# *Interaction Primitives*

- ➢ Message passing
  - ◆ send & receive primitives
  - ◆ synchronisation
  - ◆ naming
- ➢ Remote procedure call
  - ◆ IDL
  - ◆ semantics
- ➢ Object invocation (in next section)

Introduction

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

♦ A distributed system must make 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

♦ It is feasible to support shared memory abstractions such as shared objects or Linda tuple spaces implemented above message passing

---

Inter Process Communication (IPC)

Host Node — Logical Inter Process Communication — Host Node

Communication Network

Communications System
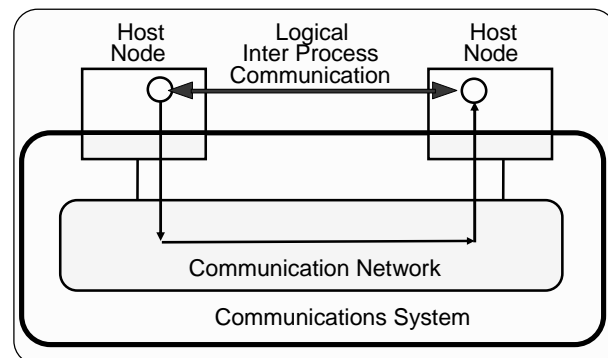
Communication Service Characteristics

The underlying communication service can be based on connectionless or connection oriented protocols

**Connectionless (Datagram)**
- ➢ "Send and pray" message could be lost, duplicated or delivered out-of-sequence. User not told.
- ➢ Potentially broadcast or multidestination
- ➢ Maintains no state information ➔ cannot detect lost, duplicate or out-of-sequence messages.
- ➢ Each message contains full source & destination address
- ➢ May discard messages which are corrupted or because of congestion ➔ No error correction
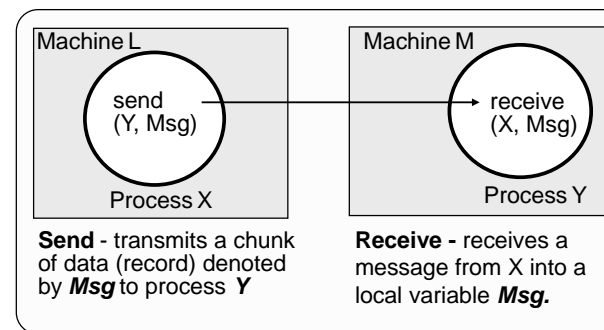
Communication Service 2

Connection Service

➢ Reliable, in-sequence delivery of messages

➢ Performs error and flow control ➔ errors reported to both ends

➢ Overheads:

 ◆ Time: establishment and termination of connection,
   error control protocol overheads.

 ◆ Space: state information maintained at both ends:

  ● unacknowledged messages

  ● remote entity address

  ● message sequence numbers

  ● flow control information

TCP or UDP
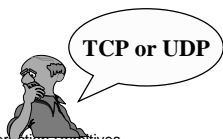
Different primitives require different services

Message Passing

Machine L	Machine M
send (Y, Msg) — Process X	receive (X, Msg) — Process Y

Send - transmits a chunk of data (record) denoted by **Msg** to process **Y**

Receive - receives a message from X into a local variable **Msg.**

*Which communication service does this map onto?*

Naming

➢ Both sender and receiver directly name each other in their programs

* *What are the disadvantages?*

* *What are the alternatives?*

Communication Patterns

Many - to - One
Many sender processes
S1,…,Sn may send to a single
receiver. The receiver R receives
the messages one at a time
from a message queue.
Typical client-server
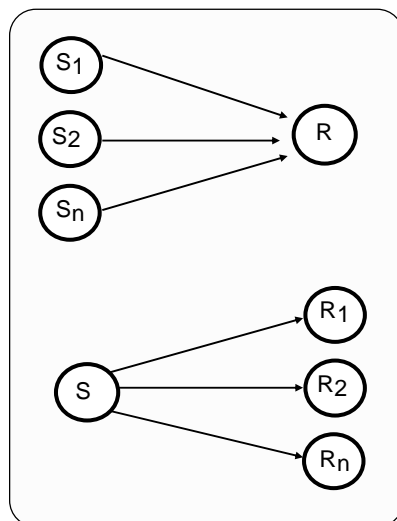
One - to - many
A single sender S sends the
message to many receivers
R1,…,Rn.
LANs such as ethernet provide
hardware support for broadcast
(all) and multicast (subset)).

Message 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)
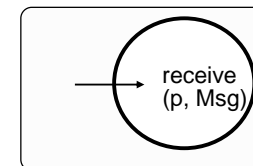
Variations:

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)

* *What additional blocking primitive is needed for a server receiving from multiple different clients?*

Asynchronous Send

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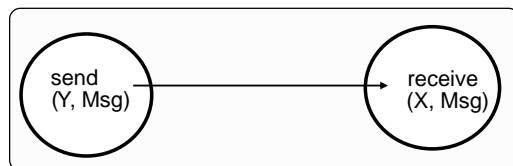
Characteristics:

Mostly used with blocking receive

Underlying system must provide buffering (usually at the receive end)

since many messages may be sent before the first is received

Loose coupling between sender and receiver(s)

Readily usable for multicast

Efficient implementation

Asynchronous Send

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)
- 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)
        Communication Service expected is "reliable".

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).

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

Bytes in Message	Length of message	Internet address	Port number

  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 between sockets. setSoTimeout for a receive to limit how long it blocks. If the timeout expires it throws an interruptedIOException.

> Tutorial http://java.sun.com/docs/books/tutorial/networking/sockets/

> Coulouris Ch. 4

12

UDP Client

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[ ]){    // args give message contents & destination hostname
      DatagramSocket aSocket = null;
      try {  aSocket = new DatagramSocket();
           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());
} }
```
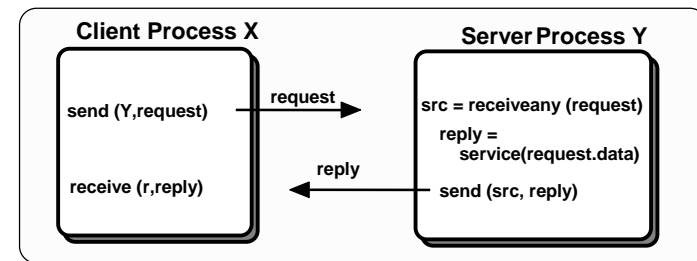
13

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);
             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());
   } }
```

14

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.
> E.g. HTTP GET and POST

> *Assuming blocking receive, how many message buffers required at the server for N clients?*

15

Message Passing Summary

Although message passing systems encourage encapsulation and information hiding, OS level message interaction primitives e.g., sockets are considered as the assembly language level of distributed systems. They are considered 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 can be used to implement higher level primitives such as Remote Procedure Call or Object invocation**

Remote Procedure Call (RPC)

➢ Basic message passing leaves a lot of work for the programmer when implementing client-server interactions e.g. constructing messages, transforming data types for heterogeneous systems, RPC is a technique which helps to reduce the effort required to program such interactions.

➢ Essentially, an attempt is made 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 a request message and the results returned in a reply message. Since all imperative languages have a procedure calling mechanism, RPC integrates well with conventional programming languages.

➢ Of course since 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
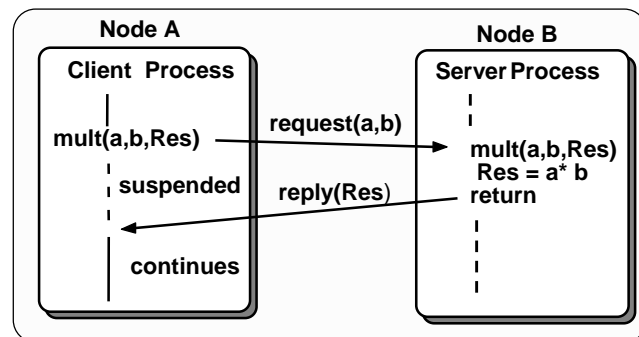
➢ Client is suspended until the call completes.

➢ Parameters must be passed by value since Client and Server processes do not share memory and consequently Client pointers have no meaning in Server address space.

RPC Interactions

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.

➢ These can be generated automatically from the interface specification.

➢ **Performs:**

◆ 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.

➢ *What failures must RPCs cope with?*
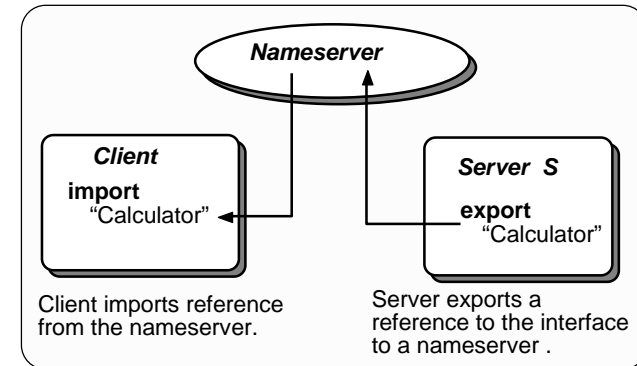
Interface Definition Language (IDL)

> IDL is a data typing language used to specify the interface operations and their parameters.
> The following Pseudocode example shows how RPC is used

> **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.

Client imports reference from the nameserver.

Server exports a reference to the interface to a nameserver .

* *How do you bind to the nameserver?*

Calculator Client Code

The following shows fragments of client code using the *calc* interface

```
#include "calc.idl"

void main()  {
   status = import  (calc c, "calculator", nameserverAddress);
```

Lookup calculator in nameserver and bind to c

```
      c.mult(1.2, 5.6, Res);
```

Use remote procedure reference c to invoke mult operation

```
      Print (Res);
}
```

Calculator Server Code

The following shows fragments of server code using the *calc* interface.
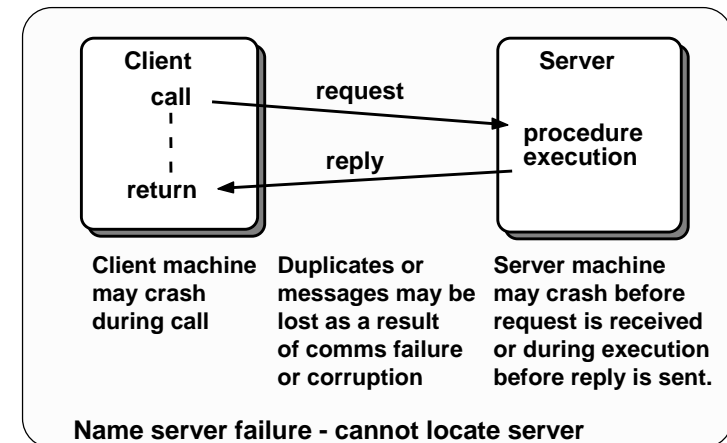
```
#include "calc.idl"

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;
}
```

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 candidates 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.

Client machine may crash during call

Duplicates or messages may be lost as a result of comms failure or corruption

Server machine may crash before request is received or during execution before reply is sent.

Name server failure - cannot locate server

RPC Failures

➤ 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.
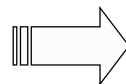
Maybe (Best-Efforts) Call Semantics

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

➤ No fault tolerance measures!! If the call fails after timeout, the caller cannot tell whether the procedure was executed or whether the request message was lost or the server crashed. On the other hand the call may have been executed and the reply message lost.

➤ 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.

➤ *What applications can this be used for?*

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

> Interaction primitive similar to but not the same as well known procedure call

- different local and remote primitives
  cannot move co-located components after compilation

> May result in deadlock
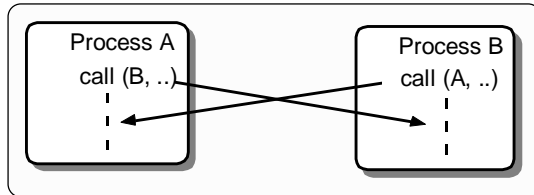
RPC Summary

◊ 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.