Chapter 12

Interprocess Communication

Recall that an important function of operating systems is to provide abstractions and services to applications. One such service is to support communication among processes, in order to enable the construction of concurrent or distributed applications. A special case is client-server applications, which allow client applications to interact with server applications using well-defined interfaces.

This chapter discusses high-level issues in communication: naming, abstractions and programming interfaces, and application structures such as client-server. The next chapter deals with the low-level details of how bytes are actually transferred and delivered to the right place.

To read more: Communication is covered very nicely in Stallings [6], Chapter 13, especially Sections 13.1 and 13.2. It is also covered in Silberschatz and Galvin [4] Sections 15.5 and 15.6. Then, of course, there are whole textbooks devoted to computer communications. A broad introductory text is Comer [1]; more advanced books are tanenbaum [9] and stallings [5].

12.1 Naming

In order to communicate, processes need to know about each other

We get names when we are born, and exchange them when we meet other people. What about processes?

A basic problem with inter-process communication is *naming*. In general, processes do not know about each other. Indeed, one of the roles of the operating system is to keep processes separate so that they do not interfere with each other. Thus special mechanisms are required in order to enable processes to establish contact.

Inheritence can be used in lieu of naming

The simplest mechanism is through family relationships. If one process forks another, as in the Unix system, the child process may inherit various stuff from its parent. For example, various communication mechanisms may be established by the parent process before the fork, and are thereafter accessible also by the child process. One such mechanism is pipes, as described below on page 214 and in Appendix D.

Exercise 168 *Can a process obtain the identity (that is, process ID) of its family members? Does it help for communication?*

Predefined names can be adopted

Another simple approach is to use predefined and agreed names for certain services. In this case the name is known in advance, and represents a service, not a specific process. The process that should implement this service adopts the name as part of its initialization.

Exercise 169 *What happens if no process adopts the name of a desired service?*

For example, this approach is the basis for the world-wide web. The service provided by web servers is actually a service of sending the pages requested by clients (the web browsers). This service is identified by the port number 80 — in essense, the port number serves as a name. This means that a browser that wants a web page from the server `www.abc.com` just sends a request to port 80 at that address. The process that implements the service on that host listens for incoming requests on that port, and serves them when they arrive. This is described in more detail below.

Names can be registered with a name server

A more general solution is to use a *name service*, maintained by the operating system. Any process can advertise itself by registering a chosen string with the name service. Other processes can then look up this string, and obtain contact information for the process that registered it.

Exercise 170 *And how do we create the initial contact with the name service?*

A sticky situation develops if more than one set of processes tries to use the same string to identify themselves to each other. It is easy for the first process to figure out that the desired string is already in use by someone else, and therefore another string should be chosen in its place. Of course, it then has to tell its colleagues about the new string, so that they know what to request from the name server. But it can't contact its colleagues, because the whole point of having a string was to establish the initial contact...

12.2 Programming Interfaces and Abstractions

Being able to name your partner is just a pre-requisite. The main issue in communication is being able to exchange information. Processes cannot do so directly — they need to ask the operating system to do it for them. This section describes various interfaces that can be used for this purpose.

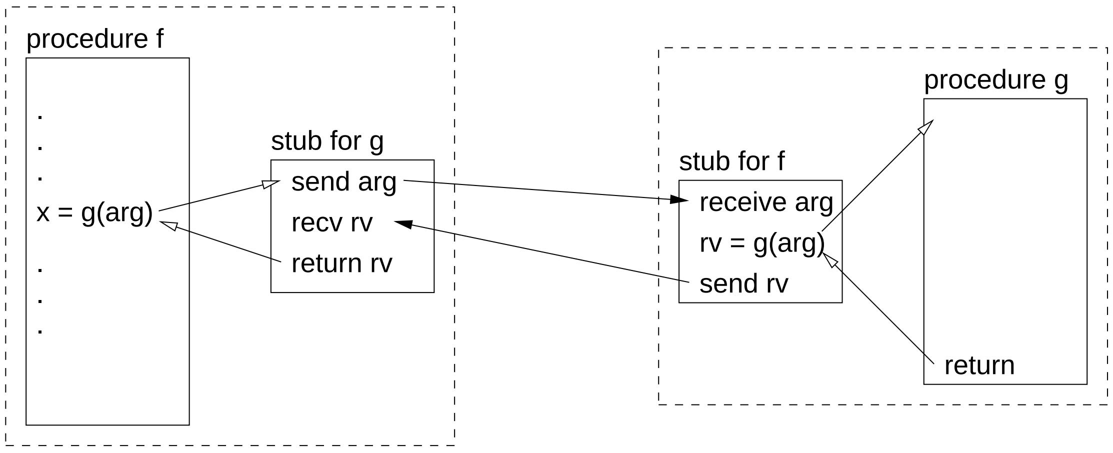
12.2.1 Remote Procedure Call

A natural extension to procedure calls is to call remote procedures

The most structured approach to communication is to extend the well-known procedure calling mechanism, and allow one process to call a procedure from another process, possibly on a different machine. This is called a *remote procedure call* (RPC). This idea has become even more popular with the advent of object-oriented programming, and has even been incorporated in programming languages. An example is Java's remote method invocation (RMI). However, the idea can be implemented even if the program is not written in a language with explicit support.

The implementation is based on stub procedures

The implementation is based on stubs — crippled procedures that represent a remote procedure by having the same interface. The calling process is linked with a stub that has the same interface as the called procedure. However, this stub does not *implement* this procedure. Instead, it sends the arguments over to the stub linked with the other process, and waits for a reply.



The other stub mimics the caller, and calls the desired procedure locally with the specified arguments. When the procedure returns, the return values are shipped back and handed over to the calling process.

RPC is a natural extension of the procedure call interface, and has the advantage of allowing the programmer to concentrate on the logical structure of the program, while disregarding communication issues. The stub functions are typically provided by a library, which hides the actual implementation.

For example, consider an ATM used to dispense cash at a mall. When a user requests cash, the business logic implies calling a function that verifies that the account balance is sufficient and then updates it according to the withdrawal amount. But such a function can only run on the bank's central computer, which has direct access to the database that contains the relevant account data. Using RPC, the ATM software can be written as if it also ran directly on the bank's central computer. The technical issues involved in actually doing the required communication are encapsulated in the stub functions.

Exercise 171 *Can any C function be used by RPC?*

12.2.2 Message Passing

RPC organizes communication into structured and predefined pairs: send a set of arguments, and receive a return value. Message passing allows more flexibility by allowing arbitrary interaction patterns. For example, you can send multiple times, without waiting for a reply.

Messages are chunks of data

On the other hand, message passing retains the partitioning of the data into "chunks" — the messages. There are two main operations on messages:

`send(to, msg, sz)` — Send a message of a given size to the addressed recipient. `msg` is a pointer to a memroy buffer that contains the data to send, and `sz` is it's size.

`receive(from, msg, sz)` — Receive a message, possibly only from a specific sender. The arguments are typically passed by reference. `from` may name a specific sender. Alternatively, it may contain a "dontcare" value, which is then overwritten by the ID of the actual sender of the received message. `msg` is a pointer to a memory buffer where the data is to be stored, and `sz` is the size of the buffer. This limits the maximal message size that can be received, and longer messages will be truncated.

Exercise 172 *Should the arguments to `send` be passed by reference or by value?*

Receiving a message can be problematic if you don't know in advance what it's size will be. A common solution is to decree a maximal size that may be sent; recipients then always prepare a buffer of this size, and can therefore receive any message.

Exercise 173 *Is it possible to devise a system that can handle messages of arbitrary size?*

Sending and receiving are discrete operations, applied to a specific message. This allows for special features, such as dropping a message and proceeding directly to the next one. It is also the basis for collective operations, in which a whole set of processes participate (e.g. broadcast, where one process sends information to a whole set of other processes). These features make message passing attractive for communication among the processes of parallel applications. It is not used much in networked settings.

12.2.3 Streams: Unix Pipes, FIFOs, and Sockets

Streams just pass the data piped through them in sequential, FIFO order. The distinction from message passing is that they do not maintain the structure in which the data was created. This idea enjoys widespread popularity, and has been embodied in several mechanisms.

Streams are similar to files

One of the reasons for the popularity of streams is that their use is so similar to the use of sequential files. You can write to them, and the data gets accumulated in the order you wrote it. You can read from them, and always get the next data element after where you dropped off last time.

Exercise 174 *Make a list of all the attributes of files. Which would you expect to describe streams as well?*

Pipes are FIFO files with special semantics

Once the objective is defined as inter-process communication, special types of files can be created. A good example is Unix *pipes*.

A pipe is a special file with FIFO semantics: it can be written sequentially (by one process) and read sequentially (by another). There is no option to seek to the middle, as is possible with regular files. In addition, the operating system provides some special related services, such as

- If the process writing to the pipe is much faster than the process reading from it, data will accumulate unnecessarily. If this happens, the operating system blocks the writing process to slow it down.
- If a process tries to read from an empty pipe, it is blocked rather then getting an EOF.
- If a process tries to write to a pipe that no process can read (because the read end was closed), the process gets a signal. On the other hand, when a process tries to read from a pipe that no process can write, it gets an EOF.

Exercise 175 *How can these features be implemented efficiently?*

The problem with pipes is that they are unnamed: they are created by the `pipe` system call, and can then be shared with other processes created by a fork. They cannot be shared by unrelated processes. This gap is filled by FIFOs, which are essentially named pipes. This is a special type of file, and appears in the file name space.

To read more: See the man page for `pipe`. The mechanics of stringing processes together are described in detail in Appendix D. Named pipes are created by the `mknode` system call, or by the `mkfifo` shell utility.

In the original Unix system, pipes were implemented using the file system infrastructure, and in particular, inodes. In modern systems they are implemented as a pair of sockets, which are obtained using the `socketpair` system call.

Sockets support client-server computing

A much more general mechanism for creating stream connections among unrelated processes, even on different systems, is provided by *sockets*. Among other things, sockets can support any of a wide variety of communication protocols. The most commonly used are the internet protocols, TCP/IP (which provides a reliable stream of bytes), and UDP/IP (which provides unreliable datagrams).

The way to set up a connection is asymmetric. The first process, which is called the *server*, does the following (using a somewhat simplified API).

`fd=socket()` First, it *creates a socket*. This means that the operating system allocates a data structure to keep all the information about this communication channel, and gives the process a file descriptor to serve as a handle to it.

`bind(fd, port)` The server then *binds* this socket (as identified by the file descriptor) to a port number. In effect, this gives the socket a name that can be used by clients: the machine's IP (internet) address together with the port are used to identify this socket (you can think of the IP address as a street address, and the port number as a door or suite number at that address). Common services have predefined port numbers that are well-known to all (to be described in Section 12.4). For other distributed applications, the port number is typically selected by the programmer.

`listen(fd)` To complete the setup, the server then *listens* to this socket. This notifies the system that communication requests are expected.

The other process, called the *client*, does the following.

`fd=socket()` First, it also creates a socket.

`connect(fd, addr, port)` It then *connects* this socket to the server's socket, by giving the server's IP address and port. This means that the server's address and port are listed in the local socket data structure, and that a message regarding the communication request is sent to the server. The system on the server side finds the server's socket by searching according to the port number.

To actually establish a connection, the server has to take an additional step:

`newfd=accept(fd)` After the `listen` call, the original socket is waiting for connections. When a client connects, the server needs to `accept` this connection. This creates a *new* socket that is accessed via a new file descriptor. This new socket (on the server's side) and the client's socket are now connected, and data can be written to and read from them in both directions.

After a connection is accepted the asymmetry of setting up the connection is forgotten, and both processes now have equal standing. However, the original socket created by the server still exists, and it may accept additional connections from other clients. This is motivated below in Section 12.3.

Exercise 176 *What can go wrong in this process? What happens if it does?*

Concurrency and asynchrony make things hard to anticipate

Distrubuted systems and applications are naturally concurrent and asynchronous: many different things happen at about the same time, in an uncoordinated manner. Thus a process typically cannot know what will happen next. For example, a process may have opened connections with several other processes, but cannot know which of them will send it some data first.

The `select` system call is designed to help with such situations. This system call receives a set of file descriptors as an argument. It then blocks the calling process until any of the sockets represented by these file descriptors has data that can be read. Alternatively, a timeout may be set; if no data arrives by the timeout, the `select` will return with a failed status.

On the other hand, using streams does provide a certain built-in synchronization: due to the FIFO semantics, data cannot be read before it is written. Thus a process may safely try to read data from a pipe or socket. If no data is yet available, the process will either block waiting for data to become available or will receive an error notification — based on the precise semantics of the stream.

To read more: See the man pages for `socket`, `bind`, `connect`, `listen`, `accept`, and `select`.

12.2.4 Shared Memory

Shared access to the same memory is the least structured approach to communication. There are no restrictions on how the communicating processes behave. In particular, there is no a-priori guarantee that one process write the data before another attempts to read it.

Within the same system, processes can communicate using shared memory

Recall that a major part of the state of a process is its memory. If this is shared among a number of processes, they actually operate on the same data, and thereby communicate with each other.

Rather than sharing the whole memory, it is possible to only share selected regions. For example, the Unix shared memory system calls include provisions for

- Registering a name for a shared memory region of a certain size.
- Mapping a named region of shared memory into the address space.

The system call that maps the region returns a pointer to it, that can then be used to access it. Note that it may be mapped at different addresses in different processes.

To read more: See the man pages for `shmget` and `shmat`.

Exercise 177 *How would you implement such areas of shared memory? Hint: think about integration with the structures used to implement virtual memory.*

In some systems, it may also be possible to inherit memory across a fork. Such memory regions become shared between the parent and child processes.

distributed shared memory may span multiple machines

The abstraction of shared memory may be supported by the operating system even if the communicating processes run on distinct machines, and the hardware does not provide them direct access to each other's memory. This is called *distributed shared memory* (DSM).

The implementation of DSM hinges on playing tricks with the mechanisms of virtual memory. Recall that the page table includes a present bit, which is used by the operating system to indicate that a page is indeed present and mapped to a frame of physical memory. If a process tries to access a page that is *not* present, a page fault is generated. Normally this causes the operating system to bring the page in from permanent storage on a disk. In DSM, it is used to request the page from the operating system on another machine, where it is being used by another process. Thus DSM is only possible on a set of machines running the same system type, and leads to a strong coupling between them.

While the basic idea behind the implementation of DSM is simple, getting it to perform well is not. If only one copy of each page is kept in the system, it will have to move back and forth between the machines running processes that access it. This will happen even if they actually access different data structures that happen to be mapped to the same page, a situation known as *false sharing*. Various mechanisms have been devised to reduce such harmful effects, including

- Allowing multiple copies of pages that are only being read.
- Basing the system on objects or data structures rather than on pages.
- Partitioning pages into sub-pages and moving them independently of each other.

Exercise 178 *Is there a way to support multiple copies of pages that are also written, that will work correctly when the only problem is actually false sharing?*

To read more: A thorough discussion of DSM is provided by Tanenbaum [8], chapter 6. An interesting advanced system which solves the granularity problem is MilliPage [3].

Sharing memory leads to concurrent programming

The problem with shared memory is that its use requires synchronization mechanisms, just like the shared kernel data structures we discussed in Section 4.1. However, this time it is the user's problem. The operating system only has to provide the means by which the user will be able to coordinate the different processes. Many systems therefore provide semaphores as an added service to user processes.

Exercise 179 *Can user processes create a semaphore themselves, in a way that will block them if they cannot gain access? Hint: think about using pipes. This solution is good only among related processes within a single system.*

Files also provide a shared data space

An alternative to shared memory, that is supported with no extra effort, is to use the file system. Actually, files are in effect a shared data repository, just like shared memory. Moreover, they are persistent, so processes can communicate without overlapping in time. However, the performance characteristics are quite different — file access is much slower than shared memory.

Exercise 180 *Is it possible to use files for shared memory without suffering a disk-related performance penalty?*

12.3 Distributed System Structures

Using the interfaces described above, it is possible to create two basic types of distributed systems or applications: symmetrical or asymmetrical.

Symmetrical communications imply peer to peer relations

In symmetric applications or systems, all processes are peers. This is often the case for processes communicating via shared memory or pipes, and is the rule in parallel applications.

Peer-to-peer systems, such as those used for file sharing, are symmetrical in a different sense: in such systems, all nodes act both as clients and as servers. Therefore some of the differences between clients and servers described below do not hold.

Client-server communications are asymmetrical

The more common approach, however, is to use an asymmetrical structure: one process is designated as a server, and the other is its client. This is used to structure the application, and often matches the true relationship between the processes. Examples include

- A program running on a workstation is the client of a file server, and requests it to perform operations on files.
- A program with a graphical user interface running on a workstation is a client of the X server running on that workstation. The X server draws things on the screen for it, and notifies it when input is events have occured in its window.
- A web browser is a client of a web server, and asks it for certain web pages.
- An ATM is a client of a bank's central computer, and asks it for authorization and recording of a transaction.

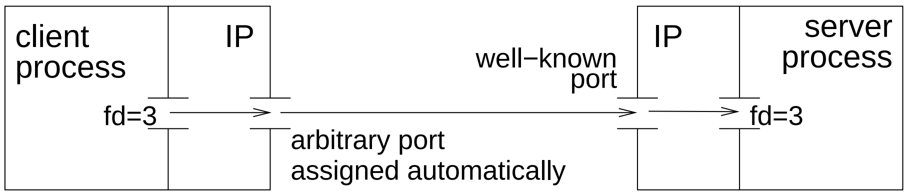
Client-server interactions can be programmed using any of the interfaces described above, but are especially convenient using RPC or sockets.

Servers typically outlive their clients

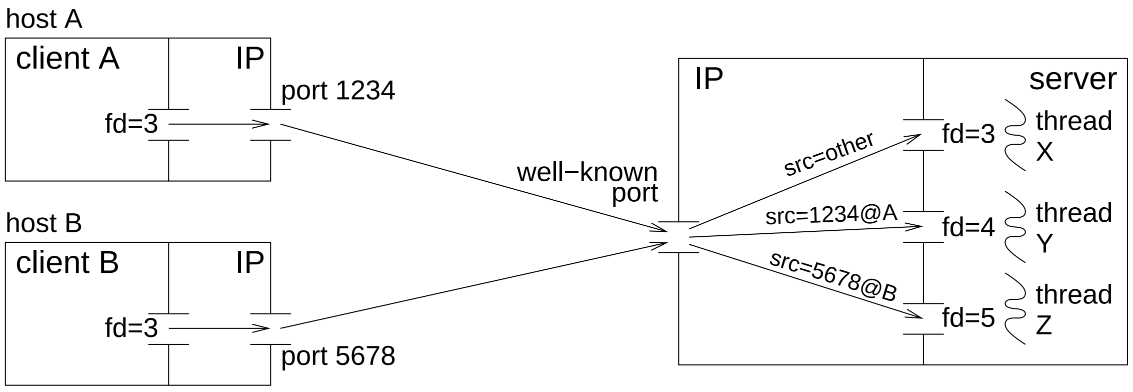
An interesting distinction between peer-to-peer communication and client-server communication is based on the temporal dimension: In peer-to-peer systems, all processes may come and go individually, or in some cases, they all need to be there for the interaction to take place. In client-server systems, on the other hand, the server often exists for as long as the system is up, while clients come and go.

The implications of this situation are twofold. First, the server cannot anticipate which clients will contact it and when. As a consequence, it is futile for the server to try and establish contact with clients; rather, it is up to the clients to contact the server. The server just has to provide a means for clients to find it, be it by registering in a name service or by listening on a socket bound to a well-known port.

Second, the server must be ready to accept additional requests from clients at any moment. This is the reason for the `accept` mentioned above. Before calling `accept`, the incoming request from the client ties up the socket bound to the well-know port, which is the server's advertised entry point.



By calling `accept`, the server re-routes the incoming connection to a new socket represented by another file descriptor. This leaves the original socket free to receive additional clients, while the current one is being handled. Moreover, if multiple clients arrive, each will have a separate socket, allowing for unambiguous communication with each one of them. The server will often also create a new thread to handle the interaction with each client, to furhter encapsulate it.



Note that the distinction among connections is done by the IP addresses and port numbers of the two endpoints. All the different sockets created by `accept` share the same port on the server side. But they have different clients, and this is indicated in each incoming communication. Communications coming from an unknown source are routed to the original socket.

12.4 Example Client-Server Systems

To read more: Writing client-server applications is covered in length in several books on TCP/IP programming, e.g. Commer [2].

Many daemons are just server processes

Unix daemons are server processes that operate in the background. They are used to provide various system services that do not need to be in the kernel, e.g. support for email, file spooling, performing commands at pre-defined times, etc. In particular, daemons are used for various services that allow systems to inter-operate across a network. In order to work, the systems have to be related (e.g. they can be different versions of Unix). The daemons only provide a weak coupling between the systems.

Naming is based on conventions regarding port numbers

The format and content of the actual communication performed by the daemons is daemon-specific, depending on its task. The addressing of daemons relies on universal conventions regarding the usage of certain ports. In Unix systems, the list of well-known services and their ports are kept in the file /etc/services. Here is a short excerpt:

port	*usage*
21	ftp
23	telnet
25	smtp (email)
42	name server
70	gopher
79	finger
80	http (web)

Exercise 181 *What happens if the target system is completely different, and does not adhere to the port-usage conventions?*

As an example, consider the `finger` command. Issuing `finger joe@hostname.dom` causes a message to be sent to port 79 on the named host. It is assumed that when that host booted, it automatically started running a finger daemon that is listening on that port. When the query arrives, this daemon receives it, gets the local information about joe, and sends it back.

Exercise 182 *Is it possible to run two different web servers on the same machine?*

To read more: Writing deamons correctly is an art involving getting them to be completely independent of anything else in the system. See e.g. Stevens [7, Sect. 2.6].

Communication is done using predefined protocols

To contact a server, the client sends the request to a predefined port on faith. In addition, the data itself must be presented in a predefined format. For example, when accessing the finger daemon, the data sent is the login of the user in which we are interested. The server reads whatever comes over the connection, assumes this is a login name, and tries to find information about it. The set of message formats that may be used and their semantics are called a *communication protocol*.

In some cases, the protocols can be quite extensive and complicated. An example is NFS, the network file system. Communication among clients and servers in this system involves many operations on files and directories, including lookup and data access. The framework in which this works is described in more detail in Section 14.2.

12.5 Middleware

Unix daemons are an example of a convention that enables different versions of the same system to interact. To some degree other systems can too, by having programs listen to the correct ports and follow the protocols. But there is a need to generalize this sort of interoperability. This is done by middleware.

Heterogeneity hinders interoperability

As noted above, various communication methods such as RPC and sockets rely on the fact that the communicating systems are identical or at least very similar. But in the real world, systems are very heterogeneous. This has two aspects:

- Architectural heterogeneity: the hardware may have a different architecture. The most problematic aspect of different architectures is that different formats may be used to represent data. Examples include little endian or big endian ordering of bytes, twos-complement or ones-complement representation of integers, IEEE standard or proprietary representations of floating point numbers, and ASCII or EBCDIC representation of characters. If one machine uses one format, but the other expects another, the intended data will be garbled.
- System heterogeneity: different operating systems may implement key protocols slightly differently, and provide somewhat different services.

For example, consider an application that runs on a desktop computer and needs to access a corporate database. If the database is hosted by a mainframe that uses different data representation and a different system, this may be very difficult to achieve.

Middleware provides a common ground

The hard way to solve the problem is to deal with it directly in the application. Thus the desktop application will need to acknowledge the fact that the database is different, and perform the necessary translations in order to access it correctly. This creates considerable additional work for the developer of the aplication, and is specific for the systems in question.

A much better solution is to use a standard software layer that handles the translation of data formats and service requests. This is what middleware is all about.

CORBA provides middleware for objects

The most pervasive example of middleware is probably CORBA (common object request broker architecture). This provides a framework that enables the invokation of methods in remote objects and across heterogeneous platforms. Thus it is an extension of the idea of RPC.

The CORBA framework consists of sevaral components. One is the interface definition language (IDL). This enables objects to provide their specification in a standardized manner, and also enables clients to specify the interfaces that they seek to invoke.

The heart of the system is the object request broker (ORB). This is a sort of naming service where objects register their methods, and clients search for them. The ORB makes the match and facilitates the connection, including the necessary translations to compensate for the heterogeneity. Multiple ORBs can also communicate to create one large object space, where all methods are available for everyone.

12.6 Summary

Abstractions

Interprocess communication as described here is mainly about abstractions, each with its programming interface, properties, and semantics. For example, streams include a measure of synchronization among the communicating processes (you can't receive data that has not been sent yet), whereas shared memory does not include implied synchronization and therefore some separate synchronization mechanism may need to be used.

Resource management

A major resource in communications is the namespace. However, this is so basic that it isn't really managed; Instead, it is either used in a certain way by convention (e.g. well-known port numbers) or up for grabs (e.g. registration with a name server).

Workload issues

Workloads can also be interpreted as a vote of popularity. In this sense, sockets are used overwhelmingly as the most common means for interprocess communication. Most other means of communication enjoy only limited and specific uses.

Hardware support

As we didn't discuss implementations, hardware support is irrelevant at this level.

Bibliography

[1] D. E. Comer, *Computer Networks and Internets*. Prentice Hall, 2nd ed., 1999.

[2] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications*. Prentice Hall, 2nd ed., 1996.

[3] A. Schuster et al., "*MultiView and MilliPage — fine-grain sharing in page-based DSMs*". In 3rd *Symp. Operating Systems Design & Implementation*, pp. 215–228, Feb 1999.

[4] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.

[5] W. Stallings, *Data and Computer Communications*. Macmillan, 4th ed., 1994.

[6] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.

[7] W. R. Stevens, *Unix Network Programming*. Prentice Hall, 1990.

[8] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice Hall, 1995.

[9] A. S. Tanenbaum, *Computer Networks*. Prentice-Hall, 3rd ed., 1996.