Socket Programming

Jon A. Solworth

Associate Professor, Department of Computer Science,
University of Illinois at Chicago

http://www.ethos-os.org/~solworth

Sockets

Sockets

Sockets are a protocol independent method of creating a connection between processes. Sockets can be either

- connection based or connectionless: Is a connection established before communication or does each packet describe the destination?
- packet based or streams based: Are there message boundaries or is it one stream?
- reliable or unreliable. Can messages be lost, duplicated, reordered, or corrupted?

Socket characteristics

Socket are characterized by their domain, type and transport protocol. Common domains are:

- AF_UNIX: address format is UNIX pathname
- AF_INET: address format is host and port number

Common types are:

virtual circuit: received in order transmitted and reliably

datagram: arbitrary order, unreliable

Socket characteristics (cont'd)

Each socket type has one or more protocols. Ex:

- TCP/IP (virtual circuits)
- UDP (datagram)

Use of sockets:

- Connection–based sockets communicate client-server: the server waits for a connection from the client
- Connectionless sockets are peer-to-peer: each process is symmetric.

Socket APIs

- socket: creates a socket of a given domain, type, protocol (buy a phone)
- bind: assigns a name to the socket (get a telephone number)
- listen : specifies the number of pending connections that can be queued for a server socket. (call waiting allowance)
- accept: server accepts a connection request from a client (answer phone)
- connect: client requests a connection request to a server (call)
- send , sendto: write to connection (speak)
- recv , recvfrom: read from connection (listen)
- shutdown: end the call

Connection-based communication

Server performs the following actions

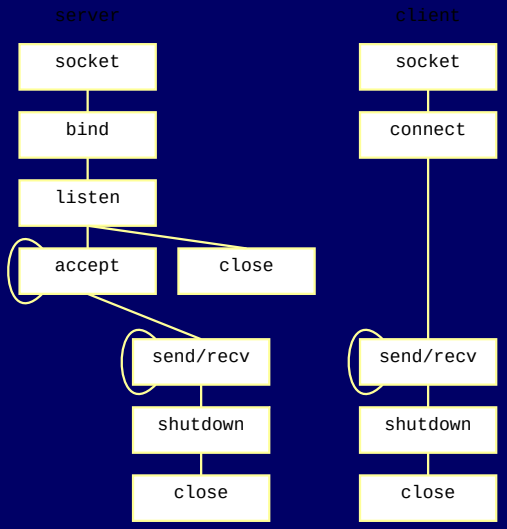
- socket: create the socket
- bind: give the address of the socket on the server
- listen : specifies the maximum number of connection requests that can be pending for this process
- accept: establish the connection with a specific client
- send , recv: stream-based equivalents of read and write (repeated)
- shutdown: end reading or writing
- close: release kernel data structures

TCP client

Client performs the following actions

- socket: create the socket
- connect: connect to a server
- send , recv: (repeated)
- shutdown
- close

TCP-based sockets



socket API

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Returns a file descriptor (called a socket ID) if successful, -1 otherwise.

The type argument can be:

- SOCK_STREAM: Establishes a virtual circuit for stream
- SOCK_DGRAM: Establishes a datagram for communication
- SOCK_SEQPACKET: Establishes a reliable, connection based, two way communication with maximum message size. (This is not available on most machines.)

socket API

Protocol is usually zero, so that type defines the connection within domain.

Note that the socket returns a socket descriptor which is the same as a file descriptor (-1 if failure).

bind

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sid, struct sockaddr *addrPtr, int len)
```

Where

- sid: is the socket id
- addrPtr: is a pointer to the address family dependent address structure
- len: is the size of *addrPtr

Associates a socket id with an address to which other processes can connect. In internet protocol the address is [ipNumber, portNumber]

sockaddr

For the internet family:

```
struct sockaddr_in {
  sa_family_t     sin_family; // = AF_INET
  in_port_t       sin_port;   // is a port number
  struct in_addr  sin_addr;   // an IP address
}
```

For unix sockets (only works between processes on the same machine)

```
struct sockaddr_un {
  uint8_t         sun_length;      //
  short           sun_family;      // = AF_LOCAL
  char            sun_path[100];   // null terminated pathname
                                   // (100 is posix 1.g minimum)
}
```

When using internet sockets, the second parameter of bind (of type sockaddr_in *) must be cast to (sockaddr *).

listen

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sid, int size);
```

Where size it the number of pending connection requests allowed (typically limited by Unix kernels to 5).

Returns the 0 on success, or -1 if failure.

accept

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sid, struct sockaddr *addrPtr, int *lenPtr)
```

Returns the socketId and address of client connecting to socket.

if lenPtr or addrPtr equal zero, no address structure is returned.

lenPtr is the maximum size of address structure that can be called, returns the actual value.

Waits for an incoming request, and when received creates a socket for it.

accept styles

There are basically three styles of using accept:

Iterating server: Only one socket is opened at a time. When the processing on that connection is completed, the socket is closed, and next connection can be accepted.

Forking server: After an accept, a child process is forked off to handle the connection. Variation: the child processes are preforked and are passed the socketId.

Concurrent single server: use select to simultaneously wait on all open socketIds, and waking up the process only when new data arrives.

Pro and Con of Accept styles

- Iterating server is basically a low performance technique since only one connection is open at a time.
- Forking servers enable using multiple processors. But they make sharing state difficult, unless performed with threads. Threads, however present a very fragile programming environment.
- Concurrent single server: reduces context switches relative to forking processes and complexity relative to threads. But does not benefit from multiprocessors.

send

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int sid, const char *bufferPtr, int len, int flag)
```

Send a message. Returns the number of bytes sent or -1 if failure. (Must be a bound socket).

flag is either

- 0: default
- MSG_OOB: Out-of-band high priority communication

recv

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(int sid, char *bufferPtr, int len, int flags)
```

Receive up to len bytes in bufferPtr. Returns the number of bytes received or -1 on failure.

flags can be either

- 0: default
- MSG_OOB: out-of-bound message
- MSG_PEEK: look at message without removing

shutdown

```
#include <sys/types.h>
#include <sys/socket.h>

int shutdown(int sid, int how)
```

Disables sending (how=1 or how=2) or receiving (how=0 or how=2). Returns -1 on failure.

acts as a partial close.

connect

this is the first of the client calls

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sid, struct sockaddr *addrPtr, int len)
```

Specifies the destination to form a connection with (addrPtr), and returns a 0 if successful, -1 otherwise.

Denoting Connections

Note that a connection is denoted by a 5-tuple:

- from IP
- from port
- protocol
- to IP
- to port

So that multiple connections can share the same IP and port.

Port usage

Note that the initiator of communications needs a fixed port to target communications.

This means that some ports must be reserved for these "well knowned" ports.

Port usage:

- 0-1023: These ports can only be binded to by root
- 1024-5000: well known ports
- 5001-64K-1: ephemeral ports

APIs for managing names and IP addresses

We next consider a number of auxiliary APIs:

- The hostent structure: describes IP, hostname pairs
- gethostbyname: hostent of a specified machine
- htons, htonl, ntohs, ntohl: byte ordering
- inet_pton, inet_ntop: conversion of IP numbers between presentation and strings

gethostname

```
#include <unistd.h>

int gethostname(char *hostname, size_t nameLength)
```

Returns the hostname of the machine on which this command executes (What host am i?). Returns -1 on failure, 0 on success.

MAXHOSTNAMELEN is defined in <sys/param.h>.

hostent structure

```
struct hostent {
    char  *h_name;      // official (canonical) name of the host
    char **h_aliases;   // null terminated array of alternative hostnam
    int    h_addrtype;  // host address type AF_INET or AF_INET6
    int    h_length;    // 4 or 16 bytes
    char **h_addr_list; // IPv4 or IPv6 list of addresses
}
```

Error is return through h_error which can be:

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA

Gethostbyname

Auxiliary functions

```
#include <netdb.h>

struct hostent *gethostbyname(const char *hostname)
```

Translates a DNS name into a hostent.

Example:

```
struct hostent *hostEntity =
        gethostbyname("bert.cs.uic.edu");
memcpy(socketAddr->sin_addr,
        hostEntity->h_addr_list[0],
        hostEntity->h_length);
```


Network byte ordering

Network ordering in big endian. (Sparc is big endian, Intel is little endian).

```
// Host to network byte order for shorts (16 bit)
uint_16t htons(uint_16t v);

// Host to network byte order for long (32 bit)
uint_32t htonl(uint_32t v);

// Network to host byte order for long (16 bit)
uint_16t ntohs(uint_16t v);

// Network to host byte order for long (32 bit)
uint_32t ntohl(uint_32t v);
```

IP Number translation

IP address strings to 32 bit number

In what follows, 'p' stands for presentation.

Hence, these routines translate between the address as a string and the address as the number.

Hence, we have 4 representations:

- IP number in host order
- IP number in network order
- Presentation (eg. dotted decimal)
- Fully qualified domain name

Only the last needs an outside lookup to convert to one of the other formats.

inet_pton

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strPtr, void *addrPtr);
```

returns 1 if OK, 0 if presentation error, -1 error

Where family is either AF_INET or AF_INET6.

The strPtr is the IP address as a dotted string.

Finally, addrPtr points to either the 32 bit result (AF_INET) or 128 bit result (AF_INET6).

inet_ntop

```
#include <arpa/inet.h>

int inet_ntop(int family, const char *addrPtr,
              char *strPtr, size_t len);
```

returns 1 if OK, 0 if presentation error, -1 error

Where family is either AF_INET or AF_INET6.

The strPtr is the return IP address as a dotted string.

Finally, addrPtr points to either the 32 bit (AF_INET) or 128 bit (AF_INET6).

Length is the size of destination.

Example: TCP/IP Server Code

Without error checking.

```
sockaddr_in serverAddr;
sockaddr &serverAddrCast = (sockaddr &) serverAddr;

// get a tcp/ip socket
int listenFd = socket(AF_INET, SOCK_STREAM, 0);

bzero(&serverAddr, sizeof(serverAddr));
serverAddr.sin_family = AF_INET;
// any internet interface on this server.
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddr.sin_port = htons(13);

bind(listenFd, &serverAddrCast, sizeof(serverAddr));

listen(listenFd, 5);

for ( ; ; ) {
    int connectFd =
        accept(listenFd, (sockaddr *) NULL, NULL);
```

Example: TCP/IP Server Code

```
20        // .. read and write operations on connectFd ..
21      shutdown(connectFd, 2);
22      close(connectFd);
23  }
```

Note that the above is an iterative server, which means that it serves one connection at a time.

To build a concurrent server:

- a fork is performed after the accept.
- The child process closes listenFd, and communicates using connectFd.
- The parent process closses connectFd, and then loops back to the accept to wait for another connection request.

Example: TCP/IP Client code

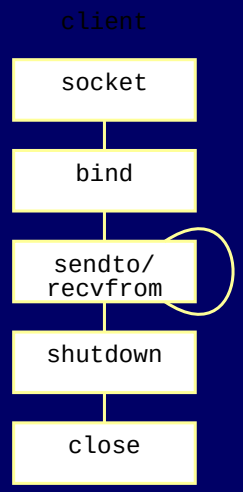
```
1  sockaddr_in serverAddr;
2  sockaddr &serverAddrCast = (sockaddr &) serverAddr;
3
4  // get a tcp/ip socket
5  int sockFd = socket(AF_INET, SOCK_STREAM, 0);
6
7  bzero(&serverAddr, sizeof(serverAddr));
8  serverAddr.sin_family = AF_INET;
9  // host IP # in dotted decimal format!
10 inet_pton(AF_INET, serverName, serverAddr.sin_addr);
11 serverAddr.sin_port = htons(13);
12
13 connect(sockFd, serverAddrCast, sizeof(serverAddr));
14       // .. read and write operations on sockFd ..
15 shutdown(sockFd, 2);
16 close(sockFd);
```


Connectionless communication

Communication is symmetric (peer-to-peer)

- socket
- bind: bind is optional for initiator
- sendto, recvfrom (repeated)
- shutdown
- close

Connectionless communication



UDP variations

It is not necessary for both sockets to bind

- The receiver gets the address of the sender

It is possible for a UDP socket to connect

- In this case, send/recv (or write/read) must be used instead of sendto/recvfrom.
- Asynchronous errors can be returned (using ICMP)

sendto

for connectionless protocols

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int sid, const void *bufferPtr,
           size_t bufferLength, int flag,
           struct sockaddr *addrPtr, socklen_t addrLength)
```

Send a buffer, bufferPtr, of length bufferLength to address specified by addrPtr of size addrLength. Returns number of bytes sent or -1 on error.

recvfrom

for connectionless protocols

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom(int sid, void *bufferPtr, int bufferLength,
             int flag, sockaddr *addrPtr, int *addrLengthPtr)
```

Receive a buffer in bufferPtr of maximum length bufferLength from an unspecified sender.

Sender address returned in addrPtr, of size *addrLengthPtr.

Returns number of bytes receive or -1 on error.

Example: UDP—server

```
int socketId = socket(AF_INET, SOCK_DGRAM, 0);

sockaddr_in serverAddr, clientAddr;
sockaddr &serverAddrCast = (sockaddr &) serverAddr;
sockaddr &clientAddrCast = (sockaddr &) clientAddr;

// allow connection to any addr on host
// for hosts with multiple network connections
// and ast server port.
serverAddr.sin_family = AF_INET;
serverAddr.sin_port   = htons(serverPort);
serverAddr.sin_addr.s_addr = INADDR_ANY;

// associate process with port
bind(socketId, &serverAddrCast, sizeof(addr));

// receive from a client
int size = sizeof(clientAddr);
recvfrom(socketId, buffer, bufferSize,
         0, clientAddrCast, &size);
```

Example: UDP—server

```
21
22  // reply to the client just received from
23  sendto(socketId, buffer, bufferSize,
24         0, clientAddrCast, size);
25
26  close(socketId);
```


Example: UDP—client

```
1   int socketId = socket(AF_INET, SOCK_DGRAM, 0);
2
3   sockaddr_in serverAddr, clientAddr;
4   sockaddr &serverAddrCast = (sockaddr &) serverAddr;
5   sockaddr &clientAddrCast = (sockaddr &) clientAddr;
6
7   // specify server address, port
8   serverAddr.sin_family = AF_INET;
9   serverAddr.sin_port   = htons(serverPort);
10  struct hostent *hp = gethostbyname(hostName);
11  memcpy((char*)&serverAddr.sin_addr,
12         (char*)hp->h_addr, hp->h_length);
13
14  // no need to bind if not peer-to-peer
15  int size = sizeof(serverAddr);
16  sendto(socketId, buffer, bufferSize, 0,
17         serverAddrCast, size);
18
19  recvfrom(socketId, buffer, bufferSize, 0,
20         serverAddrCast, &size);
```


Example: UDP—client

```
21
22  close(socketId);
```


API Index