Sockets

Introduction

- sockets provide a programming interface to network protocols
- we'll only deal with version 4 of the TCP/IP protocol suite (IPv4)
- the 2 main protocols in the suite that most applications use are TCP (Transmission Control Protocol) & UDP (User Datagram Protocol)
- both TCP & UDP run on top of IP
- some applications (e.g. `ping`, `traceroute`) use ICMP (Internet Control Message Protocol)

TCP

- data sent in "packets" called segments
- a connection-oriented protocol
  - state information is maintained about successive packets
  - a connection is established before data is transferred; at the end, the connection is torn down
- full-duplex
- reliable
  - acknowledges arrival of data
  - automatically retransmits data that do not arrive at their destination
  - uses a checksum to guarantee that data is not corrupted
  - uses sequence numbers to ensure that data arrives in order & automatically eliminates duplicate packets
- provides flow control to ensure sender does not transmit data faster than the receiver can consume
- data can be regarded as a stream of 8-bit bytes: TCP does not automatically insert record markers
- most network applications are of the client-server type; the typical sequence of function calls for TCP-based clients & servers are:
  - client: `socket` → `connect` → `read/write` → `close`
  - server: `socket` → `bind` → `listen` → `accept` → `read/write` → `close`

The `socket` Function

- a socket is an endpoint for communication
- the `socket` function is used to create a new socket; it returns a "socket" descriptor which is allocated in the same descriptor table as file descriptors

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

int socket(int domain, int type, int protocol);
/* returns a descriptor referencing the created socket on success,
   -1 on error & sets errno */
```

- `domain` specifies the communications domain & selects the protocol family to be used; possible values are defined in `sys/socket.h`

  we'll only be using `AF_UNIX` (or `PF_UNIX`) for Unix domain sockets & `AF_INET` (or `PF_INET`) for Internet domain sockets

- `type` specifies the semantics of communication; we'll only be using `SOCK_STREAM` & `SOCK_DGRAM`
  - `SOCK_STREAM` specifies a sequenced, reliable, 2-way connection based on byte streams
  - `SOCK_DGRAM` specifies support for datagrams (connectionless, unreliable messages of a fixed maximum length)
- `protocol` specifies the specific protocol within the protocol family to use; since this is usually already determined by the `domain` & `type` arguments, we'll usually set this to 0 to let the system choose for us

  Example Invocations

```
sock1 = socket(AF_INET, SOCK_STREAM, 0);  /* TCP */
sock2 = socket(AF_INET, SOCK_DGRAM, 0);   /* UDP */
```

The `bind` Function

- a server typically calls `bind` to give a socket a local address (assigning a "name" to a socket)

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
/* returns 0 on success, -1 on error & sets errno */
```

- `sockfd` is the file descriptor of the socket
- `struct sockaddr` is the address structure (see next section); the address structure pointed to by `my_addr` is used to specify the address of the local machine (recall that only the server calls `bind`; but be aware that an address structure is also required to specify the address of the server when the client calls the `connect` function)
- `addrlen` is the size of the address structure pointed to by `my_addr`

Address Structures

- the second argument to `bind` is a pointer to a `sockaddr` structure
- in order to accomodate different protocols that require different address formats, there are actually different types of address structures
- a `sockaddr` structure is a generic address structure; it is defined in `sys/socket.h`, typically as:

```
struct sockaddr {
  sa_family_t  sa_family;      /* address family, AF_xxx       */
  char         sa_data[14];    /* 14 bytes of protocol address */
};
```

Note that some definitions of this structure may have an extra member: `uint8_t sa_len;`

In that case, the address structure that follows will also have a corresponding extra member

- the address structure for an Internet domain socket (for IPv4) is defined in `netinet/in.h`, typically as:

```
struct sockaddr_in {
  sa_family_t     sin_family;  /* Address family: AF_INET                */
  in_port_t       sin_port;    /* 16-bit port number, network byte order */
  struct in_addr  sin_addr;    /* 32-bit IP address, network byte order  */
  unsigned char   sin_zero[8]; /* Pad to size of 'struct sockaddr'       */
};
```

- we'll look at the `in_addr` structure below
- byte ordering: different systems number bytes within a word differently; the convention used by a system is called its (host) byte order
  - big-endian: high-order byte at starting address
  - little-endian: low-order byte at starting address
- for communication between different machines, a byte order must be agreed upon; this byte order is called the network byte order
- the Internet protocols use big-endian byte ordering; there are functions to convert between the host byte order & the network byte order

```
#include <netinet/in.h>

uint32_t  htonl(uint32_t hostlong);
uint16_t  htons(uint16_t hostshort);
uint32_t  ntohl(uint32_t netlong);
uint16_t  ntohs(uint16_t netshort);
```

`htonl` stands for "host to network long", `htons` for "host to network short", etc

- these functions should be used when we specify the port number & IP address (for the `sin_port` & `sin_addr` members of the `sockaddr_in` structure)
- in Unix, any port less than 1024 is reserved & can only be assigned to a socket by a superuser process
- the `in_addr` structure is typically defined as:

```
struct in_addr {
  in_addr_t      s_addr;  /* 32-bit IP address, network byte order */
};
```

- for a server, it is usual to use the special constant `INADDR_ANY` (converted to network byte order) for the `s_addr` member when the address structure is used in `bind`; `INADDR_ANY` specifies a wildcard address that matches any of the host's IP addresses; this is particularly useful for multihomed hosts

Example (for server)

```
#include <sys/types.h>
#include <sys/socket.h>
...
#define PORT     1066
...
int                 sock;
struct sockaddr_in  sin;
...
if ( (sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {  /* TCP */
  perror("socket");
  return 1;
}
memset(&sin, 0, sizeof(sin));  /* zero out structure */
sin.sin_family = AF_INET;
sin.sin_port = htons(PORT);
sin.sin_addr.s_addr = htonl(INADDR_ANY);
bind(sock, (struct sockaddr *) &sin, sizeof(sin));  /* note cast */
```

- a client usually specifies a machine by its IP address in dotted-decimal format (e.g. 18.69.0.41) or by its hostname or FQDN (fully-qualified domain name; e.g. gnu.mit.edu)
- for dotted-decimal strings: there are functions to convert an IP address between a dotted-decimal string & its 32-bit network byte order format: `inet_aton` & `inet_ntoa` (also `inet_addr`, which is deprecated)

  (`cp` is the dotted-decimal string in the following prototypes)

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

int  inet_aton(const char *cp, struct in_addr *inp);
/* returns nonzero if address is valid, 0 if not */

char *inet_ntoa(struct in_addr in);
/* returns the host address in dotted-decimal notation stored in a
   statically allocated buffer which subsequent calls will overwrite */
```

Example (for client)

```
#define IPADDR   "142.232.10.1"
#define PORT     1066
...
struct sockaddr_in  sin;
...
< create socket >

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORT);
inet_aton(IPADDR, &sin.sin_addr);

< call connect function passing in sin; see later >
```

- for FQDNs: because a host can have several aliases, these are slightly more complicated to use; we look at 2 functions that deal with them:

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, size_t len, int type);
/* both return a pointer to a hostent structure on success
   or the NULL pointer on failure */
```

- the `hostent` structure is typically defined as:

```
struct hostent {
  char  *h_name;       /* official name of host */
  char **h_aliases;    /* ptr to array of ptrs to alternative host names,
                          terminated by null pointer */
  int    h_addrtype;   /* host address type; AF_INET for IPv4 */
  int    h_length;     /* length (in bytes) of address; 4 for IPv4 */
  char **h_addr_list;  /* ptr to array of ptrs to network addresses (in network
                          byte order) for host, terminated by null ptr */
};
```

  For backward compatibility, some systems define `h_addr` to be the first element in `h_addr_list` (i.e. `h_addr_list[0]`)

- `gethostbyname` searches the network database for an entry which matches the host name specified by `name`; if `name` is an alias for a valid host name, the function returns information about the host name to which the alias refers, & `name` is included in the list of aliases returned

- `gethostbyaddr` searches the network database for an entry which matches the address family specified by `type` & which matches the address pointed to by `addr`; for type AF_INET, `addr` is a pointer to an `in_addr` structure & `len` is 4

- both functions may need to contact a nameserver or do a lookup in the `/etc/hosts` file

  Example (for client)

```
#include <netdb.h>
...
#define HOSTNAME  "gnu.mit.edu"
#define PORT      1066
...
struct sockaddr_in  sin;
struct hostent      *hp;
...
< create socket >
if ( (hp = gethostbyname(HOSTNAME)) == NULL ) {
  fprintf(stderr, "Unable to resolve hostname\n");
  return 1;
}

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
```

```
sin.sin_port = htons(PORT);
memcpy(&sin.sin_addr, hp->h_addr_list[0], sizeof(sin.sin_addr));
< call connect function passing in sin; see later >
```

The `listen` Function

- after binding the socket, a connection-oriented (e.g. TCP-based) server calls the `listen` function to put the socket into passive mode, ready to accept incoming connections

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

int listen(int sockfd, int backlog);
/* returns 0 on success, -1 on error & sets errno */
```

- `sockfd` is the file descriptor of the socket
- `backlog` specifies the maximum number of pending connections in the socket's listen queue; if a connection request arrives with the queue full, the client may receive an error of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed; a typical value of `backlog` is 5

The `accept` Function

- a connection-oriented server calls `accept` to accept a new connection on a socket

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

int accept(int sockfd, struct sockaddr *addr, int *addrlen);
/* returns (nonnegative) file descriptor of new socket on success,
   -1 on error & sets errno */
```

- `sockfd` is a socket that has been created with `socket`, bound to an address with `bind`, & is listening for connections after a `listen`
- `accept` extracts the first connection on the queue of pending connections, creates a new socket with the same properties as `sockfd`, allocates & returns a new file descriptor for the new socket; the original socket `sockfd` remains open
- if no pending connections are present on the queue (& the socket is not marked as non-blocking), `accept` blocks the caller until a connection is present
- for IPv4, `addr` should be the address of a `sockaddr_in` structure (that we have created); this structure will be filled in by the system with the address information of the connecting client
- `addrlen` is a "value-result argument"; it should be the address of an integer variable that contains the size of the structure we are passing in through `addr`; on return from `accept`, the value of the integer variable is set to the actual size of the address returned
- if we are not interested in the address information, we can pass in the null pointer for both `addr` & `addrlen`

The `connect` Function

- a typical connection-oriented client will call the `connect` function to connect to a server

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

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
/* returns 0 on success, -1 on error & sets errno */
```

- `sockfd` is the file descriptor of the socket
- the address structure pointed to by `addr` contains address information of the server; for TCP, this structure is of type `struct sockaddr_in`
- `addrlen` is the size of the structure passed in through `addr`
- when a TCP client calls `connect`, a port (called an ephemeral port) is chosen by the system; usually a client does not specify a local port to use

Other Functions

- for stream sockets, after a connection has been established, the client & the server can communicate by writing to (using `write`) & reading from (using `read`) the socket
- when we are done communicating, use `close` to close the socket
- since a socket is referenced by a file descriptor, `read`, `write` & `close` all work with sockets just as with files
- however, a `read` or `write` on a stream socket may input or output fewer bytes than requested, but that is not an error; this means that we must use a loop when we want to read or write a specific number of bytes

  Examples

```
/*
   function to read a specified number of bytes
   into a buffer from a file descriptor
*/
int readn(int fd, void *buf, int n) {
  int   nread;
  char  *p = buf, *q = buf + n;

  while (p < q) {
    if ( (nread = read(fd, p, q - p)) < 0) {
      if (errno == EINTR)
        continue;
      else
        return -1;
    } else if (nread == 0)  /* EOF */
      break;

    p += nread;
  }
  return p - (char *) buf;
}
```

```
/*
   function to write a specified number of bytes
   stored in a buffer to a file descriptor
*/
int writen(int fd, const void *buf, int n) {
  int   nwrite;
  char  *p = buf, *q = buf + n;

  while (p < q) {
    if ( (nwrite = write(fd, p, q - p)) <= 0) {
      if (errno == EINTR)
        continue;
      else
        return -1;
    }

    p += nwrite;
  }

  return n;
}
```

- the reason several reads or writes may be necessary is because of the way TCP works – TCP may choose to break a block of data into pieces & transmit each piece in a segment, or it may choose to accumulate many bytes in its output buffer before sending a segment
- `read` returns 0 to indicate end-of-file for a TCP socket; usually this means the other end has closed its socket
- the `shutdown` function can be used to close all or part of a TCP connection

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

int shutdown(int sockfd, int how);
/* returns 0 on success, -1 on error & sets errno */
```

  - `sockfd` is the socket descriptor
  - if `how` is `SHUT_RD`, further receives are disallowed; if `how` is `SHUT_WR`, further sends are disallowed & if `how` is `SHUT_RDWR`, further sends & receives are disallowed