

CSIS0234B Computer and Communication Networks (Class B)

Reading for Tutorial 1

Socket API

There are two commonly used C language interfaces for programs to communicate via the Internet. One, called **socket**, is initially developed for the BSD Unix operating system. The other, called **XTI** (X/Open Transport Interface), is initially developed by AT&T for SVR3. Since now virtually everybody use sockets, we describe only sockets.

Sockets extend the notion of Unix that “everything is (looks like) a file”. **Sockets are file descriptors**, so you can read (receive), write (send) and close them just like you can read, write and close files, using the `read()`, `write()` and `close()` system calls that you’ve learnt in the OS course. Its behaviour is not unlike a Unix pipe or FIFO.

However, sockets are for communication over network, thus it needs some functionalities not provided by file descriptors. A number of system calls are added to provide these functionalities: create sockets, bind them to particular ports (possibly of particular network cards), prepare them for others to connect and wait until that happens (**passive** open), connect to another socket of another computer which is waiting (**active** open), etc. In a C program, one should include `<sys/types.h>`, `<sys/socket.h>`, `<netinet/in.h>` and `<arpa/inet.h>` to access these system calls. We will introduce them one by one. During the tutorial we will write an Internet client, so you may skim the sections for servers if time is running short.

Like all other system calls, **whenever an error is detected, -1 (of the return type) is returned**, and the error code is stored in the `errno` variable (accessible if you include `<errno.h>`). To find the possible errors, read the manpage of the corresponding system calls.

1. Creating sockets

Socket is a very flexible programming interface, and is designed to allow connections through many different types of networks and protocols. These are specified when a socket is created.

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

The `domain` argument specifies the “protocol family”, i.e., type of network being used. For using the Internet, `domain` is always set to `PF_INET`. The `type` argument specifies the “communication semantics”. We focus on a single one, `SOCK_STREAM`, which has the semantics of a sequenced, reliable, two-way, connection-based byte stream. Since Internet only has one such protocol (“TCP”, Transmission Control Protocol), one may set `protocol` to 0, or to `IPPROTO_TCP` to be explicit (see the manpage `ip(7)`). E.g.,

```
int sockfd = socket(PF_INET, SOCK_STREAM, 0);  
if (sockfd == -1)  
    /* Handle error */
```

The operating system allocates a file descriptor, ready for connecting to other computers.

2. Establishing a connection

To connect to another server, one can use the following system call:

```
int connect(int sockfd, const struct sockaddr * servaddr, socklen_t addrlen);
```

It tries to initiate a connection using the socket `sockfd`, returning 0 if it succeeds. The other end is specified by `servaddr`, of type `struct sockaddr *`. Recall that sockets can be used not only for Internet connections, but for many other types of networks as well. How can the single type, `struct sockaddr`, hold an address of many different types of networks? The answer: it can't, so the structure is in fact not a `struct sockaddr`. Instead, the `socket()` system call reads the network type and protocol you specify, and registers a function for connecting to other hosts. That function will cast the pointer to the correct type, for our case `sockaddr_in`. The `sockaddr` structure is just a place-holder, defined like this:

```
/* 16-byte structure: sa_family_t is actually a short int */
struct sockaddr {
    sa_family_t    sa_family;    /* address family, AF_XXX */
    char          sa_data[14]; /* 14 bytes of protocol address*/
};
```

The “real” address structure `sockaddr_in`, defined in `<netinet/in.h>`, looks like this:

```
struct in_addr {
    uint32_t s_addr;    /* Internet address in network byte order */
};
struct sockaddr_in {
    sa_family_t    sin_family;    /* Address family: AF_INET */
    unsigned short int sin_port;    /* Port in network byte order */
    struct in_addr    sin_addr;    /* Internet address */
    unsigned char    __pad[8];    /* Fill enough space */
};
```

So our client program uses the following to connect to a server:

```
struct sockaddr_in addr;
/* Fill in the addr */
if (connect(sockfd, (struct sockaddr *)&addr, sizeof(addr)) == -1)
    /* handle error */
```

3. Filling in the address structure

But how to fill in the address? The address of a TCP socket consists of two parts, the **Internet address** of the computer¹ and the **port number** used for the socket. Most programs leave the address of the client unspecified, so values can be chosen by the `connect()` system call. On the other hand, the server side address must be known before one can connect to it.

Most programs are given the Internet address in an ASCII form, like "147.8.177.14". This has to be converted into a 32-bit number before it can be set to the `sockaddr_in` structure. The following function performs this tedious task:

```
int inet_aton(const char *strptr, struct in_addr *addrptr);
char *inet_ntoa(struct in_addr inaddr);
```

¹Here we assume that each computer has one Internet address. This is actually not really correct: a computer can have many Internet addresses. In particular, each interface (network card, ppp connection, etc) of a computer uses a different Internet address.

The `inet_aton()` function converts the C-style string `strptr` to a 32-bit number and store it in the structure `*addrptr`. The `inet_ntoa()` function does the reverse, takes the 32-bit Internet address in a `in_addr` structure and turn it to a C-style string suitable for printing.

The fields in the `sockaddr_in` structure is assumed to be in **network byte order**. Since sockets are designed for the BSD system when the predominant computer (VAX) uses big-endian (i.e., most significant byte first), it is not surprising that network byte order means big-endian. Many computers use little-endian, so it is important to make sure byte order conversions is done when needed. The byte order used by the computer is called the **host byte order**, so there are functions to convert from host to network byte order and vice versa.

```
uint16_t htons(uint16_t host16bitvalue);    /* host to network short */
uint32_t htonl(uint32_t host32bitvalue);    /* host to network long */
uint16_t ntohs(uint16_t net16bitvalue);     /* network to host short */
uint32_t ntohl(uint32_t net32bitvalue);     /* network to host long */
```

Since `inet_aton()` fills the `in_addr` structure in network byte order, conversion should not be done for Internet addresses obtained using `inet_aton()`. Our client would thus fill the remote address as follows:

```
memset(&addr, 0, sizeof(addr));             /* clear it first */
addr.sin_family = AF_INET;
inet_aton("147.8.177.14", &addr.sin_addr);
addr.sin_port = htons(1234);                /* if the server port is 1234 */
```

It is also common to use a human-readable hostname like `virtue.csis.hku.hk` instead of Internet addresses. While it is possible to ask the user to look up Internet address using the `host` command (like `host virtue.csis.hku.hk`), it is usually better to do it in our program. This involves making a connection to a “Domain Name Server” in order to find the Internet address for the host we will contact. Again, this tedious task is done by a standard function:

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

Given a hostname in a C-style string, it returns a pointer to a structure which contains information about the host. (On error it returns NULL.) The structure is defined in `<netdb.h>` as follows:

```
struct hostent {
    char *h_name;           /* Official name of the host */
    char **h_aliases;      /* Alias list */
    int h_addrtype;        /* Host address type suitable for sin_family */
    int h_length;          /* Length of each address */
    char **h_addr_list;    /* NULL-terminated list of addresses */
}
```

So our program can do the following to find the host name:

```
struct hostent *hp;
hp = gethostbyname("virtue.csis.hku.hk");
if (hp == NULL)
    /* handle error */
addr.sin_family = hp->h_addrtype;
memcpy(&addr.sin_addr, hp->h_addr_list[0], hp->h_length);
```

4. Closing the connection

The standard `close()` system call is used to close a connection, so it should be something you are familiar with. This section describes the behaviour when a connection is closed. The system call returns immediately, even if there are data that are buffered but not sent already. Once `close()` is called, the program can no longer make use of the socket to send and receive data. However, by default the operating system does not immediately terminate the connection. Instead, it tries to send the remaining buffered data, and terminate the connection only after that. If the remote end tries to send data to it after it is closed, the remote end will receive a signal (SIGPIPE), which would normally terminate the process immediately (that is, unless a signal handler is registered for it). Again this looks very much like pipes.

5. Preparing to receive a connection

As we already noted, passively waiting for connections (used by servers) is quite different from actively making connections (used by clients). The socket returned by the `socket()` call is ready for “active open”. For “passive open”, some further steps are required. In particular, one must **bind** it to a port (and possibly address, i.e., network card) for remote computers to find. Then it must allocate a table for connections to be made.

```
int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen);  
int listen(int sockfd, int backlog);
```

The system call `bind()` assigns an address to a socket. The arguments have the same format as `connect()`, but `myaddr` specifies the address of a local network interface, not a remote one. Most servers use it to specify a well-known port for the clients to connect to, while setting `sin_addr.s_addr` to `INADDR_ANY` to allow connections from any network interfaces.

The system call `listen()` allocates a table of connections. The `backlog` argument specifies the number of table entries to allocate. Once this is done, connections can be made to the server, even when the server program is not executing `accept()` (see the next section). If there are more than `backlog` incoming connections that are not yet `accept`'ed, further connections will be refused (the remote end get the `ECONNREFUSED` error when `connect()` is called).

A simple server program would thus have the following to setup the socket:

```
struct sockaddr_in addr;  
memset(&addr, 0, sizeof(addr));  
addr.sin_family = AF_INET;  
addr.sin_port = htons(1234); /* if the server port is 1234 */  
addr.sin_addr.s_addr = INADDR_ANY;  
if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) == -1)  
    /* handle error */  
if (listen(sockfd, 5) == -1)  
    /* handle error */
```

6. Waiting for a connection

When the server is ready for it, it waits for a connection using `accept()`.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

This will block until one of the table entries is filled by a client connecting to it. Then a **new** file descriptor is returned for the connection created. So once the call returns, we have **two** sockets: the old one which can be used for `accept()` again, and the new one which can be used for data communication. We call the old one a **listening socket**, and the new one a **connected socket**. Note that the listening socket and all the connected sockets will have the same local address. This is okay, since a TCP connection is identified by the address of **both** sides. On the other hand, it is impossible for multiple processes to bind to exactly the same local address¹. It is possible to find the remote address, by setting `addr` to point to a `struct sockaddr_in` structure, and set `addrlen` to a pointer pointing to a location that stores the size of `*addr`. On return, the real length of the remote address will be stored in `*addrlen`, and the remote address will be stored in `*addr`. If this is not needed, one can set `addr` to `NULL`.

A simple **iterative server** thus has the following structure after getting the listening socket:

```
struct sockaddr_in remote_addr;
socklen_t remote_addrlen = sizeof(remote_addr);
for (;;) {
    int newfd = accept(sockfd, (struct sockaddr *)&remote_addr,
                      &remote_addrlen);
    /* read and write using newfd */
    close(newfd);
}
```

7. Concurrent server

An iterative server is good if the duration of each connection is known to be short. Otherwise many clients need to wait for a long time, since no connection can be made when the server is busy talking with another client. Since most TCP servers can have long connections, most create another thread or process to handle the new connection, and the “main” server process or thread returns to `accepting`. The simplest way to do it is by on-demand forking:

```
struct sockaddr_in remote_addr;
for (;;) {
    int newfd = accept(sockfd, &remote_addr, sizeof(remote_addr));
    pid_t pid = fork();
    if (pid == 0) { /* child */
        close(sockfd);
        if (fork() != 0) /* fork and exit so the parent can wait */
            exit(0);
        /* read and write using newfd */
        close(newfd);
        exit(0);
    }
    close(newfd); /* not my business */
    wait(0); /* to prevent zombie */
}
```

¹However, it is possible for a process to bind a port, call `listen()`, and then `fork()`. Then multiple processes will have the same listening socket, and can all call `accept()` at the same time. The result: they will contend for connections in the connection table. This can be a good idea for very busy servers, since when one process is busy forking (see the next section), the others can still accept connections. We call this “pre-forking”.

Here double forking is used to prevent zombie processes: the first child always return quickly and is waited, the second child may return slowly but the parent is dead. The `close(newfd);` call of the parent does not really close the connected file descriptor, since the child (actually, grand-child) is still holding a copy of the socket.

8. Controlling socket operations

Sometimes the default behaviour of a TCP connection is not what you want. TCP/IP has a number of options, which can be controlled by `setsockopt()`:

```
int setsockopt(int sockfd, int level, int optname, const void *optval,  
              socklen_t optlen);
```

The option of `sockfd` is manipulated. Since there are multiple layers (levels) where `sockfd` belongs, `level` is needed to select among them. E.g., a TCP connection has socket level, tcp level and ip level options, selected using `SOL_SOCKET`, `SOL_TCP` and `SOL_IP`. The option name and value are specified using `optname` and `optval`. They have to be looked up from the manpages (`socket(7)`, `tcp(7)` and `ip(7)`). For example, if you are debugging a server, you will probably start and stop the server frequently. But for safety measures, the OS normally disallow a port with previous connection to be reused for around 5 minutes to prevent old data from interfering with the new one. But you probably don't want to wait 5 minutes everytime you find a bug and restart the server. Then you can set the socket level option `SO_REUSEADDR` to 1 to disable this safety net, before binding:

```
/* FIXME: delete these after debugging */  
int val = 1;  
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
```

9. Sample code

A very simple echo server and client has been included in the source of the tutorial notes. You can type `make sample` on a Unix computer to compile and test them. The programs are called `testserver` and `testclient`.