

# CSIS0234B Computer and Communication Networks (Class B)

## Reading for Tutorial 1

### Socket API

Most C programs use the network by employing the **Berkeley socket** interface<sup>1</sup>, which is initially developed for the BSD Unix operating system but is eventually ported to virtually all other platforms. Sockets extend the notion of Unix that “everything is (looking 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. We will introduce them one by one. In a C program, one should include `<sys/types.h>` and `<sys/socket.h>` to access these system calls. During the tutorial we will write an Internet client, so you may skim the sections for servers (Section 4–6) and save them until the next week if you are too busy this week.

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.

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`.

#### 1. Where to connect to?

To create a socket, one needs to know what is the type of socket to create, e.g., which version of Internet Protocol (IP) we are using. After we have a socket we also need to know the network addresses of the computer that we want to connect to. The type of this address also differs with different protocols. The types can be hard-coded into the program, but this is not robust, especially when we are facing an upgrade to the underlying IP<sup>2</sup>. Luckily, we have an alternative: the function `getaddrinfo()` which can detect the socket and address type. We only need to tell what type of communication we want to perform, together with a network address and a service that we want to connect to (or listen to). The function has the following prototype:

```
int getaddrinfo(const char *hostname, const char *service,  
               const struct addrinfo *hints, struct addrinfo **res_ptr);
```

The `hostname` argument is a string to specify the network address to connect or bind to. It might be in a “dotted-quad” form like “147.8.178.13” (for IPv4 addresses), or in an IPv6 address notation like “2002:9308:b20d::1”, or in a “domain-name” form like “virtue.csis.hku.hk”. If a domain name is used, the function uses the domain name system (DNS) to find the address and address type needed (in this case, both IPv4 and IPv6 is possible). The argument might be `NULL`, which specifies an address for communicating with itself (but see `AI_PASSIVE` below).

---

<sup>1</sup>A competing interface, X/Open Transport Interface (XTI), is developed for SVR3. It is not as popular as sockets.

<sup>2</sup>Here we refer to the upgrade from IPv4, the version 4 of IP, to IPv6. Depending on how the limited number of IP addresses are damaging everybody, it might take several years or around a decade to be fully adopted.

The `service` argument is a string to specify the service to use. It can be a service name (like "ftp"), or a **port number** (like "1234"). In the latter case, the `hints` argument must be used to tell what type of connection you want.

Both the `hints` argument and the `res_ptr` argument use a type which is called `struct addrinfo`. It is defined in `<netdb.h>` as follows:

```
struct addrinfo {
    int ai_flags;           /* Input flags. */
    int ai_family;         /* Protocol family for socket. */
    int ai_socktype;       /* Socket type. */
    int ai_protocol;       /* Protocol for socket. */
    socklen_t ai_addrlen;  /* Length of socket address. */
    struct sockaddr *ai_addr; /* Socket address for socket. */
    char *ai_canonname;    /* Canonical name for service location. */
    struct addrinfo *ai_next; /* Pointer to next in list. */
};
```

The `hints` argument is used to specify the preferred socket type or protocol. If `hints = NULL`, everything is inferred by the `hostname` and `service` argument. Otherwise, we will set the fields in `hints` to restrict the choice. For example, we can specify the type of socket in `ai_socktype`: either to use `SOCK_STREAM`, when the socket will communicate a continuous and reliable byte stream (the one used by this tutorial); or to use `SOCK_DGRAM`, when the socket will communicate a best-effort message stream (we use it in later tutorials). Other than the socket type, one sometimes want to set `ai_family` to restrict to use either IPv4 (using `PF_INET`) or IPv6 (using `PF_INET6`). One might also want to set `ai_protocol`, which selects a protocol to use (e.g., `IPPROTO_TCP`). Finally, one might want to set `ai_flags`, which is the bitwise-or of some of the following values:

- `AI_CANONNAME`: Asks the function to obtain the “canonical name” of the computer specified in `hostname`.
- `AI_NUMERICHOST`: Asks the function not to perform any domain name lookup (which might require some time). Instead if such a name is received the function fails.
- `AI_PASSIVE`: Creates an address for server use in case `hostname` is `NULL`.

If any of the fields are not needed, they should be set to contain 0. So it is customary to set the `hints` to all zero before starting to set values in it. For example:

```
struct addrinfo *res, hints;
memset(&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_STREAM;
int ret = getaddrinfo("virtue.csis.hku.hk", "1234", &hints, &res);
if (ret != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
    exit(1);
}
```

The `getaddrinfo()` function is not a system call, so it does not follow the convention that an error always produces -1. Instead, normally it returns zero, and on error it returns a negative error code. The function `gai_strerror()` may be used to get a human-readable string from the error code.

If the function returns successfully, it allocates a linked list to contain all usable addresses in `*res_ptr`. The linked list is linked using the `ai_next` pointer within the `addrinfo` structure. Apart from the `ai_family`, `ai_socktype` and `ai_protocol` fields, each element in the linked list also has an address stored in the `ai_addr` element. Its type, `struct sockaddr`, is not “real”, since socket support many different network protocol each having a different address format. For example, for Internet communication the actual type is `struct sockaddr_in`, defined in `<netinet/in.h>`. The `ai_addrlen` specifies its length which is useful in later system calls like `connect()` and `bind()`. Finally, the `ai_canonname` field is filled with the canonical name of the host if so requested with `AI_CANONNAME`.

At the end one should deallocate the linked list by the following function:

```
void freeaddrinfo(struct addrinfo *res);
```

## 2. Creating sockets and making connections

Once you get the `addrinfo` structure, you are ready to create a socket, and connect to the specified server. These are done using the following system calls, with the arguments plugged from the `addrinfo` structure.

```
int socket(int family, int socktype, int protocol);  
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

The return value of `socket()` is a file descriptor for use in the `sockfd` argument in later system calls. On the other hand, `connect()` simply returns 0 unless there is an error.

We might get many address structures from `getaddrinfo()`. Our program should try them one by one until succeeding. For example:

```
... /* Get a linked list of addrinfo in res with code above */  
int sockfd = -1;  
struct addrinfo *curr;  
for (curr = res; curr != 0; curr = curr->ai_next) {  
    sockfd = socket(curr->ai_family, curr->ai_socktype, curr->ai_protocol);  
    if (sockfd == -1) {  
        /* notify error */;  
        continue;  
    }  
    if (connect(sockfd, curr->ai_addr, curr->ai_addrlen) == -1)  
        /* notify error */;  
    else  
        break;  
    close(sockfd);  
    sockfd = -1;  
}  
if (sockfd == -1)  
    /* handle error */  
freeaddrinfo(res);
```

### 3. Closing the connection

The standard `close()` system call is used to close a connection. Once `close()` is called (which would return immediately), the program can no longer `read()` and `write()` using the socket. However, if there are data that are written already but not yet sent, it will continue to try sending the data, and terminate the connection only after that (or after timeout). If the remote end tries to send data to the socket 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). This looks very much like pipes.

### 4. Preparing to receive a connection

Passively waiting for connections (used by servers) is quite different from actively making connections (used by clients). The following system calls setup a socket for “passive open”.

```
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. We can create such addresses using `getaddrinfo()`, with `hostname=NULL` and `hints->ai_flags=AI_PASSIVE`.

The system call `listen()` allocates a table of connections. The `backlog` argument specifies the number of table entries in the table. 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).

The following shows how a server would setup a socket for passive open.

```
struct addrinfo *res, hints, *curr;  
memset(&hints, 0, sizeof(hints));  
hints.ai_socktype = SOCK_STREAM;  
hints.ai_flags = AI_PASSIVE;  
getaddrinfo(NULL, "1234", &hints, &res); /* should also handle error */  
int sockfd = -1;  
for (curr = res; curr != 0; curr = curr->ai_next) {  
    sockfd = socket(curr->ai_family, curr->ai_type, curr->ai_protocol);  
    if (sockfd == -1)  
        continue;  
    if (bind(sockfd, curr->ai_addr, curr->ai_addrlen) == -1)  
        /* notify error */;  
    else if (listen(sockfd, 5) == -1)  
        /* notify error */;  
    else  
        break;  
    close(sockfd);  
    sockfd = -1;  
}  
/* handle error, free linked list, etc */;
```

## 5. Waiting for a connection

Servers wait for connections by using `accept()`.

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

Once `accept()` completes, a **new** file descriptor is returned for the connection created. So we get **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 the same local address<sup>1</sup>. When a connection arrives, we might want to find the remote address. This can be done by using the `addr` and `addrlen_ptr` arguments. See the man pages `accept(2)` and `getnameinfo(3)` for details. We normally don't need them, so we can set `addr` and `addrlen_ptr` to `NULL`.

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

```
for (;;) {
    int newfd = accept(sockfd, NULL, NULL);
    /* read and write using newfd */
    close(newfd);
}
```

## 6. Concurrent server

An iterative server is good if the duration of all connections are short. Otherwise other clients need to wait for too long, since no connection is accepted when the server is busy talking with another client. In these cases, another process or thread can be created to handle the connection, while the “main” process or thread `accept()` again. This can be done by on-demand forking:

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

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.

---

<sup>1</sup>However, 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. More about this in the coming tutorials.

## 7. Controlling socket operations

Sometimes the default behaviour of a TCP connection is not what you want. In some case you might be able to get the desired behaviour using a TCP/IP options. They 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, some OS disallows a port with previous connection to be reused for around 2 minutes to prevent old data from interfering with the new one. But you probably don't want to wait 2 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:

```
int val = 1;  
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
```

We need to set this option in the version of Linux that we use in our lab, although in newer versions of Linux this setting is the default.