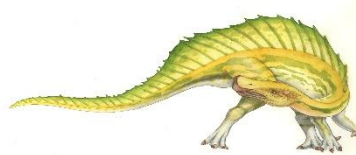




Message Passing (Cont.)

- Implementation of communication link
 - Physical:
 - ▶ Shared memory
 - ▶ Hardware bus
 - ▶ Network
 - Logical:
 - ▶ Direct or indirect
 - ▶ Synchronous or asynchronous
 - ▶ Automatic or explicit buffering





Direct Communication

- Processes must name each other explicitly:
 - **send** (P , *message*) – send a message to process P
 - **receive**(Q , *message*) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional





Indirect Communication

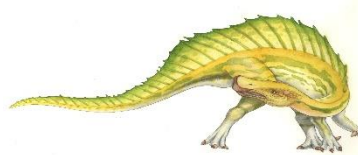
- ❑ Messages are directed and received from mailboxes (also referred to as ports)
 - ❑ Each mailbox has a unique id
 - ❑ Processes can communicate only if they share a mailbox
- ❑ Properties of communication link
 - ❑ Link established only if processes share a common mailbox
 - ❑ A link may be associated with many processes
 - ❑ Each pair of processes may share several communication links
 - ❑ Link may be unidirectional or bi-directional





Indirect Communication

- Operations
 - create a new mailbox (port)
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
 - send**(*A, message*) – send a message to mailbox A
 - receive**(*A, message*) – receive a message from mailbox A





Indirect Communication

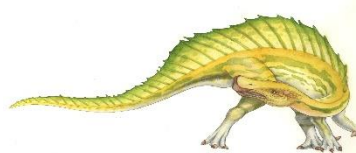
- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 sends; P_2 and P_3 receive
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.





Synchronization

- ❑ Message passing may be either blocking or non-blocking
- ❑ **Blocking** is considered **synchronous**
 - ❑ **Blocking send** -- the sender is blocked until the message is received
 - ❑ **Blocking receive** -- the receiver is blocked until a message is available
- ❑ **Non-blocking** is considered **asynchronous**
 - ❑ **Non-blocking send** -- the sender sends the message and continue
 - ❑ **Non-blocking receive** -- the receiver receives:
 - ❑ A valid message, or
 - ❑ Null message
- ❑ Different combinations possible
 - ❑ If both send and receive are blocking, we have a **rendezvous**





Synchronization (Cont.)

- Producer-consumer becomes trivial

```
message next_produced;  
while (true) {  
    /* produce an item in next produced */  
    send(next_produced);  
}
```

```
message next_consumed;  
while (true) {  
    receive(next_consumed);  
  
    /* consume the item in next consumed */  
}
```





Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits





Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)





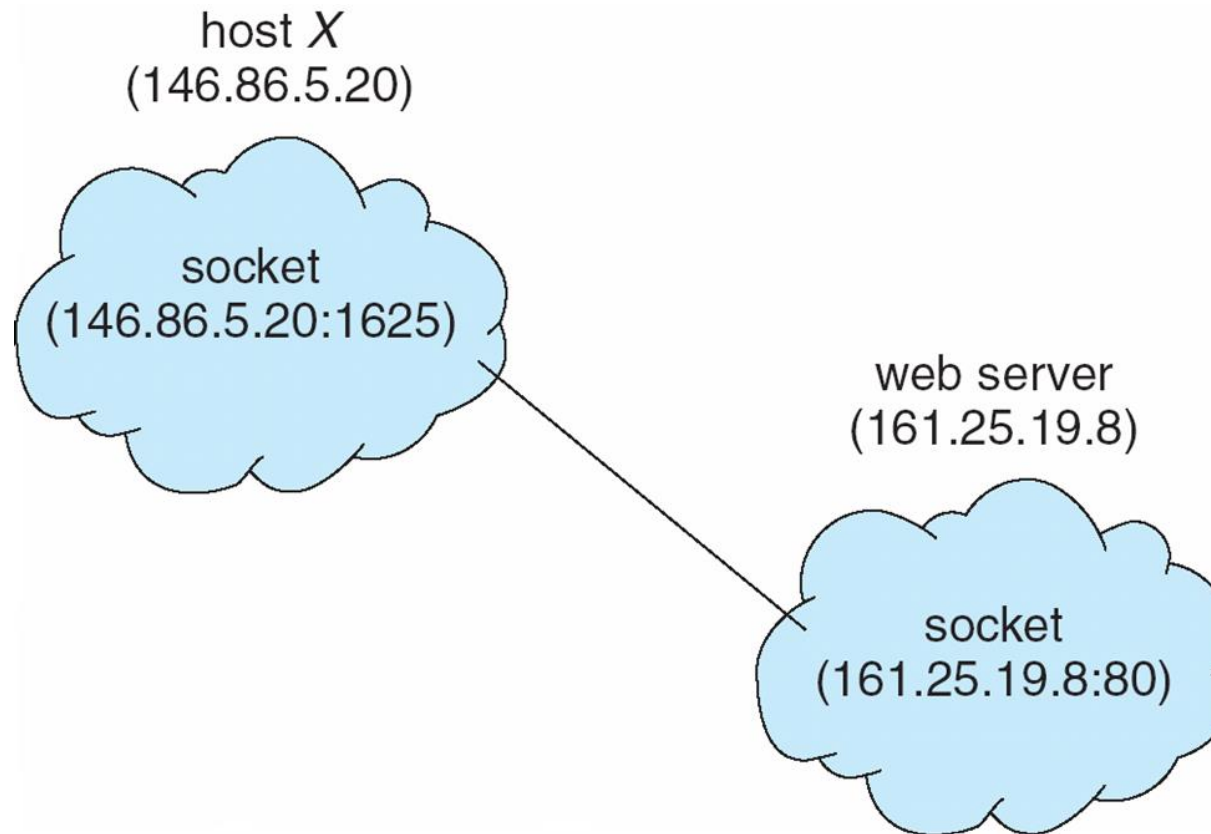
Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running





Socket Communication





Sockets in Java

- Three types of sockets
 - **Connection-oriented (TCP)**
 - **Connectionless (UDP)**
 - **MulticastSocket** class— data can be sent to multiple recipients
- Consider this “Date” server:

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

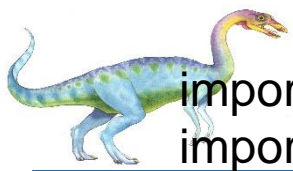
            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```





```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1",6013);
            InputStream in = sock.getInputStream();
            BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));
            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);
            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

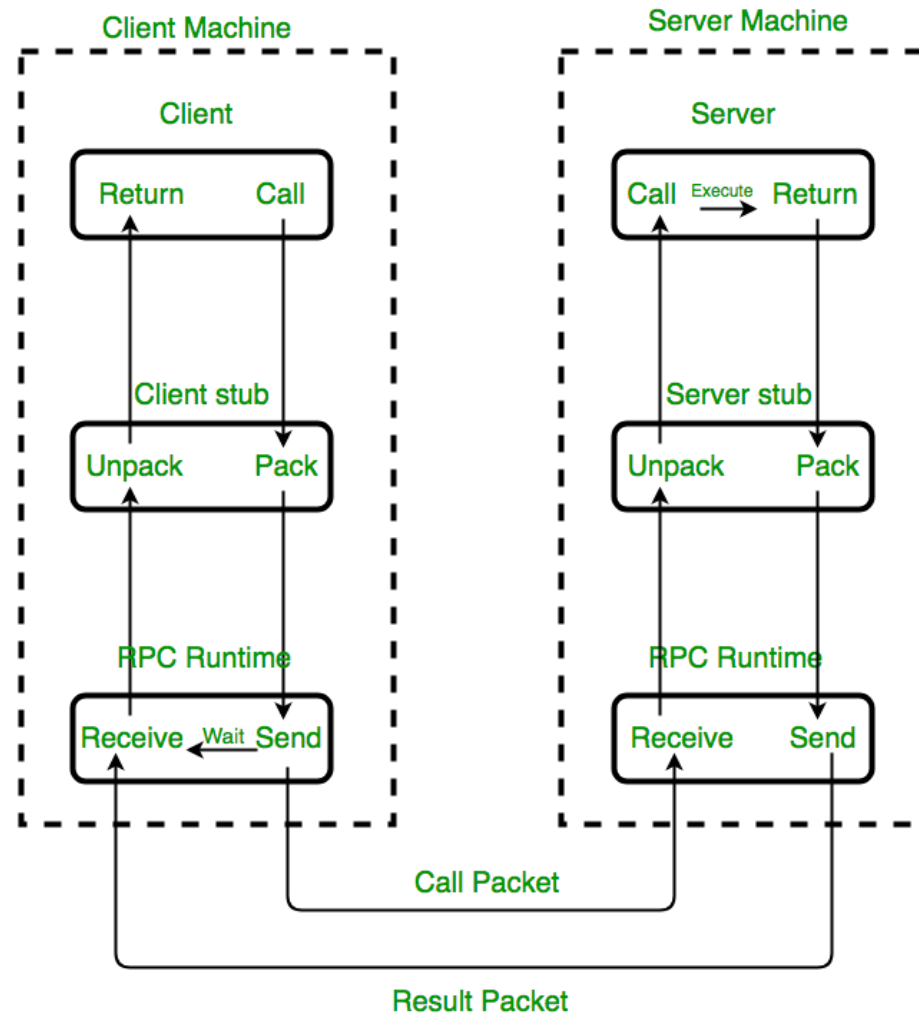




Remote Procedure Calls

- ❑ Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
 - ❑ Again uses ports for service differentiation
- ❑ **Stubs** – client-side proxy for the actual procedure on the server
- ❑ The client-side stub locates the server and **marshalls** the parameters
- ❑ The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- ❑ On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**



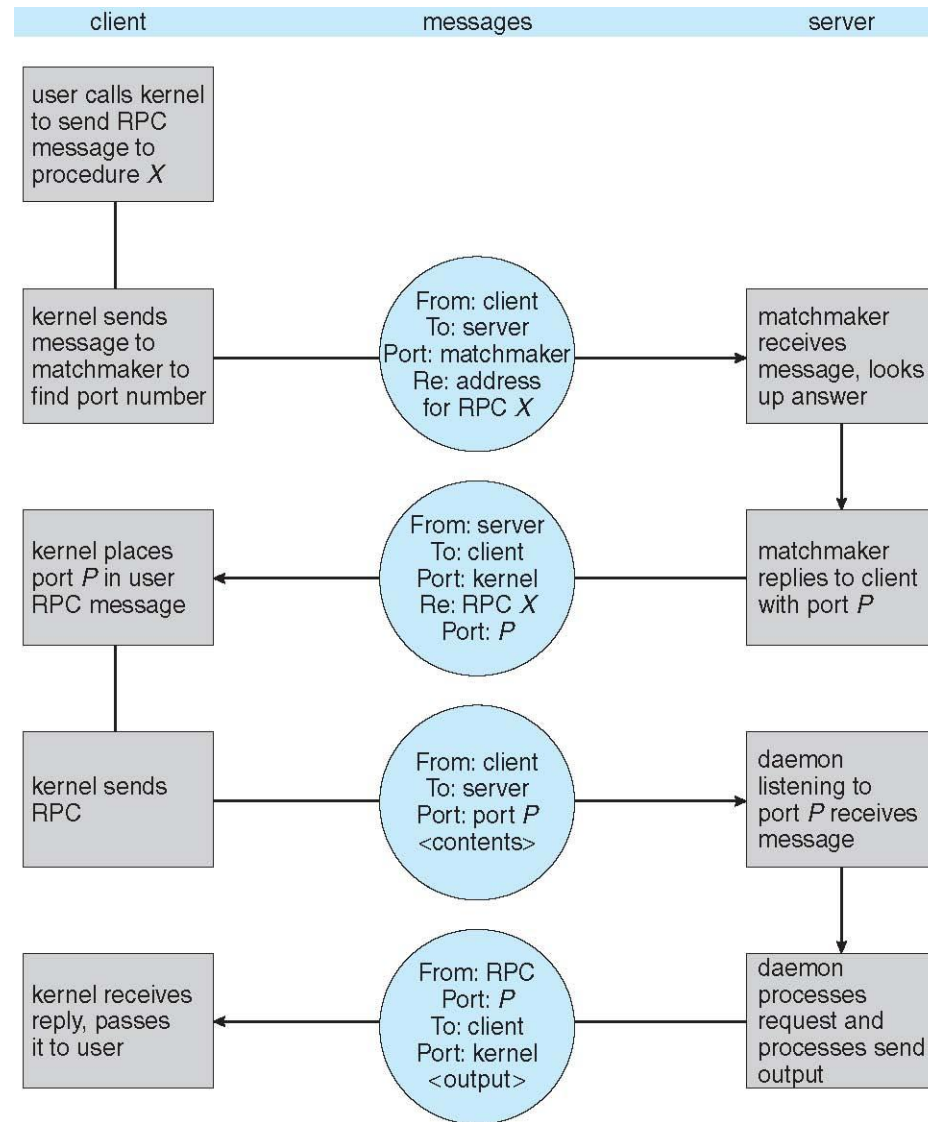


Implementation of RPC mechanism





Execution of RPC





Pipes

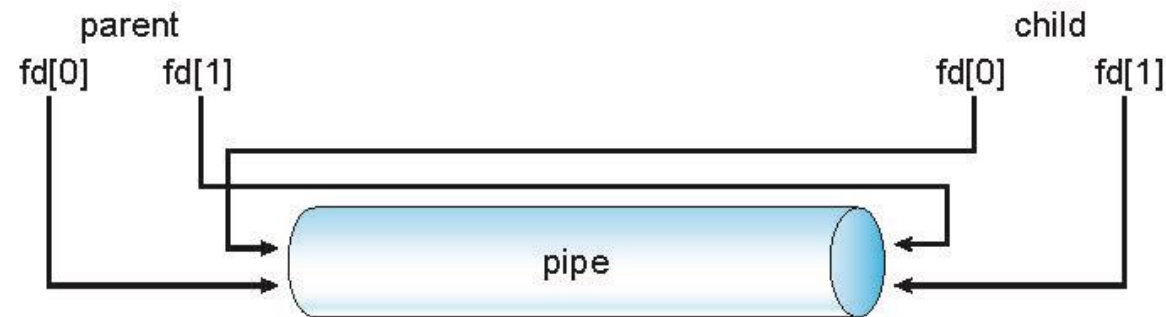
- Acts as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
 - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.





Ordinary Pipes

- ? Ordinary Pipes allow communication in standard producer-consumer style
- ? Producer writes to one end (the **write-end** of the pipe)
- ? Consumer reads from the other end (the **read-end** of the pipe)
- ? Ordinary pipes are therefore unidirectional
- ? Require parent-child relationship between communicating processes



- ? Windows calls these **anonymous pipes**
- ? See Unix and Windows code samples in textbook





Named Pipes

- ❑ Named Pipes are more powerful than ordinary pipes
- ❑ Communication is bidirectional
- ❑ No parent-child relationship is necessary between the communicating processes
- ❑ Several processes can use the named pipe for communication
- ❑ Provided on both UNIX and Windows systems



End of Chapter 3

