Concurrent & Distributed Systems

Distributed Systems 2

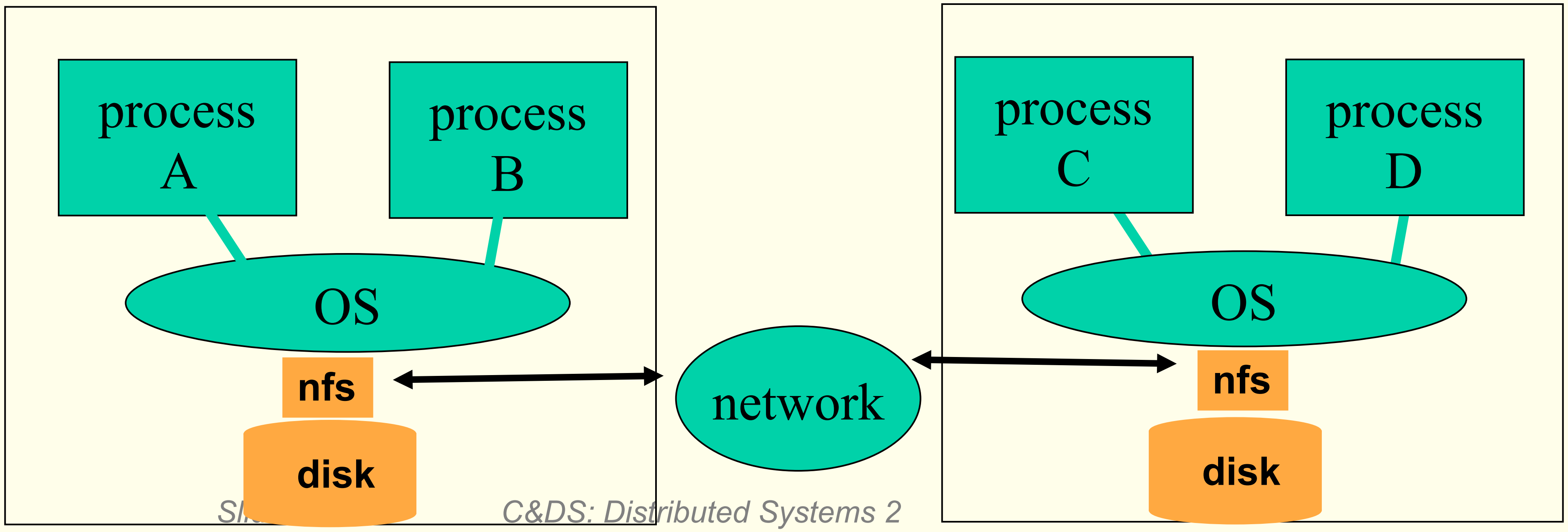
distributed communications

distributed communication

- files
- sockets
- message passing
- remote procedure calls
- remote method invocation

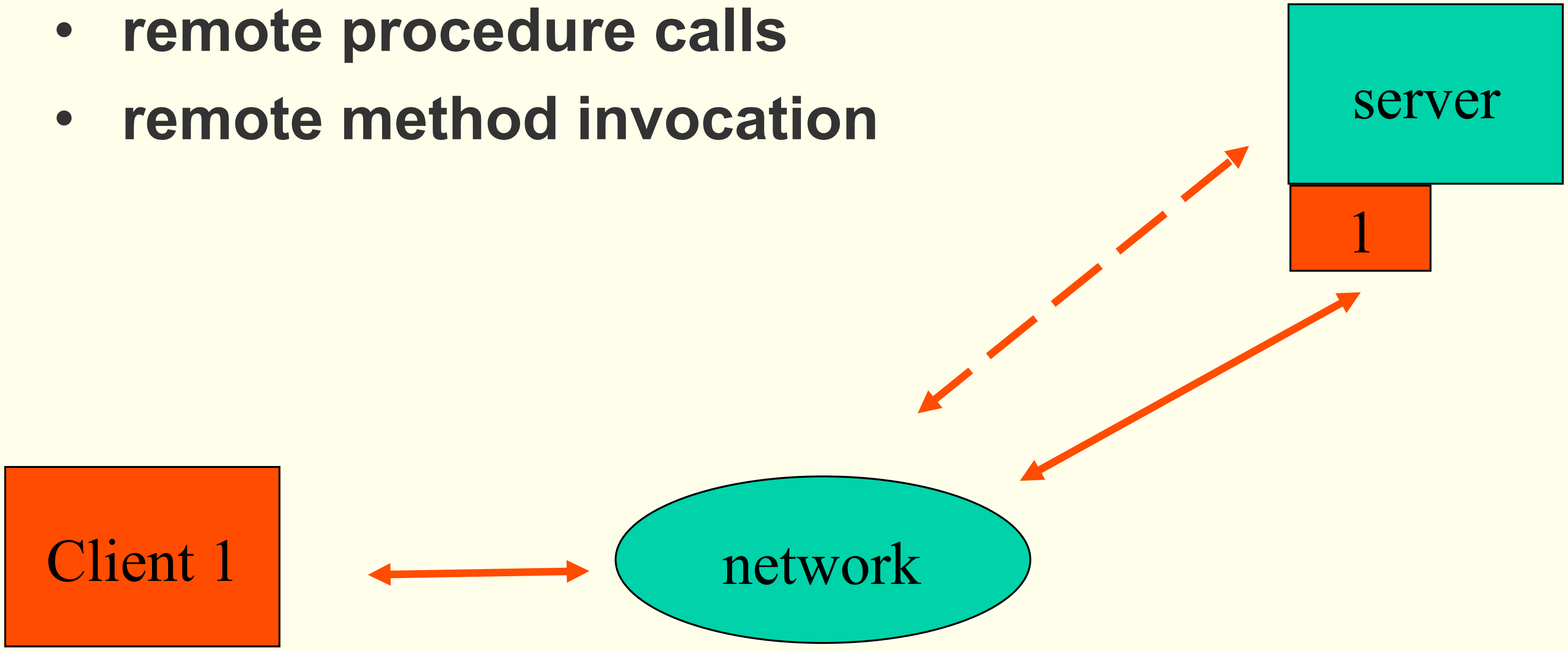
files

- This is really here for completeness
- could be used if one machine
- if distributed across sites we can “cheat” using a distributed file system such as NFS, e.g. home/~usr/temp-files/buffer
- slow
- awkward



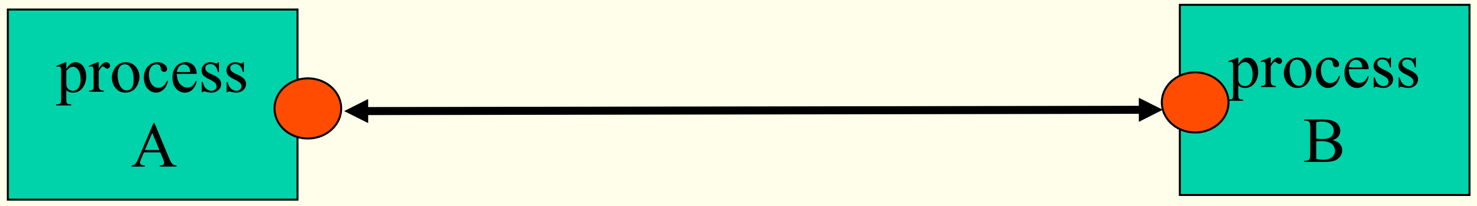
distributed communication

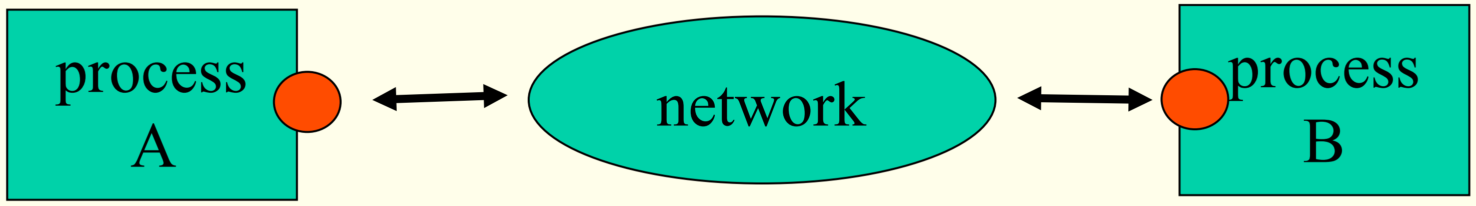
- files
- sockets
- message passing
- remote procedure calls
- remote method invocation



sockets

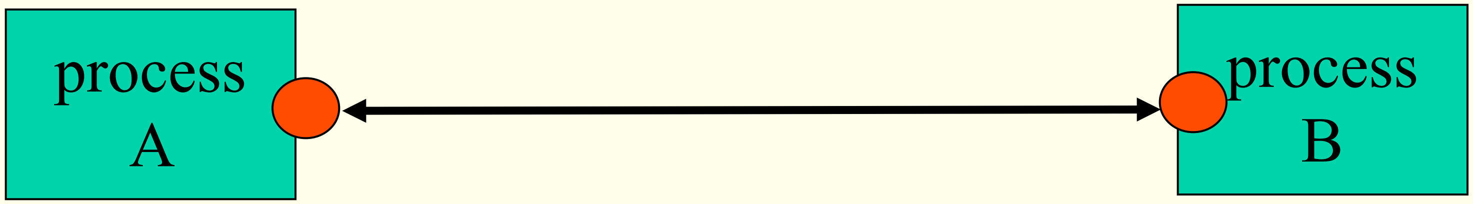
- A <u>socket</u> is a communications endpoint
- a “channel” in-between two sockets permits 2-way communication
- no emphasis on physical connection - assume a logical IP connection





socket id

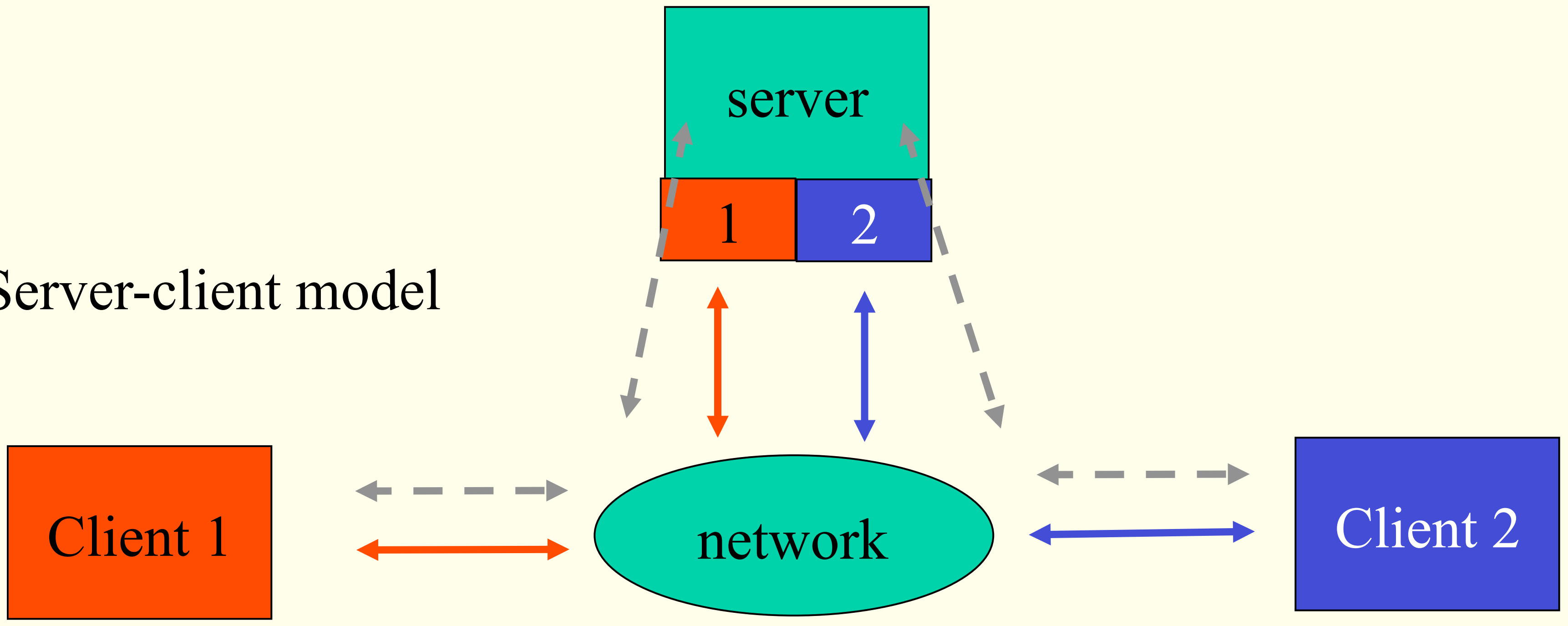
- **The id includes both IP address and a port number**
- **146.86.5.20/80**
- **port number <= 1024 are *well known***
  - **23 for telnet**
  - **22 for sftp**
  - **21 for ftp**
  - **80 for http**
- **so an http *server* would use port 80**
- **telenet, ftp, sftp, http, smtp live on top of sockets**
- **often used in server-client configuration**



server - client revisited

- first setup connection - the server often has a process awaiting requests
- normally the server fires up a new thread for each successful request

Server-client model



The server

```
import java.net.*;

public class Server{

   public Server(){
       // create the socket the server will listen to
       try {s = new ServerSocket(5155);}
       catch (java.io.IOException e)
       {System.out.println(e); System.exit(1);}

       System.out.println("Server is listening ....");

       // OK, now listen for connections and create them
       try {
          while (true) {
          >>>>>>  client = s.accept();
          // create a separate thread to service the client
          >>>>>>  c = new Connection(client);
                  c.start(); }
          }
       catch (java.io.IOException e) {System.out.println(e);}
   }
```

PTO->

The server

```
import java.net.*;
public class Server{

…

   public static void main(String args[]){
        private ServerSocket   s;
        private Socket         client;
        private Connection     c;
        Server timeOfDayServer = new Server();
   }

}
```

the server

java.net

Class ServerSocket

```
java.lang.Object
    java.net.ServerSocket
```

All Implemented Interfaces: `Closeable,AutoCloseable`

Direct Known Subclasses: `SSLServerSocket`

```
public class ServerSocket extends Object implements Closeable
```

This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester. The actual work of the server socket is performed by an instance of the SocketImpl class. An application can change the socket factory that creates the socket implementation to configure itself to create sockets appropriate to the local firewall.

Since: JDK1.0   See Also:

`SocketImpl,setSocketFactoryk(java.net.SocketImplFactory), ServerSocketChannel`

In the class `ServerSocket` we have

```
public Socket accept()
      throws IOException
```

Listens for a connection **to be made to this socket** and accepts it.

The method **blocks** until a connection is made.

A new **Socket s is created** and, if there is a security manager, the security manager's checkAccept method is called with s.getInetAddress().getHostAddress() and s.getPort() as its arguments to ensure the operation is allowed. This could result in a SecurityException.

Returns: the new Socket

the server-side connection

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

public class Connection extends Thread{
   public Connection(Socket s) {
        outputLine = s;
   }

   public void run() {
        private Socket outputLine;
        // getOutputStream rtns an OutputStream object
        // allowing ordinary file IO over the socket.

        // create a new PrintWriter with auto flushing
        try {PrintWriter pout =
                new PrintWriter(outputLine.getOutputStream(), true);
        // now send a message to the client
        pout.println("The Date and Time is " + new
                                 java.util.Date().toString());
        // now close the socket
        outputLine.close();
        }
        catch (java.io.IOException e) { System.out.println(e);}
   }}
```

the client

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

public class Client{

   public Client() {
        try { Socket s = new Socket("127.0.0.1", 5155);
              InputStream in = s.getInputStream();
              BufferedReader bin = new BufferedReader
                                       (new InputStreamReader(in));

              System.out.println(bin.readLine());
              s.close();
            }
        catch (java.io.IOException e) {
               System.out.println(e); System.exit(1);
            }
   }

   public static void main(String args[]) {
        Client client = new Client();
   }}
```

distributed communication

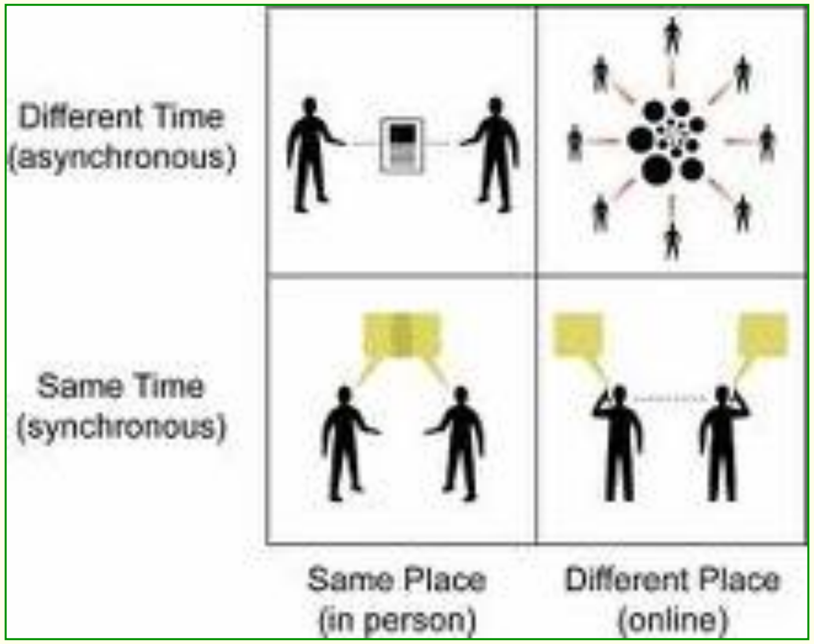
- files
- sockets

✉ message passing

- remote procedure calls
- remote method invocation

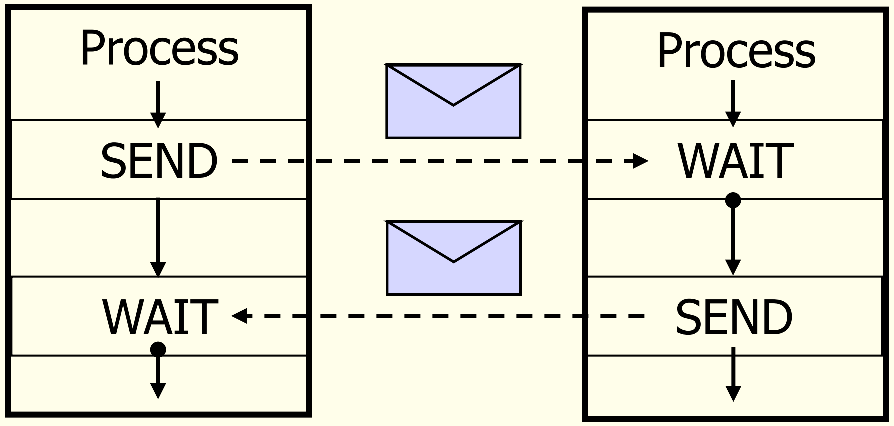
asynchronous message passing

- Been in use since 1960s
- Message passing systems support both data transfer and synchronisation
- Even in shared memory systems processes must synchronise to exchange information
- Messages typically contain:
  - A header containing:
    - Destination
    - Source
    - Type of message
  - Message body



variations of message passing

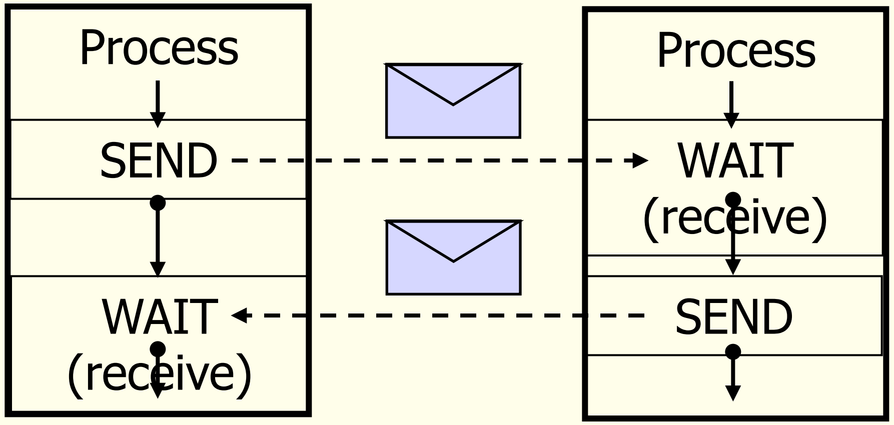
- **Textbook: Bacon §12.7**
- **The initial approach adopted was *asynchronous message passing*.**



- **Here, the sending process is *not* delayed.**
- **This means that the system must buffer the message if the receiver is not ready to receive it.**
- **This approach is also called *buffered* message passing.**

variations of message passing

- **The alternative to the previous approach is *synchronous* message passing.**



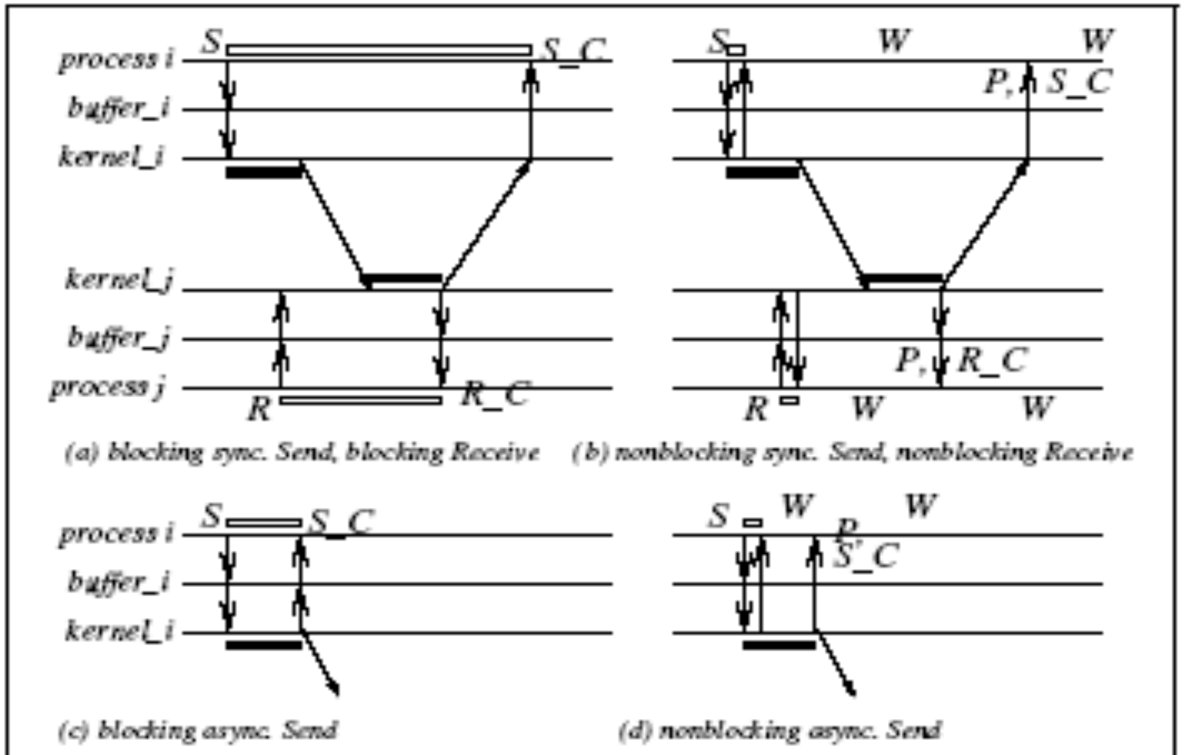
- **Here, the sender and receiver must synchronise in order for the message transfer to take place.**
  - **There is no need to buffer messages in this approach.**
- **This approach is also called *unbuffered* message passing.**

Message Passing 1

- The application marshals the data into some contiguous memory and calls the message passing interface
- The system fills in the Header with:
  - The destination, the identity of a process (or processes)
  - The identity of the sender (which namespace?)
  - The message type which may be something like:
    - initial message
    - reply_to
    - priority
    - service designator
- Header may also contain addition fields to specify transport protocol, check sum etc.

Message Passing 2

The application marshals the data into some contiguous memory and calls the message passing interface



process i
buffer_i
kernel_i
kernel_j
buffer_j
process j
S
S_C
R
R_C
W
P,
(a) blocking sync. Send, blocking Receive
(b) nonblocking sync. Send, nonblocking Receive
(c) blocking async. Send
(d) nonblocking async. Send

▬	duration to copy data from or to user buffer
▭	duration in which the process issuing send or receive primitive is blocked
S	*Send* primitive issued S_C processing for *Send* completes
R	*Receive* primitive issued R_C processing for *Receive* completes
P	The completion of the previously initiated nonblocking operation
W	Process may issue *Wait* to check completion of nonblocking operation

Message Passing 3

The application marshals the data into some contiguous memory and calls the message passing interface

- **Summing up – about blocking and synchronous:**
  - **a blocking synchronous send blocks until the *receiver* gets the data from the user remote memory & the sender is informed**
  - **a non-blocking synchronous initiates the transfer of data and then does something else (or waits) until informed that receiver has got data in user (remote) memory**
  - **a blocking asynchronous send initiates the transfer of data and blocks until it is all in kernel memory**
  - **a non-blocking asynchronous send initiates the transfer of data and then waits (or does something else) until informed the data is in local kernel memory**

Message send

- We have made a few assumptions:
  1. Each process knows the identity of the other
  2. It is appropriate for the sender to specify a single recipient and for the receiver to specify a single sender
  3. There is agreement of how to interpret the data in the message

- This last point is critical:
  - how do applications know how big the data is?
  - how do applications know how to interpret the data?

- Can use a language that allows the two ends to encode and decode the information?

Message Passing Variations (Bacon 13.5)

Variations include :

1. **The communicating parties may not know or need to know the parties with which they are communicating: example: Unix pipes**
2. **Processes may need to send a message to multiple parties (broadcast): example: alarm monitoring**
3. **May need to discriminate between messages:**

   example high/low priority in process control systems
4. **Processes may not want to WAIT forever for a single specific message: example: an acknowledgement**
5. **Messages may become out of date**

   example: real-time control systems (missiles)
6. **It may be appropriate to reply to some process other than sender**

   example: primary in replicated systems