Imperial College London

Computer Networks and Distributed Systems

Part 2.2 – Interaction Primitives

Course 527 – Spring Term 2017-2018

Emil C Lupu

e.c.lupu@imperial.ac.uk

Contents

Introduction

Components of a distributed system communicate in order to cooperate and synchronise their actions.

A distributed system makes use of *message passing* as a basic mechanism supported by the communication system as there is no shared memory.

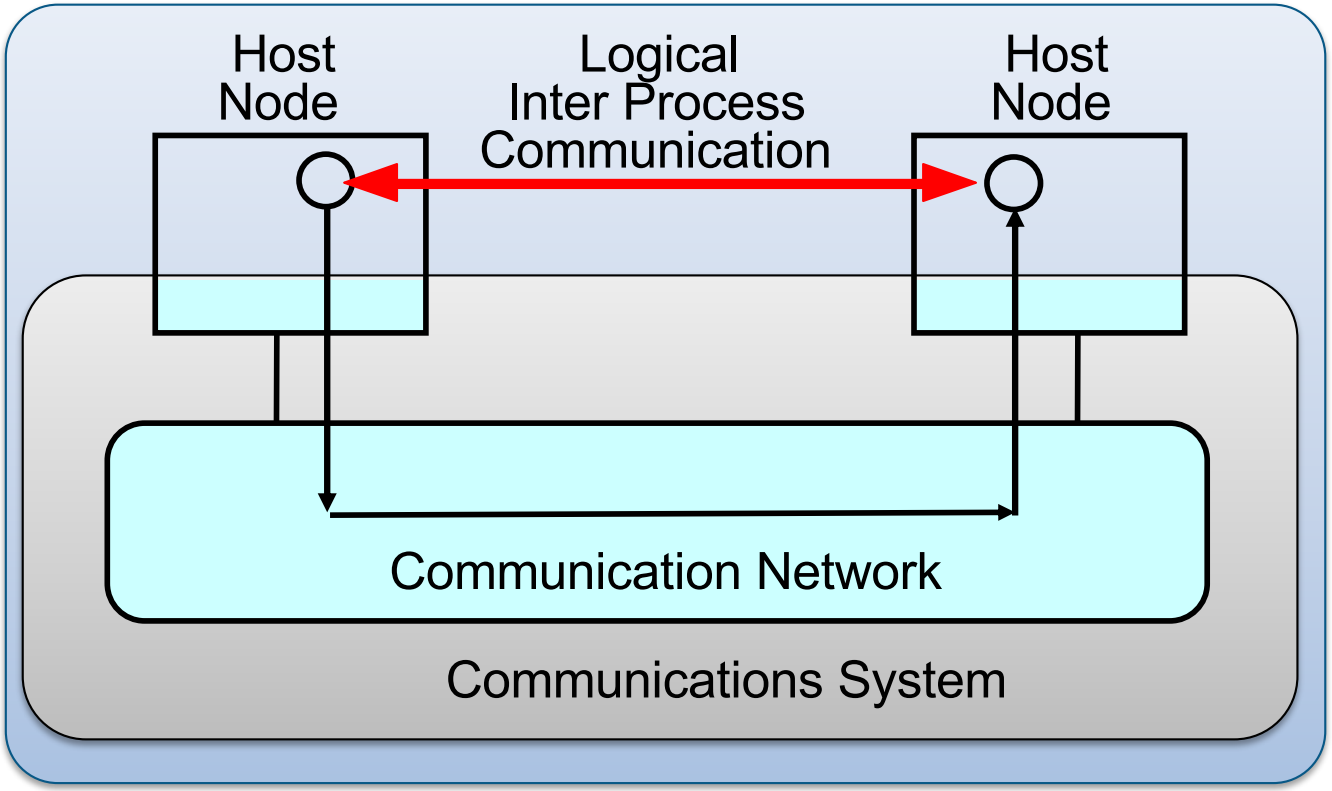
OS primitives -> low level + untyped data

Language primitives -> higher level + typed data

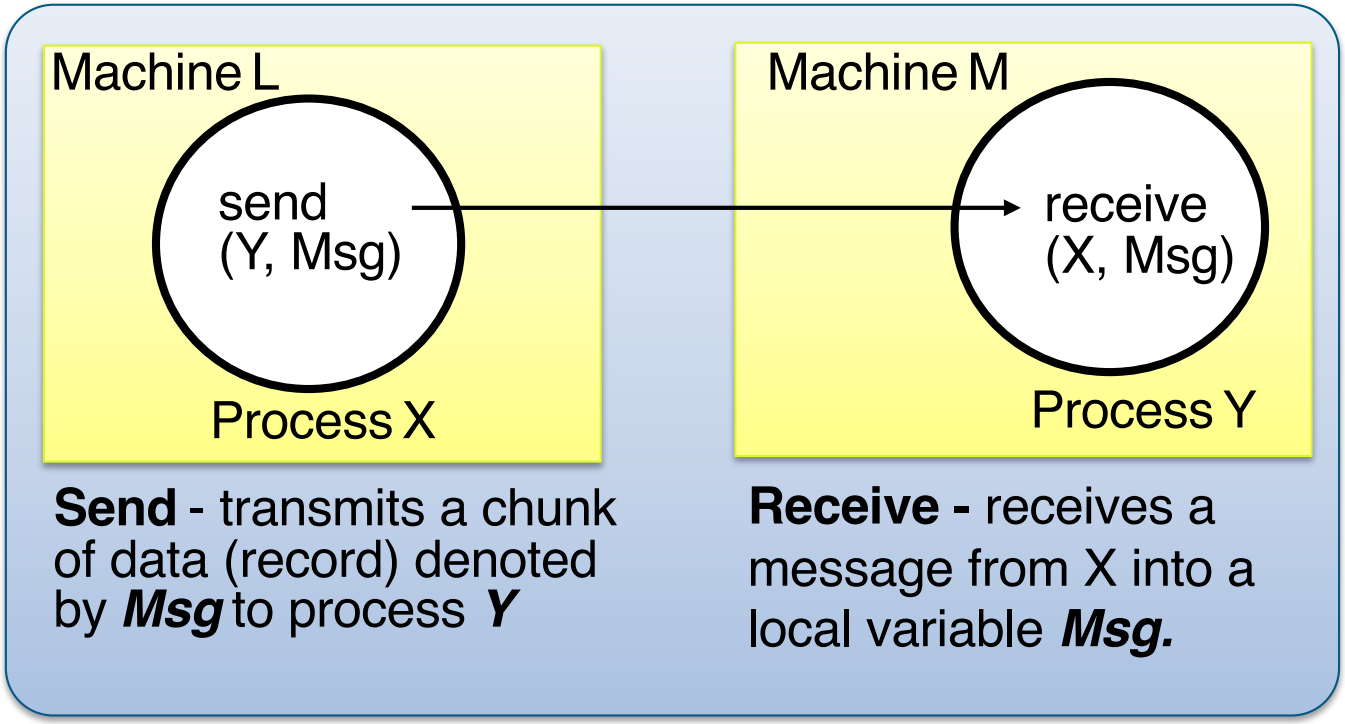
Shared memory abstractions can be implemented above message passing

Basic Message Passing
Inter Process Communication

Inter Process Communication (IPC)



Message Passing

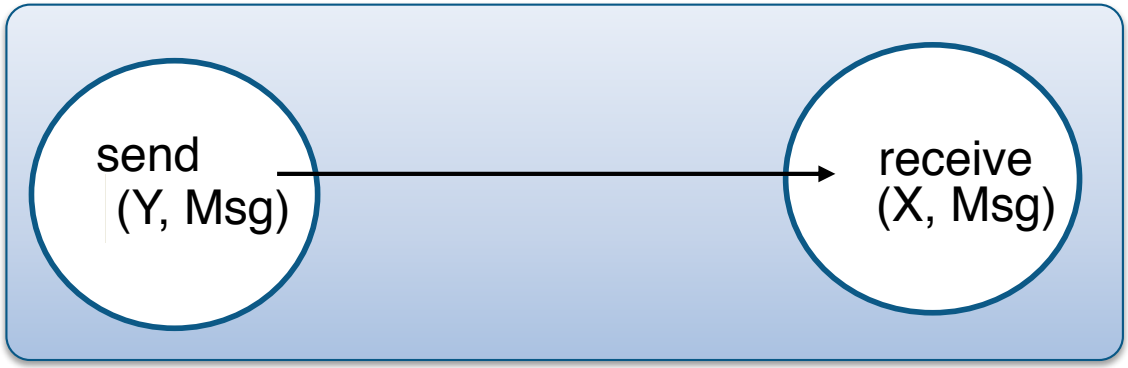


Both sender and receiver directly name each other in their programs

- What are the disadvantages?
- What are the alternatives?

Asynchronous Send

Asynchronous send is an **unblocked send**, where the sender continues processing once the message has been copied out of its address space.



Characteristics:

- Mostly used with blocking receive
- Underlying system must provide buffering (usually at the receive end)
- Loose coupling between sender and receiver(s)
- Readily usable for multicast. Efficient implementation

Problems

- Buffer exhaustion (no flow control). What should happen if the destination runs out of buffers?
- Error reporting of lost messages is not sensible. Difficult for sender to match error report with affected message.
- Formal verification is more difficult, as need to account for the state of the buffers.

Maps closely onto connectionless communication service but can be implemented via a reliable connection service.

How can 2 processes use async send and blocking-receive, synchronise to perform an action at the same time?

Synchronous Send

Synchronous send is a **blocked send**, where the sender is held up until actual receipt of the message by the destination. It provides a synchronisation point for the sender and receiver (cf. handshaking).

CSP: `clock! start`

Characteristics:

- Tight coupling between sender and receiver (tends to restrict parallelism)
- For looser coupling, use explicit buffer processes
- Generally easier to reason about synchronous systems

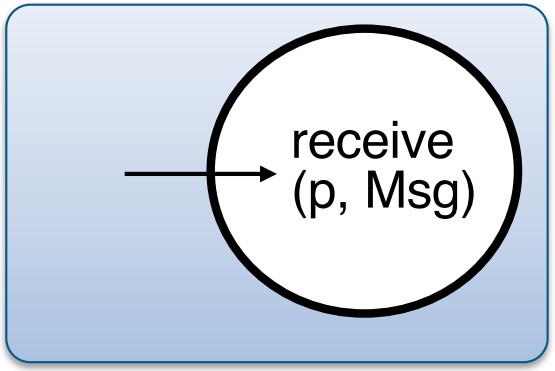
Problems:

- Failures and indefinite delays -> indefinitely blocked ?
  -> create a thread to delegate the send responsibility
- No multidestination
- Implementation more complex (especially if include timeout)
- The underlying communication service is expected to be **reliable**.

Blocked Receive Primitive

Blocked receive - The destination process blocks if no message is available, and receives it into a target variable when it is available.

```
Ada   ACCEPT p ( mess: IN ....)
CSP   source?msg { direct naming }
Unix  size = recvfrom( socket, buffer,
                       from)
```



Variations:

- Timeout – specify a limit on the amount of time prepared to wait for a message, otherwise take an alternative action.
- Conditional receipt – the receive returns an indication of whether a message was received (e.g. true or false) and the process continues. A non-blocking receive lets a process check if a message is waiting from different clients (cf. Polling)

Unix Socket Interface

Internet Addresses

- Messages in the Internet (IP packets) are sent to addresses which consist of three components: (Netid, Hostid, Portid)
- The network identifier and host identifier take up 32 bits divided up according to the class of the address (A,B,C). The port identifier is a 16 bit number significant only within a single host. (By convention port numbers 0..511 are reserved and are used for well known services e.g. telnet, ftp, …).

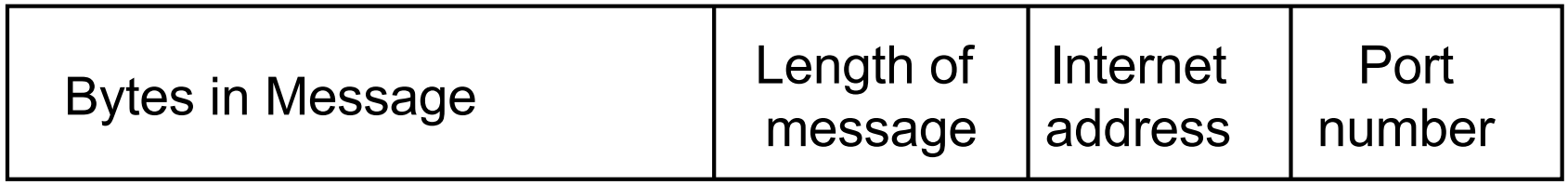
Sockets

- Sockets are the programming abstraction provided by Unix to give access to transport protocols or for local IPC. A socket provides a file descriptor at each end of the communication link.

Java API for UDP Datagrams

Two Java classes:

- `DatagramPacket` provides a constructor to make a UDP packet from an array of bytes



- Another constructor is used when receiving a message. Methods `getData, getPort, getAddress` can be used to retrieve fields of `DatagramPacket`
- DatagramSocket. Methods `send` and `receive` for transmitting `datagramPacket`. `setSoTimeout` for a receive to limit how long it blocks. If the timeout expires it throws an `interruptedIOException`.

http://docs.oracle.com/javase/tutorial/networking/sockets/index.html
http://docs.oracle.com/javase/tutorial/networking/datagrams/

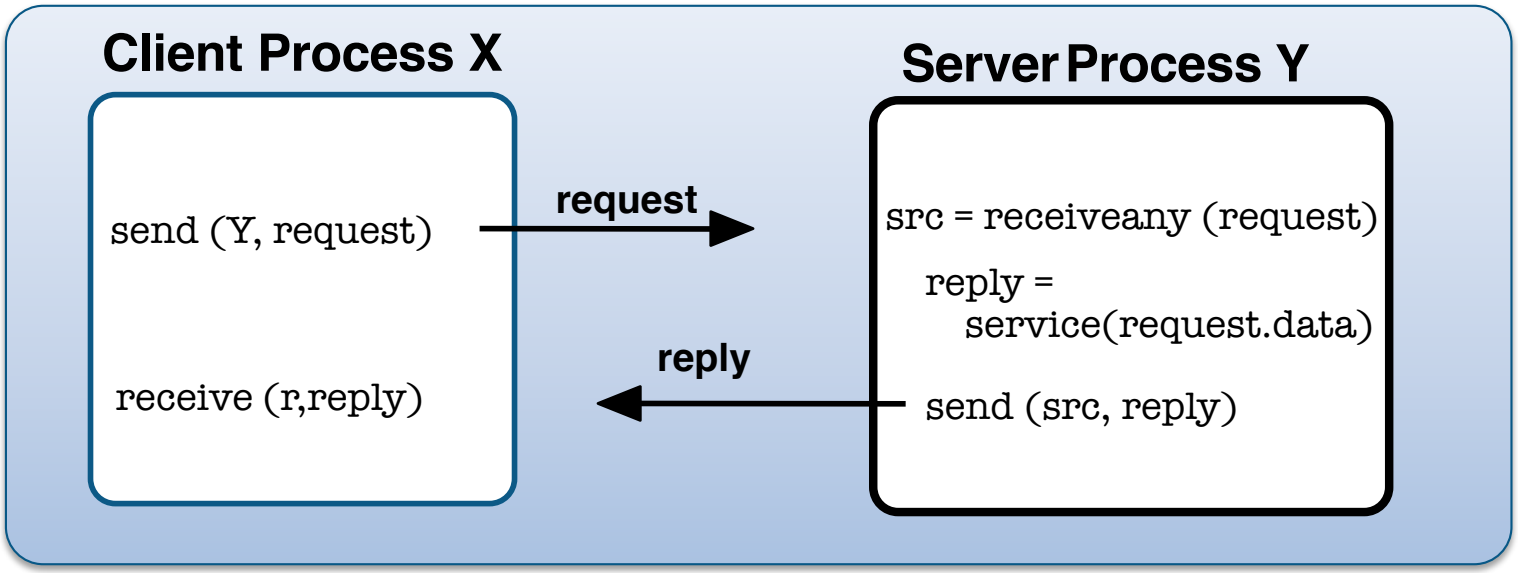
UDP Client

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[ ]){      // args give: msg. contents, destination
      DatagramSocket aSocket = null;
      try { aSocket = new DatagramSocket();  // creates socket on any available port no.
        byte [ ] m = args[0].getBytes();
        InetAddress aHost = InetAddress.getByName(args[1]); //convert name to IP addr.
        int serverPort = 6789;
        DatagramPacket request =
                        new DatagramPacket(m,  args[0].length(), aHost, serverPort);
        aSocket.send(request);
        byte[ ] buffer = new byte[1000];
        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
        aSocket.receive(reply);
        System.out.println("Reply: " + new String(reply.getData()));
     }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
     }catch (IOException e){System.out.println("IO: " + e.getMessage());
} }
```

UDP Server

```
import java.net.*;
import java.io.*;
public class UDPServer {  // echoes received data
   public static void main(String args[ ]){
     DatagramSocket aSocket = null;
     try{
          aSocket = new DatagramSocket(6789);   // create socket at agreed port
          byte[ ] buffer = new byte[1000];
          while(true){
               DatagramPacket request = new DatagramPacket(buffer, buffer.length);
               aSocket.receive(request);     // now send the packet back
               DatagramPacket reply = new DatagramPacket(request.getData(),
               request.getLength(), request.getAddress(), request.getPort());
               aSocket.send(reply);
          }
     }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
     }catch (IOException e) {System.out.println("IO: " + e.getMessage());
   } }
```

Client Server Interactions



- Request-reply is used to implement client - server communication

- The Process Y address must be known to all client processes. This is usually achieved by publishing it in a nameserver.

Message Passing Recap

OS level message interaction primitives e.g., sockets are considered as the assembly language level of distributed systems. They are too low-level an abstraction to be productively used by programmers.

- Message passing systems do not deal with the problems of marshalling a set of data values into a single contiguous chunk of memory which can be sent as a message.
- Data types are represented in memory in different ways by different machines and by different compilers. Message passing does not address this heterogeneity.
- Programming paradigms such as client-server is cumbersome.
- Component interface is not explicit – many different types of messages may be received by a process into a single message variable

However

- Asynchronous message passing permits parallelism
- Message passing is the fundamental means of interacting in distributed systems and is used to implement higher level primitives such as Remote Procedure Call or Object invocation

Remote Procedure Calls (RPC)

Remote Procedure Call (RPC)

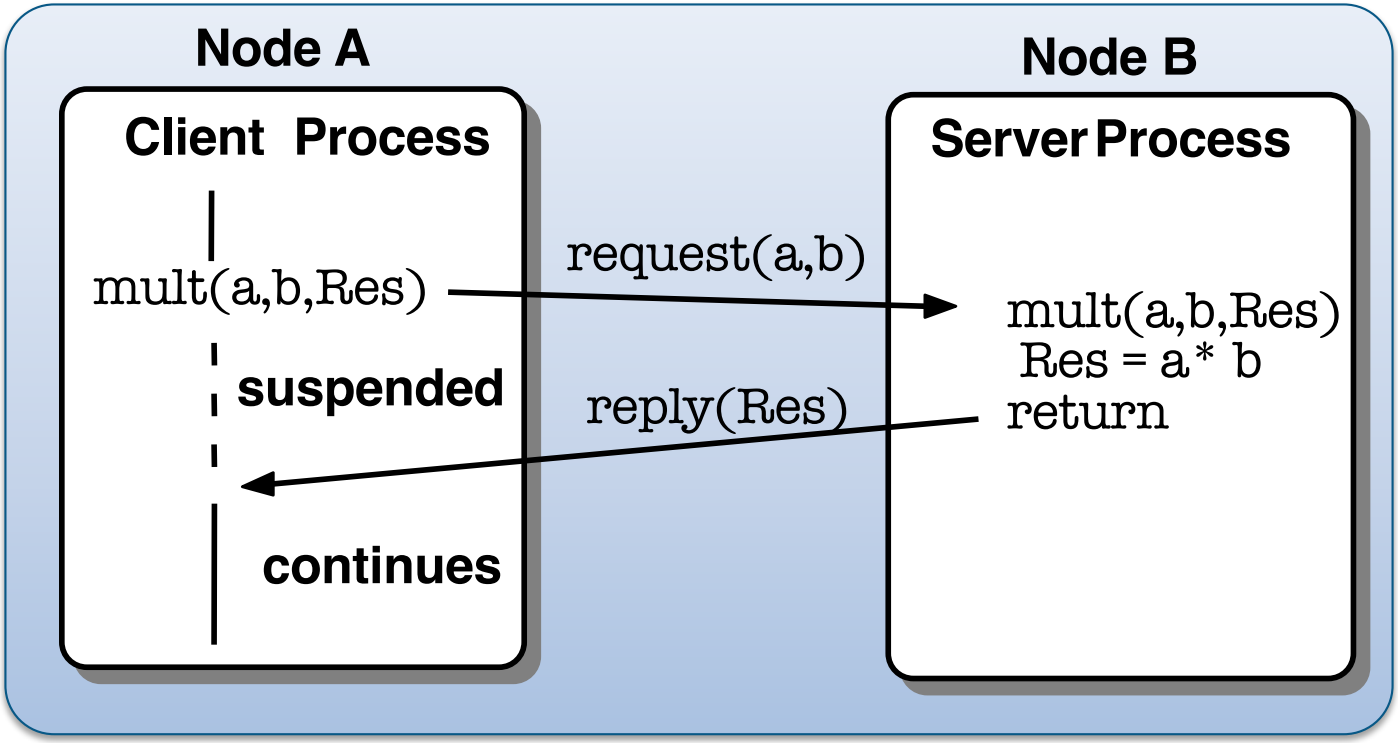
Basic message passing leaves a lot of work for the programmer e.g. constructing messages, transforming data types

Remote Procedure Calls aim to make a call to a remote service look the same as a call to a local procedure.

- The parameters to the call are carried in one or several request messages and the results returned reply message(s).
- However calls to procedures implemented remotely can fail in different ways to procedures implemented locally, the semantics of an RPC are different from those of a local procedure.

Birrel A.D. and Nelson B.J. (1984). Implementing remote procedure calls. ACM Transactions on Computer Systems, vol. 2, pp. 39-59.

RPC Interactions



- Client is suspended until the call completes.
- Parameters must be passed **by value** since Client and Server processes do not share memory and thus Client pointers have no meaning in Server address space.
- We need code to encode the parameters and method name in messages, and to decode them when receiving.

Stub Procedures

Client has a local stub for every remote procedure it can call.

Server has local stub (skeleton) for every procedure which can be called by a remote client.

Methods called on stub. Stub procedures perform:

- Parameter marshalling (packing) - assemble parameters in communication system messages
- Unpacking received messages and assigning values to parameters.
- Transform data representations if necessary
- Access communication primitives to send/receive messages.

Stubs can be generated automatically from an interface specification.

Interface Definition Language (IDL)

IDL is a data typing language used to specify the interface operations and their parameters.

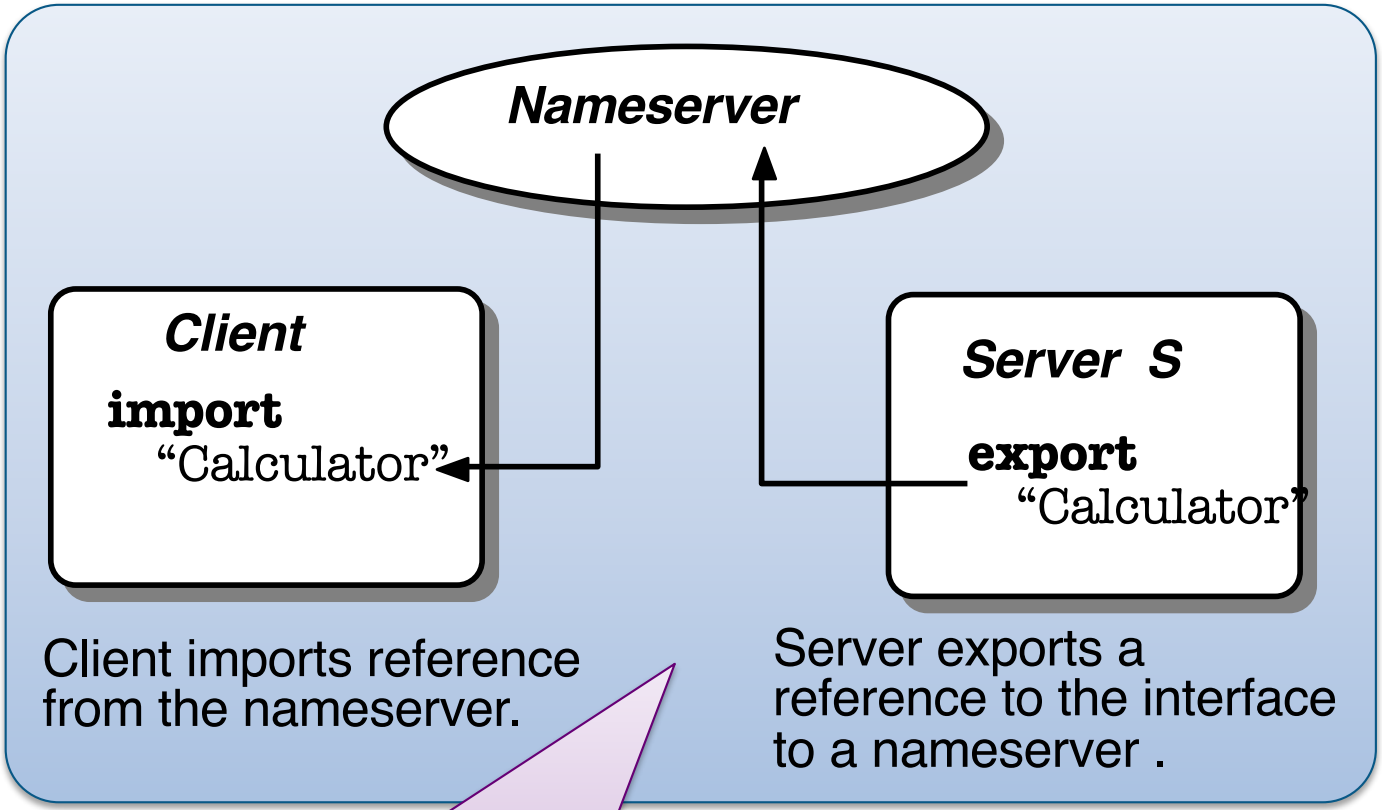
- Pseudo code example

```
interface calc  {
        void mult ( [in]  float a, [in]  float b, [out] float Res );
        void square ( [in]  float a, [out] float Res );
}
```

- The IDL compiler uses this interface definition to generate the code for both client and server stubs in calc.idl.
- The above assumes multiple in and out parameters but Sun RPC is more primitive and allows only a single input parameter and a single result although these may be complex data structures.

Binding

Binding maps an RPC interface used by a client to the implementation of that interface provided by a server.



Calculator Client Code

```
#include “calc.idl”

void main()  {
    status = import  (calc c, “calculator”, nameserverAddress);
    c.mult(1.2, 5.6, Res);
    print (Res);
}
```

Lookup calculator in nameserver and bind to c

Use remote procedure reference c to invoke mult operation

Calculator Server Code

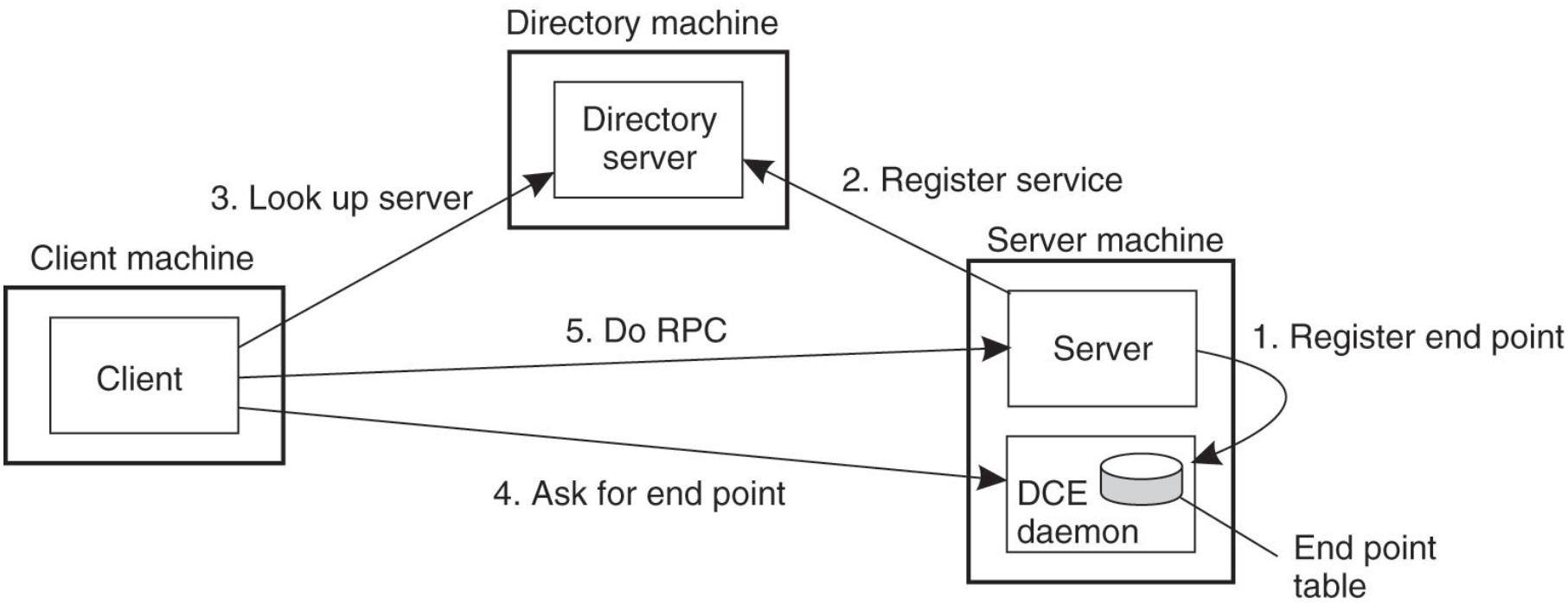
```
#include "calc.idl"

void main() {
   status = Export  (calc, "calculator", nameserverAddress);
   status = RpcServerListen ();
   //tells runtime system that server is ready to receive calls.
}

// implementation for interface procedures
void  mult (float a, float b, float  Res) {
   Res = a * b;
}

void square (float a, float Res) {
   Res = a * a;
}
```

Example: DCE RPC Overview



Server location is done in two steps:

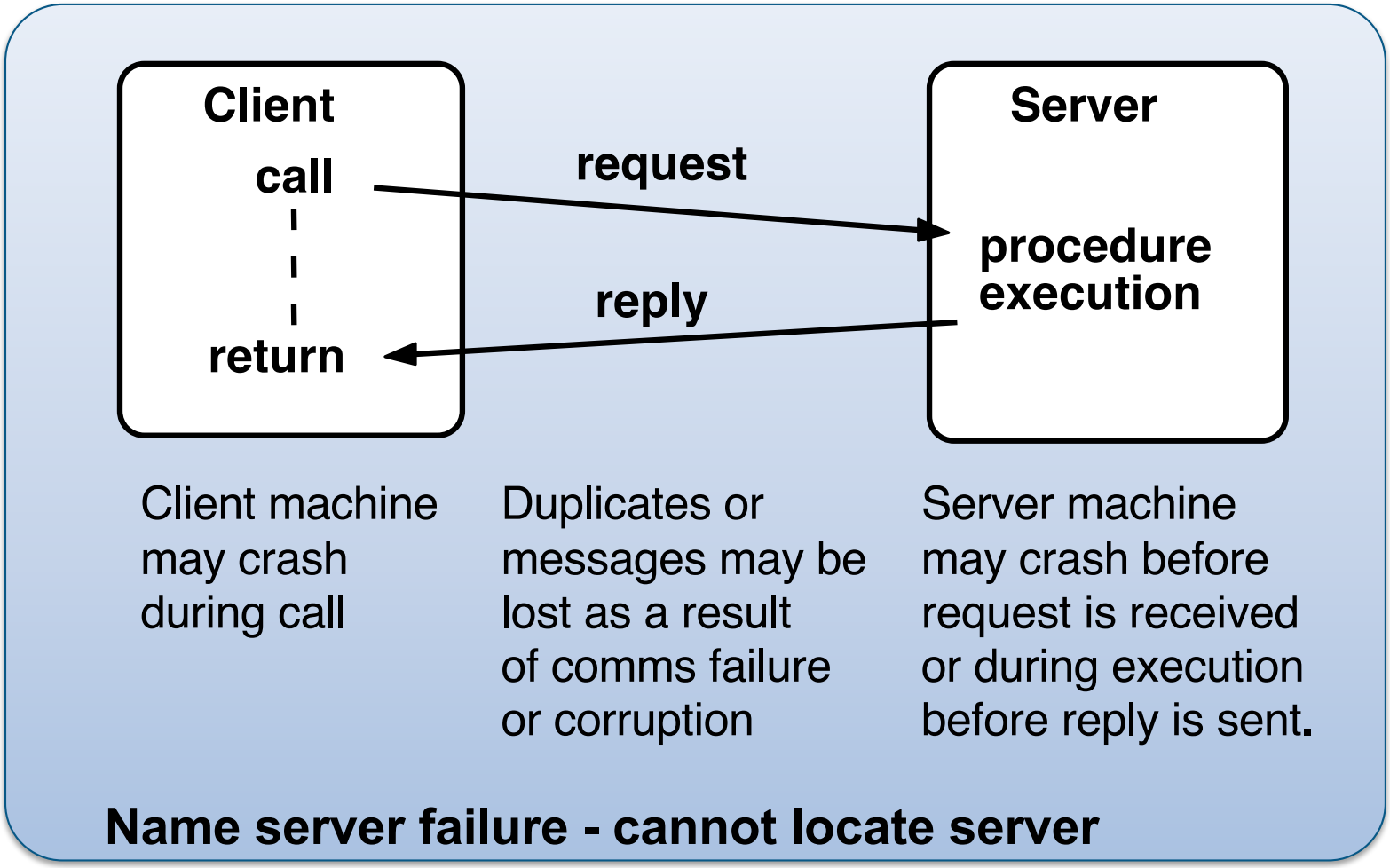
1. Locate the server's machine.
2. Locate the server on that machine.

RPC Election Service

Specify an RPC interface to an Election Service which allows a client to both query the current number of votes for a specified candidate and vote for one of the set of candidates. Each client has a voter number used for identification in requests and candidates are identified by a string name.

RPC Failures

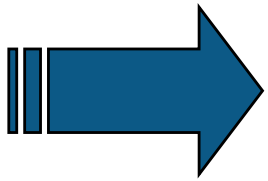
Remote procedure calls differ from local procedure calls in the ways that they can fail.



A server may also fail and quickly recover in-between client calls without the client knowing, but this may result in loss of state information pertaining to the client interaction - so the client needs to know about server epochs.

Orphan executions – result from a client crashing while the server is executing the procedure. For long running procedures, to avoid wasting resources, the server may wish to be informed of client crashes so that it can abort orphan executions.

A number of different **call semantics** are possible depending on the fault-tolerance measures taken to overcome these failures.



Maybe (Best-Efforts) Call Semantics

```
bool call (request, reply) {
   send(request);
   return receive(reply,T)    // return false if timeout;
}
```

What applications can this be used for?

- **No fault tolerance measures!!** If the call fails after timeout, the caller cannot tell whether the procedure was executed, whether request/reply were lost or the server crashed.
- This is known as **maybe call semantics** because if the call fails the client cannot tell for sure whether the remote procedure has been called or not. If the call succeeds, the procedure will have been executed exactly once if using a communication service which does not generate duplicate messages.
- Lightweight but leaves issues of state consistency of the server, with respect to the client, up to the application programmer.

At-Least-Once Call Semantics

```
bool call (request, reply) {
        int retries = n;
        while(retries--) {
                send(request);
                if (receive(reply,T)) return true;   }
        return false; // return false if timeout;     }
```

Retries up to n times – if the call succeeds then procedure has been executed one or more times as duplicate messages may have been generated.

If the call fails, a permanent communication failure (e.g., network partition) or server crash is a probability.

Useful for idempotent server operations i.e they may be executed many times and have the same effect on server state as a single execution.

Sun RPC supports this semantic since it was originally designed for NFS in which the file operations are idempotent and servers record no client state. A call may thus succeed if the server has crashed and recovered within time n * T.

At-Most-Once Call Semantics

This guarantees that the remote procedure is either never executed or executed partially (due to a server crash) or once.

To do this, the server must keep track of request identifiers and discard retransmitted requests that have not completed execution. On the other hand, the server must buffer replies and retransmit until acknowledged by the client.

Most RPC systems guarantee at-most-once semantics in the absence of server crashes.

Why in the absence of server crashes?

Transactional Call Semantics

(Zero or Once)

This guarantees that either the procedure is completely executed or it is not executed at all.

To ensure this, the server must implement an atomic transaction for each RPC i.e., either the state data in the server is updated permanently by an operation taking it from one consistent state to another or it is left in its original state, if the call is aborted or a failure occurs.

This requires **two phase commit** type of protocol.

Election Service Implementation

Give a pseudocode implementation for the Election Service (server only) which would permit the interface to be invoked using an RPC mechanism. The RPC implementation supports **at-least-once** calling semantics but clients must only vote once.

RPC Summary

Often specific to a particular programming language or operating system

RPC calls suspend client for
network roundtrip delay + procedure execution time

Not suitable for multimedia streams or bulk data transfer

Not easy to use

No reuse of interface specifications

Servers are usually a heavyweight OS process.

Object Interactions (e.g., Java RMI)

References

Latest Java documentation from

- http://www.oracle.com/technetwork/java/index.html

RMI Tutorials:

- http://docs.oracle.com/javase/tutorial/rmi/

Couloris ch. 5, Tanenbaum 2.3

Wollrath, A, Riggs, R and J. Waldo. A Distributed Object Model for the Java System. Proc. Usenix 1996, Toronto

Object Interaction vs. RPCs

Encapsulation via fine to medium grained objects (e.g. threads or C++ objects)

- Data and state only accessible via defined interface operations
- RPC based systems -> encapsulation via OS processes

Portability of objects between platforms.

- RPC clients and servers are not usually portable

Typed interfaces

- Object references typed by interface -> bind time checking
- RPC interfaces often used in languages which do not support type checking

Support for inheritance of interfaces

- Use inheritance to extend, evolve, specialise behaviour.
- New server objects with extended functionality (subtypes) can replace existing object and still be compatible with clients.
- RPC replacements must have identical interface i.e., usually no inheritance.

Interaction Types

- Two-way synchronous invocation c.f. RPC – Java

Pass objects as invocation parameters (Java only)

Parameterised exceptions -> Simpler error handling

Location transparency

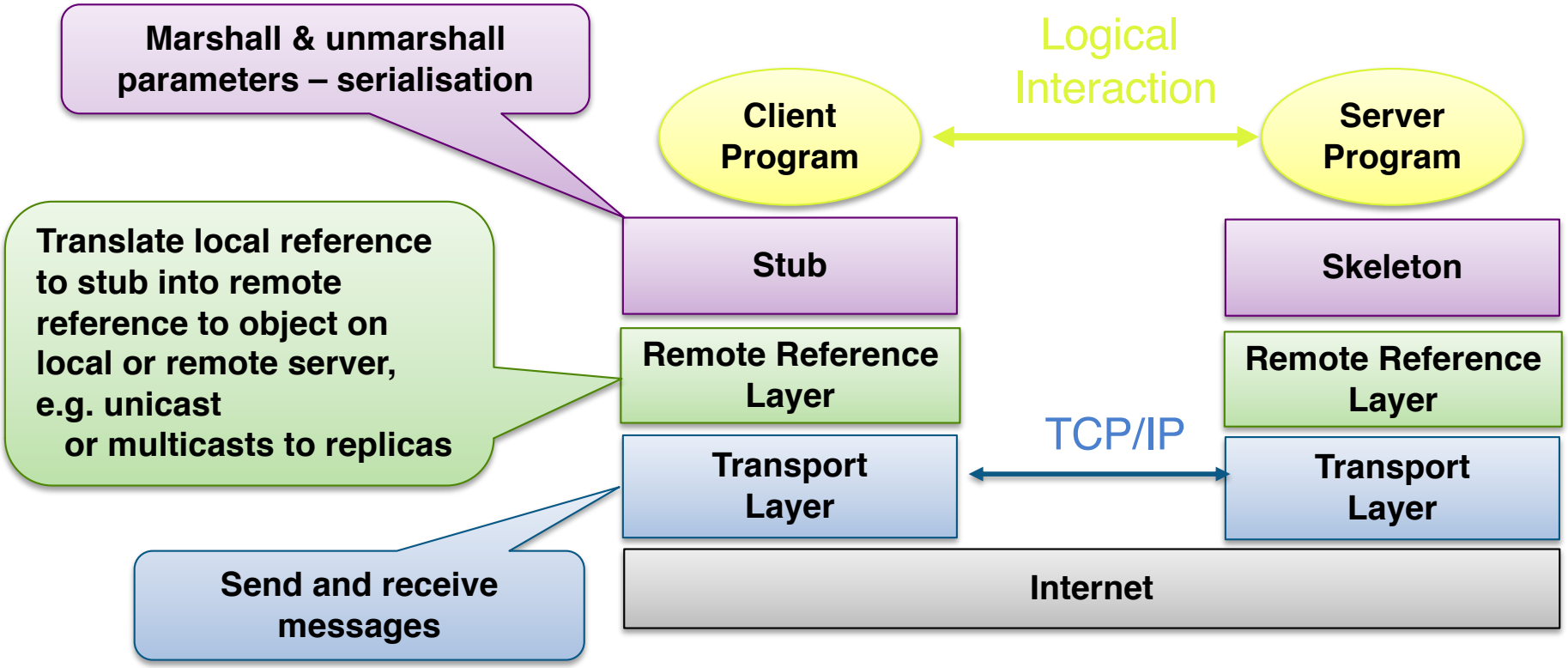
- Service use orthogonal to service location

Access transparency

- Remote and co-located services accessed by same method invocation.
- RPC only used for remote access.

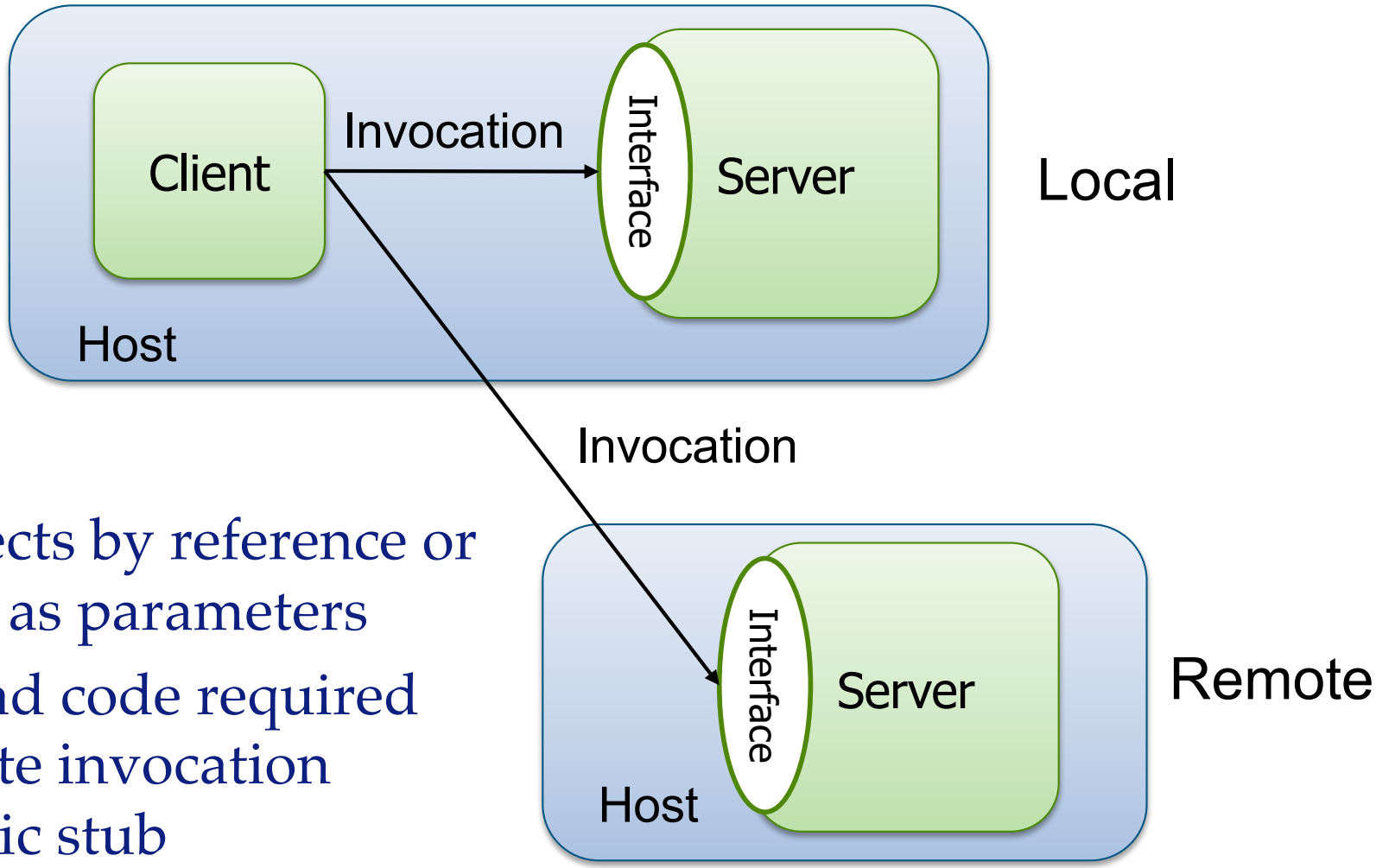
Use invocations to create/destroy objects
RPC systems (often) use OS calls to create/destroy processes

Java RMI Architecture



See http://docs.oracle.com/javase/tutorial/rmi/index.html

Transparent Invocation



Pass objects by reference or by value as parameters

Download code required for remote invocation – dynamic stub

Java Interfaces

Java is a class-based OO programming language.
Supports single inheritance

A Java **interface** defines a new type

- a collection of methods (and constant definitions)
- methods and constants declared in an interface are implicitly public

An interface may be derived from **one or more** further interfaces

A class can implement **one or more** interfaces

- as well as being derived from at most one other class

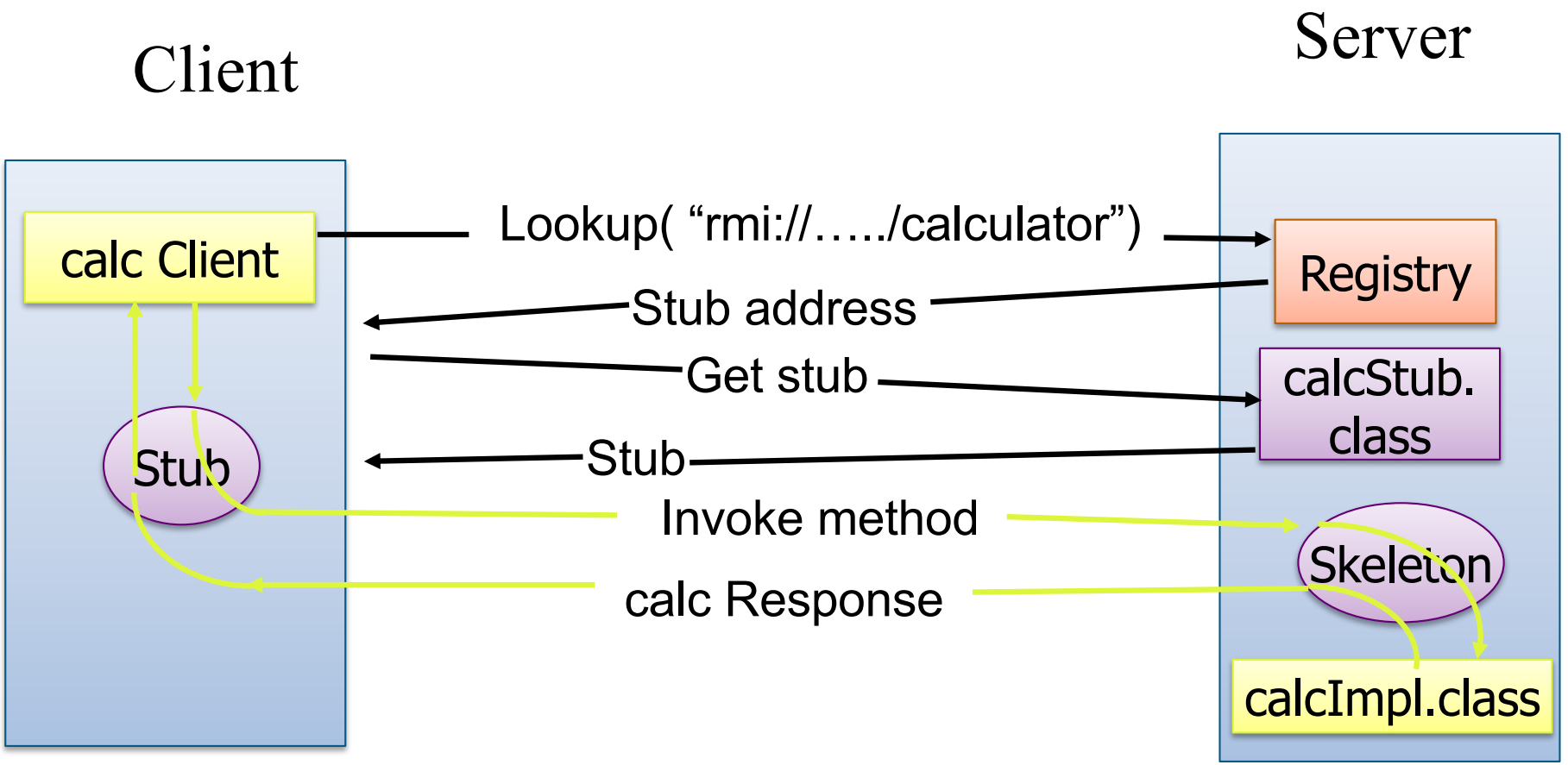
Remote Interface

A type whose interfaces may be invoked remotely is defined as a remote interface. A remote interface extends the java.rmi.Remote and must be public

The methods of a remote interface must be defined to throw the exception java.rmi.RemoteException for comms failures.

```
import java.rmi.*
public interface Calculator extends Remote {
   public long add(long a, long b) throws RemoteException;
   public long sub(long a, long b) throws RemoteException;
   public long mul(long a, long b) throws RemoteException;
   public long div(long a, long b) throws RemoteException;
}
```

Client Server Interaction



Note: skeleton not needed in later versions of Java

Remote Objects

Remote objects are instances of classes that implement **remote interfaces** eg. `CalculatorImpl` implements `Calculator`. Coulouris calls them servants. A remote object class simply implements the methods defined in the remote interface

Remote objects execute within a **server** which may contain multiple remote objects. An object is implicitly **exported** if its class derives from `java.rmi.server.UnicastRemoteObject`

Note operations invoked on remote objects, not on server containing them.

Remote Object Implementation

```
import javarmi.*
public class CalculatorImpl
   extends UnicastRemoteObject
   implements Calculator {
```

UnicastRemoteObject constructor exports the object as single server – not replicated

```
    public CalculatorImpl() throws RemoteException {
       super();
      }
```

Call to super activates code in UnicastRemoteObject for RMI linking & object initialisation

```
   public long add(long a, long b) throws RemoteException {
      return a + b;
   }
     public long sub(long a, long b) throws RemoteException {
      return a - b;
   }
   . . .
    }
```

Server Implementation

A server program creates one or more remote objects as part of mainline code. For simple single object applications it is possible to combine server and object implementation.

A server may advertise references to objects it hosts via the local RMI registry

Registry allows a binding between a URL and an object reference to be made and subsequently queried by potential clients

Server Implementation

The server listens for incoming invocation requests which are dispatched to appropriate object.

Note: there may be multiple servers and multiple clients within an application. Client is not created within a server.

Server Mainline code

```
import java.rmi.Naming;
public class CalculatorServer {
  public static void main(String args[]) {
    if System.getSecurityManager() == null {
            System.setSecurityManager (new RMIsecurityManager ());
    }
    try {
     Calculator c = new CalculatorImpl();
     Naming.rebind("rmi://localhost/CalcService", c);
            }
     catch (Exception e) {
     System.out.println("Trouble: " + e);
    }
  }
}
```

Create security manager

Create server object

Register it with the local registry: URL-reference binding

Calculator Client Implementation

```
import java.rmi.*;
import Calculator;
public class CalculatorClient {

  public static void main(String[] args) {
   try {
            if System.getSecurityManager() == null {
              System.setSecurityManager (new RMIsecurityManager ());
              Calculator c = (Calculator) Naming.lookup(
                                                  "rmi://remotehost/CalcService");
```

Get ref to CalcServer stub from remote registry

```
      System.out.println( c.sub(4, 3) );
```

Invoke sub operation on remote calculator

```
      . . .  other calls ;
          } catch (RemoteException e) {
      System.out.println(" Exception:" + e);
   }
}
```

RMIRegistry

- Must run on every server computer hosting remote objects
- Advertises availability of Server's remote objects
- name is a URL formatted string of form **`//host:port/name`**. Both host and port are optional

Functions

- lookup (String name) called by a remote client. Returns remote object bound to **name**
- bind(String name Remote obj) Called by a server – binds name to remote object obj Exception if name exists
- rebind(String name Remote obj) binds name to object obj discards previous binding of name (safer than bind)
- unbind(String name) removes a name from the registry
- String [] list() returns an array of strings of names in registry

Using Registry

Server remote object making itself available

```
Registry r = Locateregistry.getRegistry();
r.rebind ("myname", this)
```

Remote client locating the remote object

```
Registry r =
   LocateRegistry.getRegistry("thehost.site.ac.uk");
RemObjInterface remobj =
                        (RemObjInterface) r.lookup ("myname");
remobj.invokemethod ();
```

RMI Security Manager

Single constructor with no arguments

```
System.setSecurityManager(new RMISecurityManager());
```

Needed in server and in client if stub is loaded from server

Checks various operations performed by a stub to see whether they are allowed eg

- Access to communications, files, link to dynamic libraries, control virtual machine, manipulate threads etc.

In RMI applications, if no security manager is set, stubs and classes can only be loaded from local classpath – protect application from downloaded code

Parameter Passing

Clients always refer to remote object via remote interface type not implementation class type

A reference to a remote object can be passed as a parameter or returned as a result of any method invocation

Remote objects passed by reference – stub for remote object is passed

Given two references, r1 and r2, to a remote object (transmitted in different invocations):

- r1 == r2 is false -> different stubs
- r1.equals (r2) is true -> stubs for same remote object

Parameters can be of any Java type that is serialisable

- primitive types, remote objects or objects implementing java.io.serializable
- Non-remote objects can also be passed and returned by value i.e. a copy of the object is passed

  -> new object created for each invocation

Garbage Collection of Remote Objects

RMI runtime system automatically deletes objects no longer referenced by a client

- When live reference enters Java VM, its reference count is incremented
- First reference sends “referenced” message to server
- After last reference discarded in client “unreferenced” message sent to server.
- Remote object removed when no more local or remote references exist.

Network partition may result in server discarding object when still referenced by client, as it thinks client crashed

Dynamic Invocation

Single method interface

Invocation identifies method to be called + parameters

User programs marshalls/demarshalls parameters

Optional invocation primitive for object environments such as CORBA and for Web services.

```
public byte[] doOperation (RemoteObjectRef o,
                           int methodId, byte[] arguments)
```

- Sends a request message to the remote object and returns the reply.
- The arguments specify the remote object, the method to be invoked and the arguments of that method.
- Server has to decode request and call method

Summary

RMI provides access transparency, object oriented concepts for IDL specification, object invocations and portability.

Inheritance supports reuse -> high level programming concepts

High implementation overheads due to

- Byte code interpretation in Java
- Marshalling/Demarshalling of parameters
- Data copying
- Memory management for buffers etc.
- Demultiplexing and operation dispatching