Chapter 2: Interprocess Communication

Topics: IPC (Inter-Process Communication) primitives, blocking/nonblocking `send/receive`, transient/persistent communication, *Mach* IPC, Java and Unix sockets.

2.1 Layered Communication Protocols

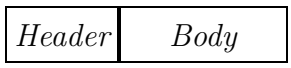
Level	Layer Name	Protocol examples
5	Application	`ftp` (file transfer), `telnet` (virtual terminal), `http`
4	Transport	TCP, UDP (Internet)
3	Network	IP

The primitives we discuss in this chapter belong to the transport layer. The transport layer of the Internet provides two types of services: **UDP** (User Datagram Protocol) service is fast but does not guarantee delivery. **TCP** (Transmission Control Protocol) service provides a reliable FIFO delivery.[1]

2.2 IPC Primitives

Unlike other synchronization mechanisms (e.g., semaphores), in which the meaning of signals is implicit, messages can carry explicit information. Since both interprocess synchronization (e.g., mutual exclusion which is commonly implemented by semaphores) and more explicit communication (exchanging data) are necessary to support concurrent process execution, it may be desirable to integrate the two functions within a single mechanism (i.e., messages only, even within the same machine, as in *Mach*), to provide greater uniformity and ease of use, at the same time reducing overhead by virtue of one mechanism.

A message consists of a **header** and a **body**.



The header typically contains the sender id, receiver id, length, type, etc. Generic message operations are

`send`(*destination*, &*msg*)

`receive`(*source*, &*buf*)

where &*msg* and &*buf* are pointers to the message to be sent and the buffer area to store the received message, respectively. The first argument can be a process (direct communication) or **mailbox** (indirect communication).

`receive`(*mailBox*, &*buf*) returns the sender identity, revealing the sender of the received msg. This type is useful for servers which may receive requests from any client. In the case of `receive`(&*buf*) without *source*, it is implied that *source* is the receiver process' **port**. A port is a special kind of mailbox, such that there is a unique process that can receive from it. Messages may be of fixed or variable length.

[1] This is usually accomplished by using sequence numbers and retransmission of lost packets; it is transparent to the user.

Design and implementation options

As shown in the table below, `send` and `receive` may come in two flavors, **blocking** and **non-blocking**. A **non-blocking** call returns after minimal delay due to local operations, so that the caller is not blocked. A **blocking** `receive` returns when a message is placed in the calling process' buffer, blocking if there is no message to be received from the specified source.

	Blocking	**Non-blocking**
`send`	blocks till the msg has been *transmitted*, or *received* by remote kernel.	returns as soon as the msg has been *queued* for subsequent transmission
`receive`	blocks until a msg is received.	returns an indication (e.g., –1), if no message is available.

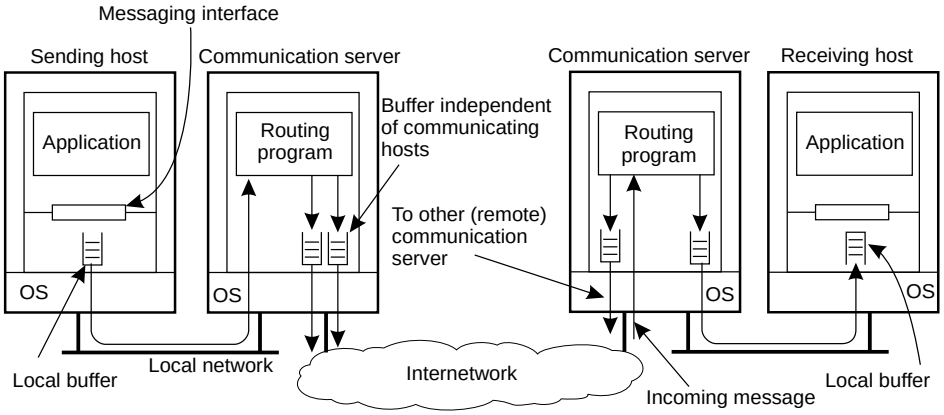


Figure 1: A communication system.

Let us now examine the steps involved in a `send`:

1. [This step may not exist, if the message is sent directly from the user space.] Copy the message to the kernel space. After this step, the sender process' message buffer, if it exists, can be reused.
2. Transmit the message. If step 1 doesn't exist, after this step the sender process' message buffer can be reused.
3. Message received by the destination kernel. The destination kernel may return an ack.
4. The message was **delivered** to the destination process, which called `receive`.

Normally, a `send` is called non-blocking if it returns before step 2; it is called blocking if it returns after step 2.

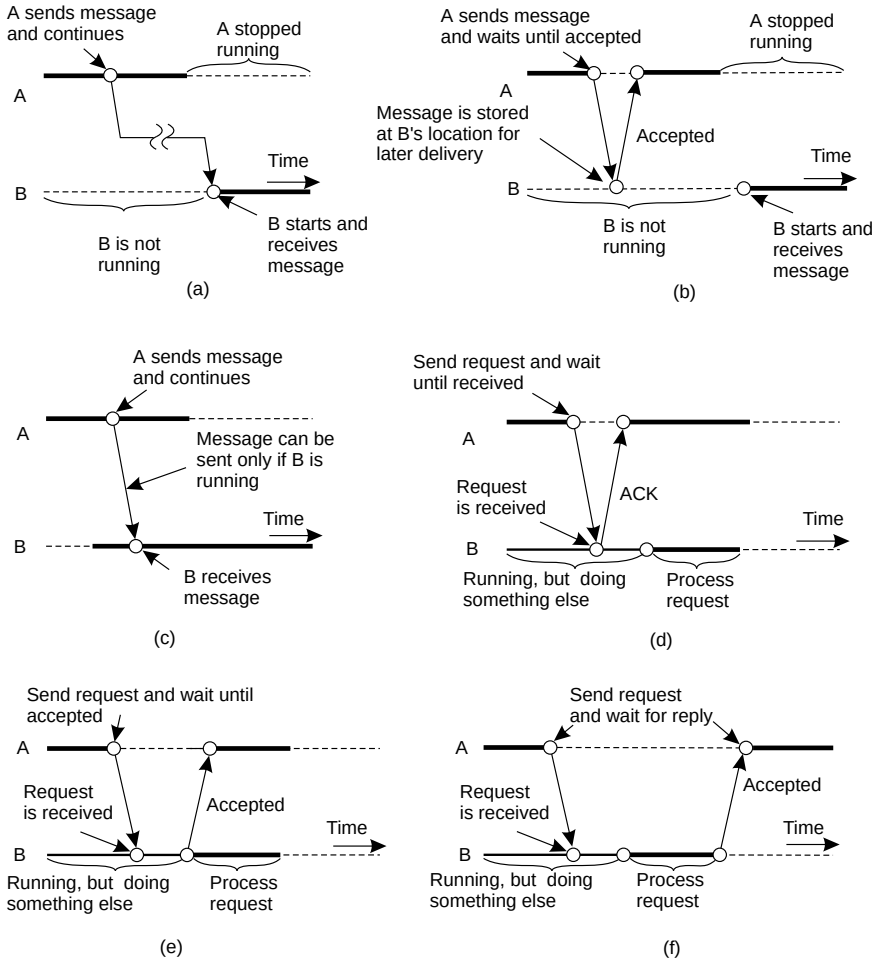


Figure 2: Six different forms of communication: (a) Persistent asynchronous, (b) Persistent synchronous, (c) Transient asynchronous, (d) Transient synchronous (receipt-based), (e) Transient synchronous (delivery-based), and (f) Transient synchronous (response-based)

Comments

- There is no wide agreement as to exactly when a blocking `send` should return: different systems adopt different definitions. It may block until the message has been (a) transmitted, or (b) received. (a) is preferred by OS designers, who are concerned with message buffer management (e.g., when it can be reused), while (b) is preferred by programming language designers, who are interested in information exchange between communicating processes.
- Non-blocking `receive` can simulate its blocking counterpart (by busy wait), but not vice versa.

- Non-blocking `send` and blocking `receive` can, together with a mailbox, simulate a semaphore. (Think about how it can be done.)
- A communication scheme is said to be **synchronous**, if the sender blocks until the message has been "received" by the receiving side (remote kernel or application). Synchronous communication means that the sender and receiver get 'in sync'; one communication partner must wait for the other to catch up, so that they reach a well-defined point before proceeding. (See Text p. 128.) □

Practical Implementation

1. (†)In the non-blocking `send`, when the message has been successfully transmitted, the calling program may be informed by an interrupt,[2] upon which the user message area may be reused. However, user-level interrupts are difficult to program. Moreover, program execution is timing-dependent, making debugging extremely difficult. Thus, this option is not recommended.
2. Similarly, an interrupt informs the receiver process of the completion of non-blocking `receive`. A `test` primitive may also be provided for the receiver process to find out if a msg has been placed in the user buffer.
3. A `send` can also block or return an error on "full buffer", since the OS typically provides only finite space for message buffering. In some OS (e.g., BSD Unix), if no system buffer for *sending* is available, non-blocking `send` returns –1 and proceeds.
4. With blocking versions, **timeout** option is available in some systems, e.g., *Mach, Solaris*. This is especially important for inter-machine communication, due to failures of communication or remote machine.
5. If the destination is on the same machine, **send by reference**, together with **copy-on-write** (see Text p. 219) is very efficient. □

The major advantage of blocking `send` over the non-blocking counterpart is its ease and lower overhead in implementation (e.g., no queueing of messages required at the sender, or no need to copy to the kernel space). However, with blocking `send`, there is less concurrency, i.e., the thread cannot execute until transmission is complete.

The non-blocking `send` primitive is more efficient, "immediately" returning control to the sender thread, allowing it to proceed to the execution of the steps after the `send`, in parallel with the message transmission (I/O). However, this implies queueing, since there may be many messages that have been "sent" by the sending threads but not actually transmitted.

Implementation example:

To implement (intra-machine) blocking `send`(*destination, &msg*):

[2]E.g., in most versions of UNIX, the kernel sends the **signal** SIGIO to the process.

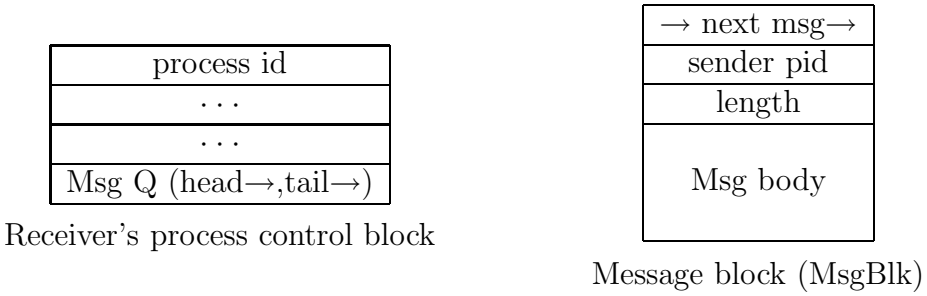


Figure 3: Data structures for messages.

`if` *receiver* not known
  {return "*receiver* unknown"}
`else` call *memoryallocate* to get system space for MsgBlk;
copy sender processid, msg length, body, into MsgBlk;
add MsgBlk at the tail of the receiver's msg queue;

In this implementation, as soon as a `send` call returns the sender's buffer can be reused.

Implementation issues

- **Copying**

  Entails some overhead (time and space). In some implementations, when the sender calls a non-blocking `send`, the contents of its message buffer is copied from the sender's address space to system MsgBlk; so that, when the call returns, the sender can modify its message buffer. When the receiver calls `receive`, the contents of system MsgBlk is copied to the receiver's address space.

  For local messages, where the sender and receiver are in the same machine, there is a trade-off between efficiency and safety. The most efficient implementation (e.g., *Mach*) will pass the pointer (first to the kernel and then) to the receiver (**send-by-reference**).[3] Copy-on-write can be used if the sender wants to modify the message.

- **Mailbox and ports**

  Mailbox is a message queue to/from which messages can be sent/received by several processes. It has a certain size and when it's full, the sender blocks on `send`.

  In a distributed environment, the implementation of a mailbox can be quite costly, since the receivers may reside on different computers (where do you maintain the received message queue?). Thus, a limited form of a mailbox, called a **port**, which is associated with only one receiver, is often used (e.g., *Mach*). The term **port** is used to mean many different things, including a hardware port and logical communications port. The port number may be unique within a host as in Unix or a process as in *Mach* (a process in *Mach* is called a *task*).

[3]See the section on *Mach* later in this chapter.

2.3 *Mach* Messages

[Textbook Chap. 18]
`http://cs-www.cs.yale.edu/homes/avi/os-book/osc/mach.pdf`

2.3.1 Overview *Mach* introduced many innovative ideas in IPC. It is commented that *"In* Mach*, IPC is the central and most important kernel component. Instead of the OS supporting IPC mechanisms,* Mach *provides an IPC facility that supports much of the OS."*

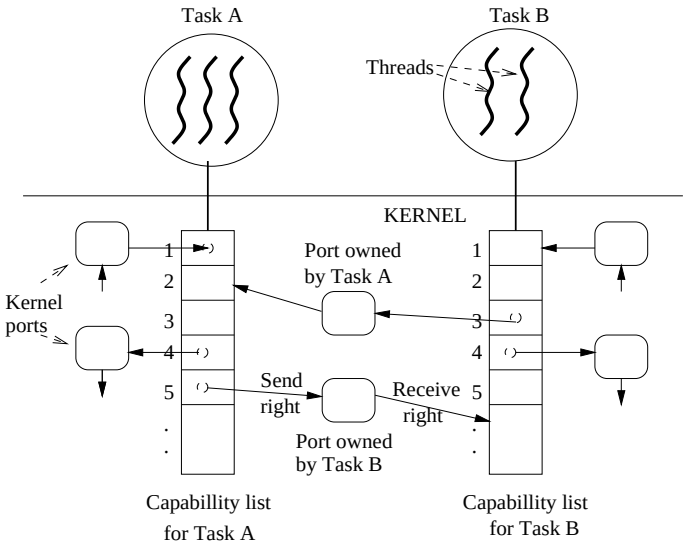


Figure 4: *Mach* structure.

In *Mach* a **task** is the basic unit of resource allocation and protection. It mainly consists of an address space, holding programs and data, and **threads** executing in the address space. (See Fig. 4.) So, a Unix process is like a task with a single thread. All threads in a task have equal privileges, sharing all resources of the task. A **port** provides the basic object-reference mechanism in *Mach*. Messages are always sent to a port and received via a port. All threads in a task have the same access rights (send-only, or send-and-receive) to ports. Only one task, i.e., the **owner** task, has a send-and-receive right to a port.[4]

A port is implemented by a data structure, containing information such as its owner task, a pointer to the (bounded) queue of messages addressed to it, a pointer to the queue of threads blocked on this port, etc. On creation, each task or thread is allocated a few **kernel ports**, including a `task_self` port or a `thread_self` port, **bootstrap** port, **exception** port, etc., which are owned by the kernel, and the `task_notify` and **reply** ports to which the task has receive rights. They are introduced to reduce the system calls to a bare minimum, i.e., most system calls and their replies are sent by means of these ports. For example, a thread can create a new port by sending a message through its `thread_self` port, and a thread can be notified of an exception via the exception port. The messages sent to a kernel port affect

[4]When we say that a task has a send or receive right to a port, it means that all threads in it have a send or receive right to it.

the corresponding task or thread. For example, `task_terminate(`*task*`)` sent to the kernel port *task* destroys the named task (and all threads in it).

The bootstrap port provides access to a **name server**, through which a task can obtain the send rights to the ports of publicly available servers. (The servers register themselves with the name server.)

Each task has a **capability list**, listing the set of ports accessible to the threads in the task. (See Fig. 4.) In order to find its `thread_self` port *number*, indicating its position in the capability list, the executing thread can call `thread_self()`. (`task_self()` returns the task port of the task to which the thread belongs). The kernel port of one thread can be accessed by another thread in the same task. If thread T_1 makes a kernel call to the `thread_self` port of T_2, it is T_2 that is affected by the call.

Each task has its own port name space used for port names (small integers). When any thread creates a port, the same rights to it are shared by all the threads in the task. Thereafter, any thread within the task can deallocate any of these rights, or transfer them to other tasks.

Port and port set:

Messages are always sent to a **port** and received from a port or a **port set**. A port set is a group of ports. A port may be a member of at most one port set at any time.

A port set comes in handy, if you want to use one thread to service requests coming in on multiple ports (say, for multiple objects), without having to dedicate one thread to each. If a thread receives from a port set, a message sent to any of the ports in the set will be returned. A message can be sent to a port, but not to a port set.

A *Mach* kernel call

`port_allocate`(*thread_self*(), &*myport*)

creates a new port for the calling thread and returns the port number in *myport*. The task to which the calling thread belongs becomes the owner of the port, possessing both the receive and send rights to it. Any thread within the task can deallocate any of the port access rights possessed by the task, or transfer them to other tasks.[5] Send and receive rights can be given to other tasks by

`port_insert_send`(*task*, *myport*, *its_name*),

and

`port_insert_receive`(*task*, *myport*, *its_name*)

respectively, where *its_name* is the "name" (i.e., a number) by which the receiving *task* will know the new port.[6]

2.3.2 Implementation

A port is implemented as a data structure (protected bounded queue), consisting of several fields including a pointer to the message queue addressed to this port, the current

[5] During the time between sending and receiving the transferred rights, the kernel holds the rights, and any messages sent to the port will be queued on the port.

[6] If *task* already has a port named *its_name*, or has some other name for *myport*, the call will return an error code. If 0 is specified as *its_name*, the kernel finds an appropriate name.

message count, a pointer to the port's owner task, a pointer to the list of threads blocked on this port, etc.

Intra-machine out-of-line message delivery: [Coulouris §18.6.3]

As we saw earlier, many systems copy messages from sender's address space to receiver's address space, often via a system buffer in the kernel space. The reason for using a system buffer is to enable the sender to reuse its msg buffer as soon as the `send` call returns (non-blocking `send`). This is inefficient especially for large messages. In sending an **out-of-line** message, *Mach* modifies the page table of the receiving task to include the pages of the message.[7] The message body will contain a pointer to the region containing the message. The sender's pages containing the message are now tagged as **copy-on-write** in order not to destroy the original contents of the message. Message passing is thus implemented via virtual memory management. There is actually an intermediate stage to this operation: The kernel first maps that region of the sender's memory containing the message into its own space, and sets the sender's memory map to copy-on-write mode. (See Fig. 5.) In copy-on-write mode, any attempt to write into the page will cause a protection fault.[8] The interrupt is used to create a new copy to be used by the writer, keeping the original intact.

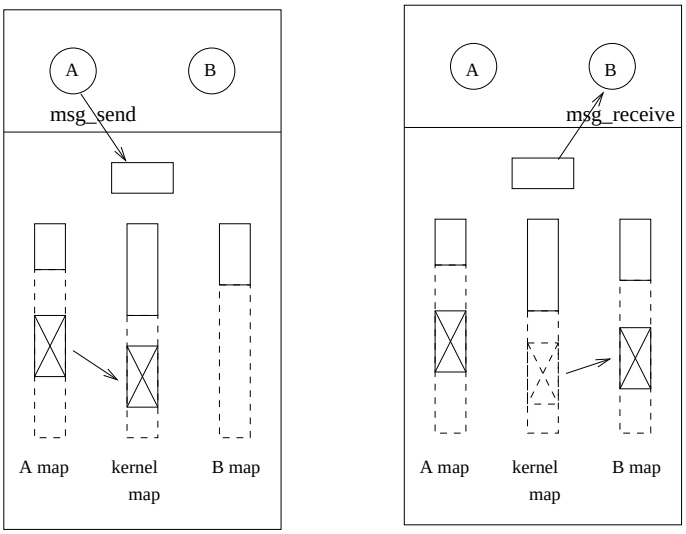


Figure 5: Out-of-line message transfer: copy-on-write mapping first to kernel then to the receiver task B, when a thread in B invokes receive.

Inter-machine message delivery: [Coulouris §18.5.1]

In Fig. 6, in order to send a message to a remote port, *RP*, a thread on machine *A* sends a message to a **proxy port**, *PP*, to which it has a send right. The kernel on *A* forwards the message to the owner of *PP*, the *NetMsg Server* (NMS; a user-level daemon) on *A*. As

[7]Messages should be aligned on page boundaries. Otherwise, the receiver may see part of a page which contains other (non-message) info.

[8]Incidentally, this is also used in implementing UNIX `fork` in *Mach*; immediately after a fork, the child process shares the parent's memory space in copy-on-write mode.

explained later in the discussion of the *Network Name Server*, this proxy port was created when the same or another thread in the same task was given the send right to access the remote port, *RP*. A thread on the NMS is constantly listening to this port as well as other proxy ports in the same port set. Upon receiving a message on a proxy port, the NMS consults its database,[9] finds the global **network port** corresponding to the proxy port and the fact that the destination port, *RP*, is on machine B. It then passes the message on to the NMS on machine B.[10] This NMS translates the network port to the corresponding local port name on B, and sends the message to the kernel with the local port name.

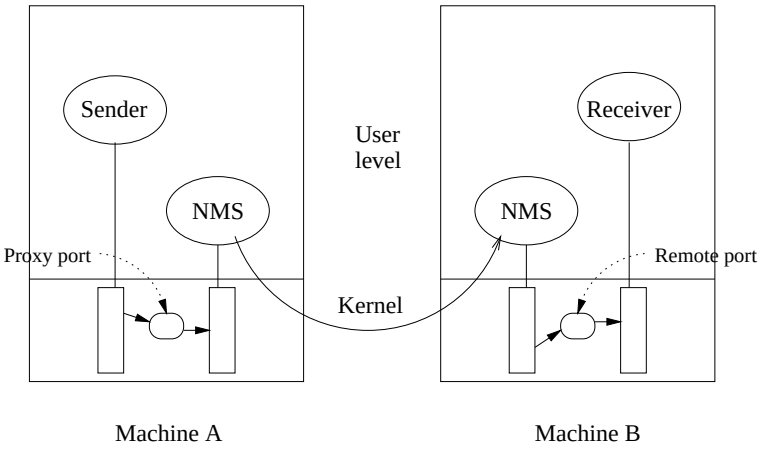


Figure 6: Message transfer between machines.

Finally the kernel on B delivers the message to a thread with a receive right to the port, when the thread executes a `msg_receive` call. *NMS*s communicate with each other by means of an appropriate communication protocol agreed-upon by the two communicating parties.

Inter-machine out-of-line message delivery:

Copy-on-reference. (Not implemented) The sender's message is first mapped to the address space of the NMS in the sender machine. The data field descriptor in the message indicates that out-of-line data is involved, and shows the data field type and size. Access by receiving task's threads cause page faults: these page faults are passed by the kernel to the sender's NMS, which, acting as a remote page server, retrieves the pages and transmits them.

In the current implementation, if any out-of-line data is involved, the NMS converts it into an in-line message before sending it to a remote machine.

2.4 Socket

[9]NMS provides a primitive name service. The network ports are assigned globally unique numbers. Tasks can register their ports with NMS for look-up by other tasks. NMSs communicate with each other using RPCs for "add port", "remove port", and "look up".

[10]If the machines A and B are of different types, the conversion of data representations may be required. To facilitate this, each data field in the message body has an associated field descriptor, which indicates the data type (integer, character, real, capability, etc).

The BSD Unix introduced the **socket** mechanism to enable communication between processes on different machines, as well as unrelated processes on the same machine. It is a service access point (SAP) to the TCP/IP transport service. The sockets must be **bound** to **port**s. In this section, we give examples for Java and Unix socket programming, in terms of client-server communication using streams (TCP). Note that although a TCP connection is specified by a 4-tuple, `(host_IP_address, local_port_number, remote_IP_address, local_port_number)`, while a UDP socket is uniquely identified by a 2-tuple, `(host_IP_address, local_port_number)`. It is possible that several TCP connections share the same first two components of the 4-tuple, provided, of course, the remote sockets are different.[11]

For use of sockets using datagrams (UDP), the reader is referred to Text §4.2.3.

2.4.1 Java Socket

For examples of Java socket programming, see Figures 4.3–4.6 of the Textbook. The following program segment is the class definition for `ToDServer` from Assignment 1, which is similar to `TCPServer` of Figure 4.6:

```
import java.net.*; //Defines socket-related classes
import java.io.*;  //Defines stream-related classes

public class ToDServer {
   public static void main(String args[]) {
      try {
        ServerSocket listenSocket = new ServerSocket(5555); //5555 is a port number
        while ( true ) {
           Socket clientSocket = listenSocket.accept(); //Wait for and accept a request
           Connection c = new Connection(clientSocket); //Create a Connection instance
           c.start();    //This replaces this.start in the original Connection class.
        }
      } catch(IOException e) {System.out.println("Listen:"+e.getMessage());}
   }
}
```

Suppose we replace the `Connection` method of Text Figure 4.6 by the one given below. Then, instead of the client program (Figure 4.5), you can use the following `telnet` command to interact with the TCP server and get the Date and Time:

```
telnet 127.0.0.1 5555
```

(The IP address 127.0.0.1 means the local machine.)

```
public class Connection extends Thread
```

[11] A http server, for example, uses a new TCP socket with the same port# as itself, i.e., 80, to communicate with each new client. To find the active sockets, just type command `netstat -a`, which is available in both Unix and MSDOS. It will print a list of local and remote port numbers associated with the active sockets.

```
{
   private Socket clientSocket;
   private PrintWriter  pOut;  //This class allows ordinary file IO over the socket
   public Connection(Socket aClientSocket) {  //This is the constructor.
      clientSocket = aClientSocket;
   }

   public void run() {
      try {
           // Create a new PrintWriter with automatic flushing;
           // getOutputStream returns an OutputStream object
         pOut = new PrintWriter(clientSocket.getOutputStream(), true);
           // now send a message to the client
         pOut.println("The Date and Time is " + new java.util.Date().toString());
         clientSocket.close();
      }
     catch (java.io.IOException e) {System.out.println("Connection:"+e.getMessage());}
   }
}
```

Notes: Remote information for the returned `ClientSocket`, for example, can be found by `ClientSocket.getHostName` and `ClientSocket.getPort`. The local port can be found by `ClientSocket.getLocalPort`. In the above porgram `ClientSocket.getLocalPort` will return 5555. For the server socket, the remote port#=0. □

2.4.2 BSD Unix Socket[12]

Here I give a very brief summary of Unix sockets, since most of you are already familiar with them.

Server	**Client**
1. `s_listen = socket()`	1. `s_req =socket()`
2. `bind(s_listen,`*serverAddr*`)`	2. `connect(s_req,`*serverAddr*`)`
3. `listen(s_listen,`*backlog*`)`	
Repeat {	
4. `s_serv=accept(s_listen)`	
5. `fork` a thread	
}	3. `write(s_req)`
Within a thread {	
6. `read(s_serv)`	
7. `write(s_serv)`	4. `read(s_req)`
}	5. `close(s_req)`

[12]See the Unix manual entry for `socket`(2).

Note that, for simplicity, not all arguments of the calls are shown. In particular `socket()` takes three parameters as in

$$\texttt{s\_listen} = \texttt{socket}(\texttt{PF\_INET}, \texttt{SOCK\_STREAM}, 0),$$

where the protocol (or address) family, Internet, and the STREAM type (TCP) are specified.[13] The `bind` call binds a socket to a unique address which consists of a 32-bit host IP address and a 16-bit port number. In Java `ServerSocket` this binding is implicit. The server uses the `listen` system call to tell the kernel to set up a queue of up to *backlog* connection requests from clients. In Java, listening is also implicit. The server's `accept` call is equivalent to Java's `Socket.accept()`. As in Java, a client must create a socket to send a request to a server. It needs to connect to the server's address by `connect`.[14] The server's `accept` call returns a new socket descriptor (`s_serv` in the above table) corresponding to the new connection with a client. (Both `connect` and `accept` are blocking by default. A socket can made non-blocking, in which case these calls return with an error indication, if they cannot be immediately executed.) The original socket descriptor (`s_listen`) is still open for further listening. The server would use `fork` to create a new thread to service each request.

In the Internet port numbers less than 1024 are reserved as **well-known ports**, for well-known services such as `http` (port 80), `ftp` (port 21) and `telnet` (port 23).

[13] `SOCK_DGRAM` specifies a socket for connectionless datagram service (UDP), which is accessed by `sendto` and `recvfrom` calls, instead of `write` and `read`.

[14] The client need not explicitly `bind` a specific name (address) to his/her socket; binding takes place automatically as a side effect of `connect`. To find the name assigned to the socket, use `getsockname`. `connect` returns with an error indication, ECONNREFUSED, when there is already max number (*backlog*) of `connect` requests pending.