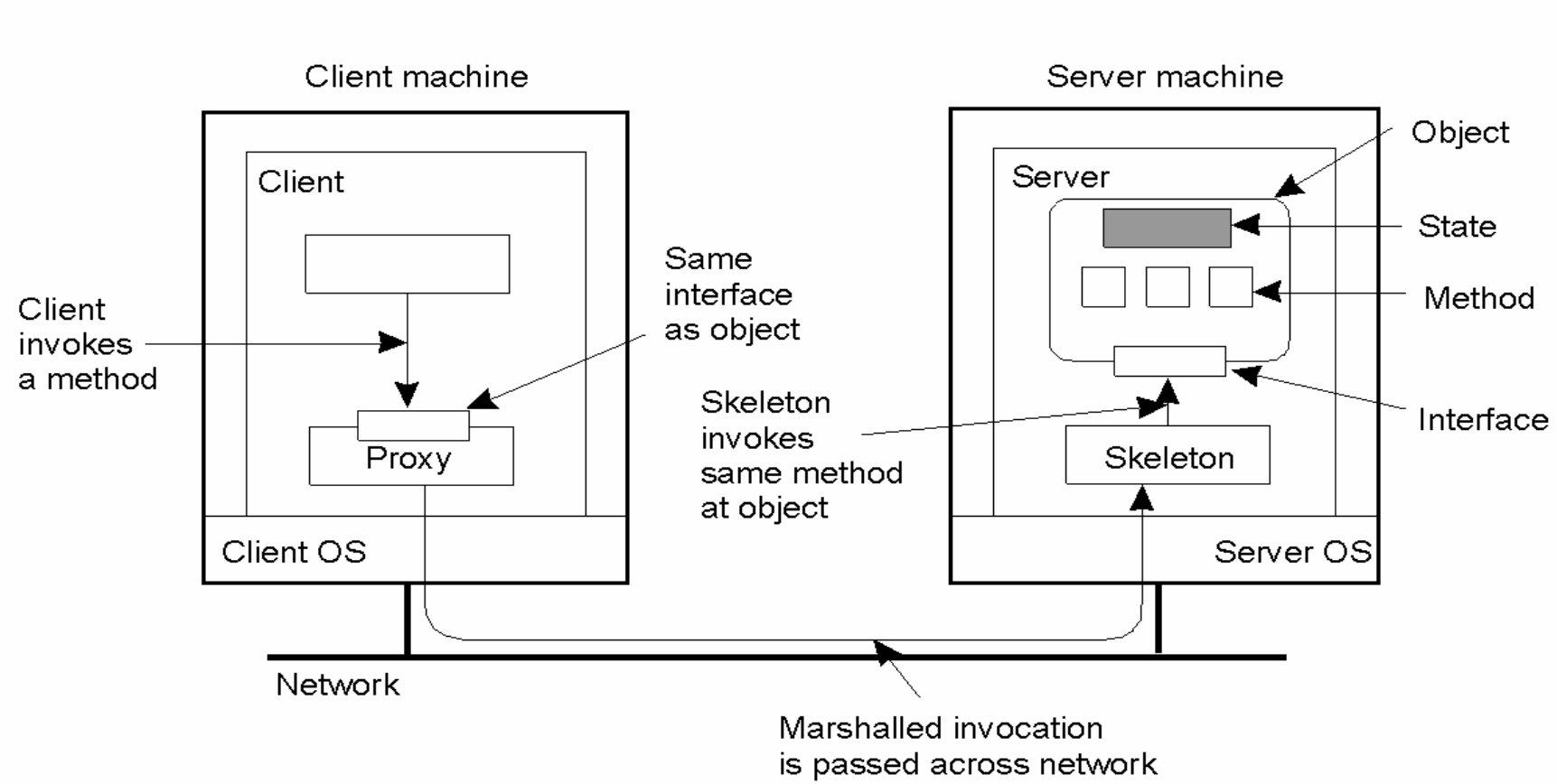
Asynchronous RPC (2)

- A client and server interacting through two asynchronous RPCs



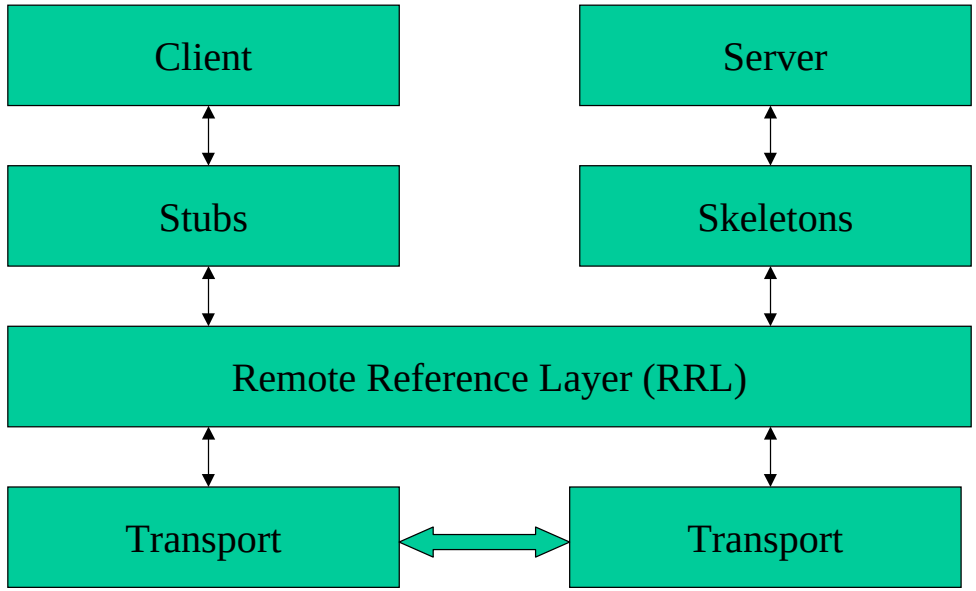
Distributed Objects

- Not only remote methods are provided, but also a *remote (usually hidden) state space* (i.e. instance variables) is maintained



Java RMI (Remote Method Invocation)

- RMI architecture
  - ➢ Innovative is the Remote Reference Layer
- The RRL can be used to implement different semantics, e.g. persistent pointers



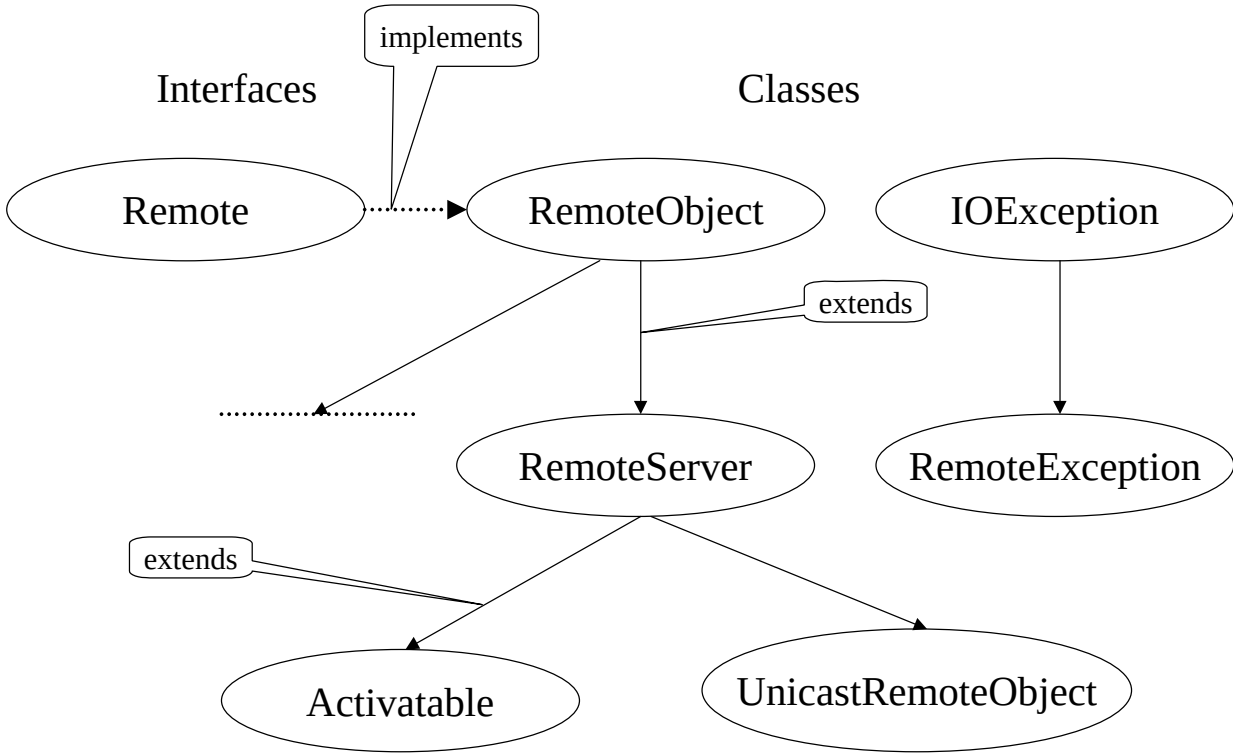
Remote vs. Non-remote objects (1)

- Similarities
  - A reference to a remote object can be passed as an argument or as a result in any method call (local or remote)
  - A remote object can be cast to any of the set of remote interfaces actually supported with the usual syntax
  - The built-in *instanceof* operator can be used as usual

Remote vs. Non-remote objects (2)

- Differences
  - Clients of remote objects interact with remote interfaces, never with implementation classes
  - Non-remote arguments and results are passed to remote objects *by value*, i.e. in the case of an object, a *deep copy* is sent. The object must be *serializable*
  - Remote object arguments and results are passed *by reference*
  - The semantics of some methods of *java.lang.Object* must be specialized:
    - *equals, hashCode, toString*
  - Cloning of remote objects is not possible
  - Exception handling is more complex

RMI Interfaces and Classes (1)



RMI Interfaces and Classes (2)

- A remote interface must extend (directly or indirectly) *java.rmi.Remote*
  - java.rmi.Remote is an empty interface:
    public interface Remote {}
- A remote method declaration must satisfy:
  - It must throw *java.rmi.RemoteException* or a superclass of it (such as *io.IOException* or *lang.Exception*)
  - A remote object declared as a parameter or a return value must be declared as a remote interface
  - A remote interface may extend a non-remote interface, if all methods of the latter satisfy the requirements above
- Use of RMI
  - Create a remote interface
  - Create implementations of the remote interface
  - Create clients of the remote interface

Simple RMI Examples (1)

```
public interface BankAccount extends java.rmi.Remote {

    public void deposit (float amount)
        throws java.rmi.RemoteException;

    public void withdraw (float amount)
        throws Overdrawn, java.rmi.RemoteException;

    public float getBalance ()
        throws java.rmi.RemoteException;

} // BankAccount
```

Simple RMI Examples (2)

```
public interface Alpha {
    public final String const = “constants are o.k. as well”

    public Object x (Object obj)    throws java.rmi.RemoteException;
    public void y ()                throws java.rmi.RemoteException;
} // Alpha

// Beta extends Alpha, which is non-remote, but conform with the remote
    object rules

public interface Beta extends Alpha, java.rmi.Remote {
    public void ping ()             throws java.rmi.RemoteException;
} // Beta
```

Simple RMI Implementation Example

```
public class BankAccountImpl  extends UnicastRemoteObject
                              implements BankAccount {
    private float balance = 0.0;
    public BankAccountImpl(float initalBalance)
                                        throws java.rmi.RemoteException
    {
      balance = initalBalance;
    }
    public void deposit (float amount)   throws java.rmi.RemoteException
      { . . . }
    public void withdraw (float amount)
                              throws Overdrawn, java.rmi.RemoteException
      { . . . }
    public float getBalance ()  throws java.rmi.RemoteException
      { . . . }
} // BankAccountImpl
```

RMI Registration (1)

- Servers must *register* their services by name
  - The registration must be done on the same machine where the service is running
- Clients may *lookup* for services by name on any computer they can reach
  - The client must know where the service is located
  - To find a service on the network an additional *directory service* is needed
- Basic interface
  - *Registry* in the *java.rmi.registry* package

RMI Registration (2)

- The *Naming* class provides a comfortable set of the most important methods for registration
- Naming.bind, Naming.rebind

  Binds a service to a name
  (bind, if not yet exists, rebind does this anyway)

```
TimeServerImpl TSI = new TimeServerImpl();
Naming.rebind("TimeServer", TSI);
```

- Naming.lookup

  Looks for the required service

```
TimeServer TS = null;
try { TS = (TimeServer)
                    Naming.lookup("rmi://lpc1/TimeServer"); . .  . }
```

- Naming.list

  Returns an array of the names of the registered services

```
String [] remObjs =  Naming.list("rmi://lpc1/”);
```

Example Time Server (Interface+Policy)

1. Remote Interface TimeServer

```
import java.util.*;

public interface TimeServer extends java.rmi.Remote {
      public Date getTime ()   throws java.rmi.RemoteException;
} // TimeServer
```

2. Policy file (preferably .java.policy in the home directory)

```
grant {                    // Allow everything for now
        permission java.security.AllPermission;
};
```

Example Time Server (Compilation+Start)

3. Compilation + stub/skeleton generation

   javac *.java
   rmic TimeServerImpl

4. Starting on the server site

   rmiregistry & (except the registry is started by the program)
   java TimeServerImpl &

   (java -Djava.security.policy=java.policy TimeServerImpl &
   if policy file is not stored as .java.policy in the home directory)

5. Starting on the client sites

   java Time lpc5

Example Time Server (Server Impl.-1)

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

public class TimeServerImpl  extends UnicastRemoteObject
                             implements TimeServer {

        public Date getTime()    throws RemoteException  {
            return new java.util.Date();
        } // Date

        public TimeServerImpl() throws RemoteException {
            System.out.println("Initializing Timeserver");
        } // TimeServerImpl
```

Example Time Server (Server Impl.-2)

```
    public static void main(String arg[]) {
            System.setSecurityManager(new RMISecurityManager());
            try {
                TimeServerImpl TSI = new TimeServerImpl();
                Naming.rebind("TimeServer", TSI);
                System.out.println("Registered with registry");
            } catch (RemoteException e) {
                System.out.println("Error: " + e);
            } catch (java.net.MalformedURLException e) {
                System.out.println("URL Error: " + e);
            } // try-catch
    } // main

} // TimeServerImpl
```

Example Time Server (Client Impl.-1)

```
import java.rmi.*;  import java.util.*;
static final String SERVER = "lpc4", ServiceName = "TimeServer";
public class Time {
   public static void main(String arg[]) {
      String ServerName = (argv.length == 1 ? argv[0] : SERVER);
      TimeServer TS = null;                  // The client knows the server site
      try {     TS = (TimeServer) Naming.lookup("rmi://" +
                              ServerName + "/" + ServiceName);
          } catch (NotBoundException e) {
              System.out.println("TimeServer was not found in registry");
              System.exit(0);
          } catch (java.net.MalformedURLException e) {
              System.out.println("URL error: " + e); System.exit(0);
          } catch (RemoteException e) {
              System.out.println("Time error: " + e); System.exit(0);
      } // try
```

Example Time Server (Client Impl.-2)

```
        Date remoteTime = null, localTime = null;
        while (true) {
            try {  remoteTime = TS.getTime();
                     localTime = new java.util.Date();
            } catch (RemoteException e) {
                 System.out.println("Time error: " + e); System.exit(0);
            } // try
            if(remoteTime != null) {
                 System.out.println("remote time: " + remoteTime);
                 System.out.println("local  time: " + localTime);
                 try { Thread.sleep(1000);
                 } catch (java.lang.InterruptedException e) { /* do nothing */ }
            } // if
        } // while
    } // main
} // Time
```

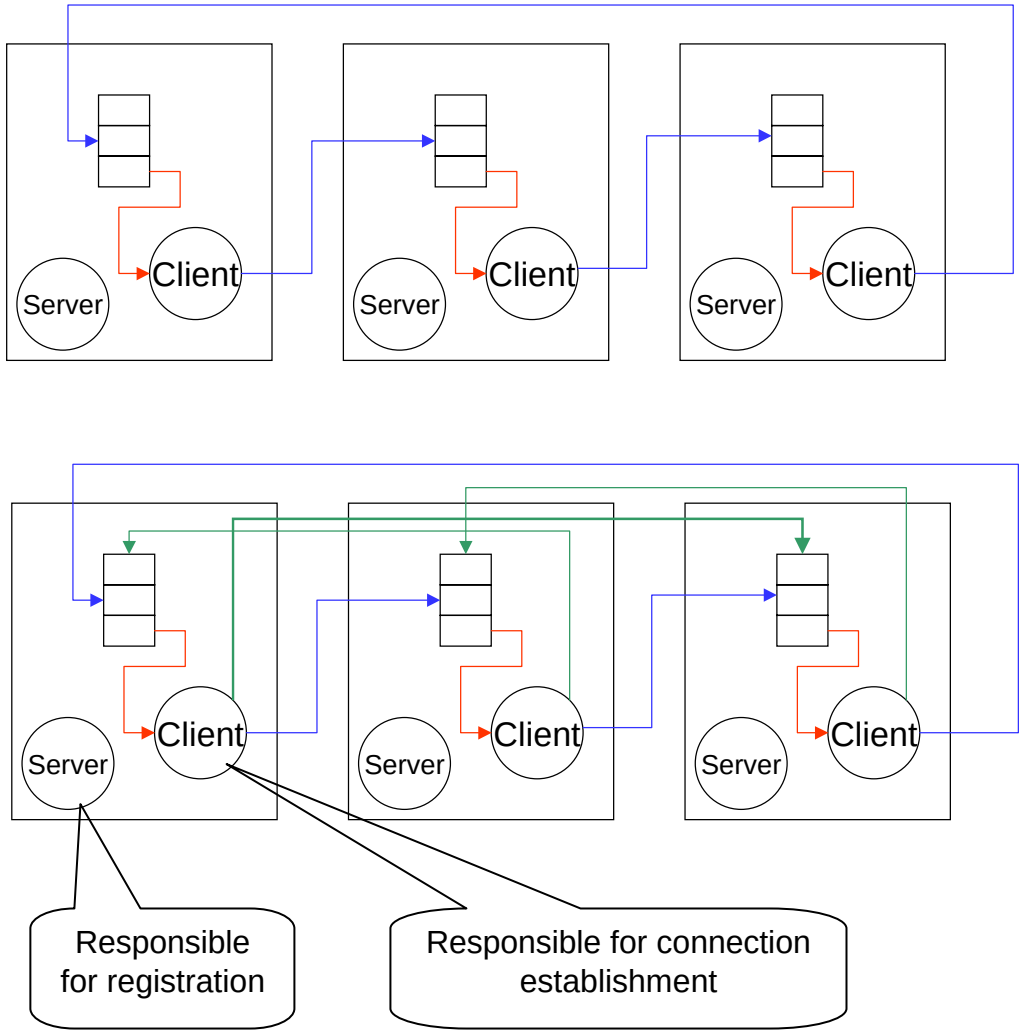
RMI and Firewalls

- RMI provides a means for clients behind firewalls to communicate with remote servers
- Traversing the client's firewall can slow down communication, so RMI uses the fastest successful technique it can detect
- The technique is discovered by the reference for *UnicastRemoteObject* on the first attempt of the client
- Each of three possibilities are tried in turn
  - Communicate directly to the server's port using sockets
  - If this fails, build a URL to the server's host and port and use an HTTP POST request on that URL, sending the information to the skeleton as the body of the POST. If successful, the results of the post are the skeleton's response to the stub
  - If this also fails, build a URL to the server's host using the standard HTTP port (80), using a CGI script that will forward the posted RMI request to the server
- Whichever of these three techniques succeeds first is used for all future communication with the server
- If none of these succeeds, the remote method invocation fails

Message Passing with RMI

- A remote version of a bounded buffer
  - Implementation is almost identical with the *bounded buffer* implementation
  - Except the main method establishing the *registration*
- We install one buffer on each node
- Each node receives messages from its local buffer
  - Client is *consumer* of the local buffer
- Each nodes sends messages to remote buffer(s)
  - Client is *producer* for the remote buffer
- We can build arbitrary connection networks, e.g.
  - A ring: every node sends to its neighbor
  - Double ring: nodes send their direct and second neighbor

Message Passing Networks



Message Passing Interface

```
public interface MessagePassing extends java.rmi.Remote {
        public static final String ServiceName = "MessagePassing";
        public static final int DefaultPort = 1099;

        public int capacity () throws java.rmi.RemoteException;
        // invariant: 0 < capacity

        public int count ()  throws java.rmi.RemoteException;
        // invariant: 0 <= count <= capacity

        public void   send (Object x) throws java.rmi.RemoteException;
        // add only when count < capacity

        public Object receive () throws java.rmi.RemoteException;
        // remove only when count > 0
}// MessagePassing
```

Message Passing Server (1)

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

public class MessagePassingImpl        extends UnicastRemoteObject
                                       implements MessagePassing {
    protected Object[]  array_;                    // content
    protected int putPtr_ = 0, getPtr_ = 0,        // circular indices
    usedSlots_ = 0;                                // num. of elems
    public MessagePassingImpl(int capacity) throws RemoteException,
                                                   IllegalArgumentException {
        if (capacity <= 0) throw new IllegalArgumentException();
        array_ = new Object[capacity];
    }
    public int count()             {return usedSlots_;}      // Sync?
    public int capacity()          {return array_.length;}   // Sync?
```

Message Passing Server (2)

```
public synchronized void send (Object x) throws RemoteException
        while (usedSlots_ == array_.length)         // wait until not full
            try { wait(); } catch(InterruptedException ex) {};
        array_[putPtr_] = x;                        // At least 1 free place exists
        putPtr_ = (putPtr_ + 1) % array_.length;    // cyclically increment
        if (usedSlots_++ == 0) notifyAll();         // If previously empty: notify
} // send
public synchronized Object receive () throws RemoteException
        while (usedSlots_ == 0)                     // wait until not empty
            try { wait(); } catch(InterruptedException ex) {};
        Object x = array_[getPtr_];                 // At least 1 element exists
        array_[getPtr_] = null;
        getPtr_ = (getPtr_ + 1) % array_.length;    // cyclically increment
        if (usedSlots_-- == array_.length) notifyAll();  // If previously full: notify
        return x;
} // receive
```

Message Passing Server (3)

```
    public static void main(String arg[]) { // Establish new, or locate old registration
        Registry reg = null;
        System.setSecurityManager(new RMISecurityManager());
        try { reg = LocateRegistry.createRegistry(DefaultPort);}  // Create new registry
        catch (RemoteException e) {
            try { reg = LocateRegistry.getRegistry(); }  // Locate old registry if available
            catch (RemoteException e2) {
                System.out.println("Registry cannot.."+e); System.exit(0);
        }   } // try-catch-e2 , try-catch-e
        System.out.println("Registry established");
        try  { MessagePassingImpl mp = new MessagePassingImpl();
            reg.rebind(ServiceName, mp);              // Bind service to service name
        } catch (RemoteException e) {
            System.out.println("Binding cannot be established "+ e);
            System.exit(0);
        }  // try-catch
        System.out.println("Registration is successful");
}      } // main,  MessagePassingImpl
```

Message Passing Producer Thread

```
class Producer extends Thread  {
    private MessagePassing buf = null;
    private String target; private boolean stopped = false;
    Producer (MessagePassing sendBuf, String targetName)  {
        buf = sendBuf; target = targetName; }
    public void stopp () {stopped = true;}
    public void run ()  {     // produces and sends time stamps in a loop
        while (!stopped)  {
            Date nextTS = new java.util.Date();        // Sets e.g. a time stamp
            System.out.println("sends= " +  nextTS.toString());
            try { buf.send(nextTS); }
            catch (java.rmi.RemoteException e)
                  {System.out.println("send-error" + e); }
            try { Thread.sleep(1000);}
            catch (java.lang.InterruptedException e) { }
}  }    } // while, run, Producer
```

Message Passing Consumer Thread

```
class Consumer extends Thread  {
    private MessagePassing buf = null;
    private boolean stopped = false;
    Consumer (MessagePassing recBuf)  {buf = recBuf;}
    public void stopp () {stopped = true;}

    public void run ()  {          // receives time stamps in a loop
        while (!stopped)  {
            Date receivedTS = null;
            try  { receivedTS = (Date) buf.receive();
                System.out.println("rcvd= "+receivedTS.toString());
            }  catch (java.rmi.RemoteException e)
                {System.out.println("receive-error" + e); }
            try { Thread.sleep(1000);} catch (java.lang.InterruptedException e) { }
}  }    } // while, run, Consumer
```

Message Passing Help Class

```
import java.net.InetAddress;
public class MyHost { // hostname 3 chars, hostnum 1 digit; as lpc1-lpc9

    public static int NameToNum (String name) {
        return Integer.parseInt(name.substring(3,4)) ;
    } // NameToNum

   public static String NumToName (int num)  {
        return Name().substring(0, 3) + num;
   } // NumToName

    public static String Name () {
        try {
            InetAddress myAddress = InetAddress.getLocalHost();
            return  myAddress.getHostName().substring(0, 4);
        } catch (java.net.UnknownHostException e) {
            return "0000";
}   }    } // try, Name, MyHost
```

Message Passing Client (1)

```
public class Client {
    public static final String ServiceName = "MessagePassing"; public static final int Nodes = 3;
    public static MessagePassing Connect(String processorName) {
        boolean notBound = true; MessagePassing mps = null;
        String target = "rmi://" + processorName + "/" + ServiceName;
        while (notBound)  {
          try { mps = (MessagePassing) Naming.lookup(target);
                notBound = false;
          } catch (NotBoundException e) {
                System.out.println("Trying to connect " + target);
                try { Thread.sleep(100000);} catch (InterruptedException e2) {}
          } catch (RemoteException e) {
                System.out.println("Trying to connect " + target);
                try { Thread.sleep(100000); } catch (InterruptedException e2) { }
          } catch (java.net.MalformedURLException e) {
                System.out.println("URL error: " + e);  System.exit(0);
        }  } // try, while
        System.out.println("Connection established to " + target);
        return mps;
    } // Connect
```

Message Passing Client (2)

```
public static String NextName (String name)  {
      return MyHost.NumToName((MyHost.NameToNum(name) % Nodes) + 1);
} // NextName
public static void main  (String arg[]) {              // Establishes double-ring connection
      String myName = MyHost.Name();
      String next = NextName(myName), nextNext = NextName(next);
      System.out.println("Process on " + myName + " starts");
      MessagePassing recBuf = Connect (myName) ;
      MessagePassing sendNext1 = Connect (next), sendNext2 = Connect (nextNext);
      Producer prod1 = new Producer(sendNext1, next);          // sends to neighbour
      Producer prod2 = new Producer(sendNext2, nextNext);    // sends to next neighbour
      Consumer cons = new Consumer(recBuf);                    // receices from own buffer
      Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
      prod1.start(); prod2.start(); cons.start();
      int duration = 10000; if (arg.length > 0) duration = Integer.parseInt(arg[0]);
      try { Thread.sleep(duration); }
      catch (java.lang.InterruptedException e) { prod1.stopp(); prod2.stopp(); cons.stopp();
      } // try-catch
      System.out.println("Process on " myName + " finishes"); System.exit(0);
} } // main, Client
```