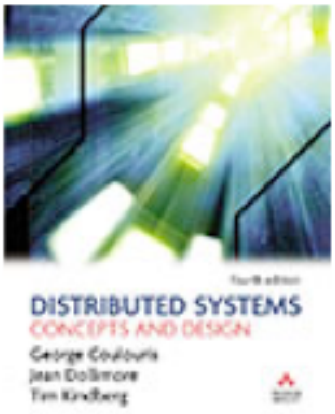


Archive material from *Edition 4* of Distributed Systems: Concepts and Design

Section CDK4–4.6

Originally published as pp. 168- 171 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 4, 2005.

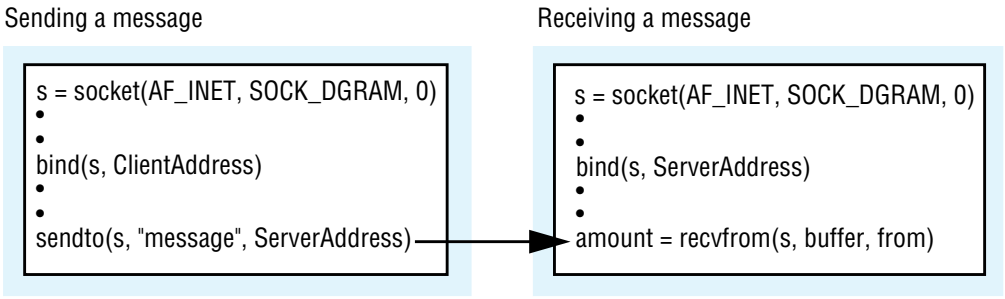
4.6 Case study: interprocess communication in UNIX

The IPC primitives in BSD 4.x versions of UNIX are provided as system calls that are implemented as a layer over the Internet TCP and UDP protocols. Message destinations are specified as *socket addresses* – a socket address consists of an Internet address and a local port number.

The interprocess communication operations are based on the socket abstraction described in Section 4.2.2. As described there, messages are queued at the sending socket until the networking protocol has transmitted them, and until an acknowledgement arrives, if the protocol requires one. When messages arrive, they are queued at the receiving socket until the receiving process makes an appropriate system call to receive them.

Any process can create a socket for use in communication with another process. This is done by invoking the *socket* system call, whose arguments specify the communication domain (normally the Internet), the type (datagram or stream) and sometimes a particular protocol. The protocol (for example, TCP or UDP) is usually selected by the system according to whether the communication is datagram or stream.

Figure 4.21 Sockets used for datagrams



ServerAddress and *ClientAddress* are socket addresses

The socket call returns a descriptor by which the socket may be referenced in subsequent system calls. The socket lasts until it is *closed* or until every process with the descriptor exits. A pair of sockets may be used for communication in both or either direction between processes in the same or different computers.

Before a pair of processes can communicate, the recipient must *bind* its socket descriptor to a socket address. The sender must also bind its socket descriptor to a socket address if it requires a reply. The *bind* system call is used for this purpose; its arguments are a socket descriptor and a reference to a structure containing the socket address to which the socket is to be bound. Once a socket has been bound, its address cannot be changed.

It might seem more reasonable to have one system call for both socket creation and binding a name to a socket, as for example in the Java API. The supposed advantage of having two separate calls is that sockets can be useful without socket addresses.

Socket addresses are public in the sense that they can be used as destinations by any process. After a process has bound its socket to a socket address, the socket may be addressed indirectly by another process referring to the appropriate socket address. Any process, for example a server that plans to receive messages via its socket, must first bind that socket to a socket address and make the socket address known to potential clients.

4.6.1 Datagram communication

In order to send datagrams, a socket pair is identified each time a communication is made. This is achieved by the sending process using its local socket descriptor and the socket address of the receiving socket each time it sends a message.

This is illustrated in Figure 4.21, in which the details of the arguments are simplified.

- Both processes use the *socket* call to create a socket and get a descriptor for it. The first argument of *socket* specifies the communication domain as the Internet domain and the second argument indicates that datagram communication is required. The last argument to the socket call may be used to specify a particular

protocol, but setting it to zero causes the system to select a suitable protocol – UDP in this case.

- Both processes then use the *bind* call to bind their sockets to socket addresses. The sending process binds its socket to a socket address referring to any available local port number. The receiving process binds its socket to a socket address that contains its server port and must be made known to the sender.
- The sending process uses the *sendto* call with arguments specifying the socket through which the message is to be sent, the message itself and (a reference to a structure containing) the socket address of the destination. The *sendto* call hands the message to the underlying UDP and IP protocols and returns the actual number of bytes sent. As we have requested datagram service, the message is transmitted to its destination without an acknowledgement. If the message is too long to be sent, there is an error return (and the message is not transmitted).
- The receiving process uses the *recvfrom* call with arguments specifying the local socket on which to receive a message and memory locations in which to store the message and (a reference to a structure containing) the socket address of the sending socket. The *recvfrom* call collects the first message in the queue at the socket, or if the queue is empty it will wait until a message arrives.

Communication occurs only when a *sendto* in one process addresses its message to the socket used by a *recvfrom* in another process. In client-server communication, there is no need for servers to have prior knowledge of clients' socket addresses, because the *recvfrom* operation supplies the sender's address with each message it delivers. The properties of datagram communication in UNIX are the same as those described in Section 4.2.3.

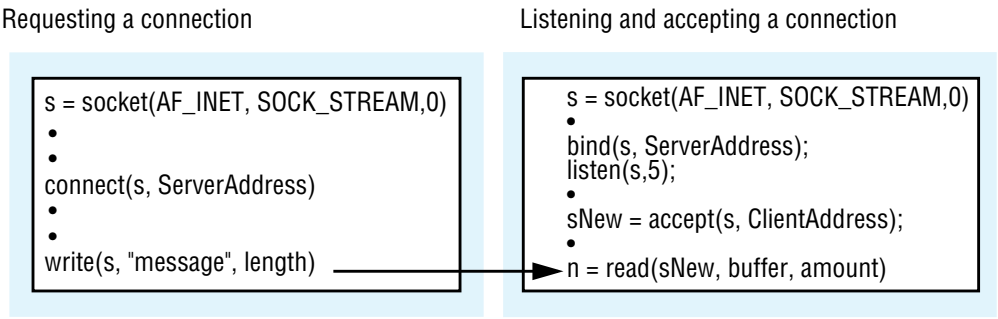
4.6.2 Stream communication

In order to use the stream protocol, two processes must first establish a connection between their pair of sockets. The arrangement is asymmetric because one of the sockets will be listening for a request for a connection and the other will be asking for a connection, as described in Section 4.2.4. Once a pair of sockets has been connected, they may be used for transmitting data in both or either direction. That is, they behave like streams in that any available data is read immediately in the same order as it was written and there is no indication of boundaries of messages. However, there is a bounded queue at the receiving socket and the receiver blocks if the queue is empty; the sender blocks if it is full.

For communication between clients and servers, clients request connections and a listening server accepts them. When a connection is accepted, UNIX automatically creates a new socket and pairs it with the client's socket so that the server may continue listening for other clients' connection requests through the original socket. A connected pair of stream sockets can be used in subsequent stream communication until the connection is closed.

Stream communication is illustrated in Figure 4.22, in which the details of the arguments are simplified. The figure does not show the server closing the socket on which it listens. Normally, a server would first listen and accept a connection and then

Figure 4.22 Sockets used for streams



ServerAddress and *ClientAddress* are socket addresses

fork a new process to communicate with the client. Meanwhile, it will continue to listen in the original process.

- The server or listening process first uses the *socket* operation to create a stream socket and the *bind* operation to bind its socket to the server's socket address. The second argument to the *socket* system call is given as SOCK_STREAM, to indicate that stream communication is required. If the third argument is left as zero, the TCP/IP protocol will be selected automatically. It uses the *listen* operation to listen on its socket for client requests for connections. The second argument to the *listen* system call specifies the maximum number of requests for connections that can be queued at this socket.
- The server uses the *accept* system call to accept a connection requested by a client and obtain a new socket for communication with that client. The original socket may still be used to accept further connections with other clients.
- The client process uses the *socket* operation to create a stream socket and then uses the *connect* system call to request a connection via the socket address of the listening process. As the *connect* call automatically binds a socket name to the caller's socket, prior binding is unnecessary.
- After a connection has been established, both processes may then use the *write* and *read* operations on their respective sockets to send and receive sequences of bytes via the connection. The *write* operation is similar to the write operation for files. It specifies a message to be sent to a socket. It hands the message to the underlying TCP/IP protocol and returns the actual number of characters sent. The *read* operation receives some characters in its buffer and returns the number of characters received.

The properties of stream communication in UNIX are the same as those described in Section 4.2.4.