COMS 3157: Advanced Programming
Networking and Sockets

May 3, 2025

Patrick Shen

pts2125@columbia.edu

---

1. What are the steps in order to send and recieve data from a client to a server?

Ans:

(a) create client socket

(b) bind client socket

(c) connect to server

(d) send data to server

(e) receive data from server

2. Explain the each parameter in socket(int domain, int type, int protocol) (3 marks)

   (a) int domain

   (b) int type

   (c) int protocol

Ans:

(a) int domain: defines the address family commonly used in networking

- AF_INET - IPv4 (most commonly used)
- AF_INET6 - IPv6 (the future)
- AF_UNIX - UNIX domain
- AF_UNSPEC

(b) int type

- SOCK_STREAM - provides sequenced, reliable, two-way, connection-based byte streams.

- SOCK_DGRAM - supports datagrams (connectionless, unreliable messages of a fixed maximum length)
- SOCK_SEQPACKET - provides a sequenced, reliable, two-way, connection-based data transmission path for datagrams of fixed maximum length.

(c) int protocol

- UNSPEC: unspecified
- (AF_INET and SOCK_STREAM already implies TCP)

3. Syntax for establishing UDP/TCP sockets (2 marks)
   (a) What is the syntax to establish a UDP socket?
   (b) What is the syntax to establish a TCP socket?

Ans:

(a) socket(AF_INET, SOCK_DGRAM, 0)

(b) socket(AF_INET, SOCK_STREAM, 0)

4. What is the purpose of binding in networking (used in both UDP/TCP)?

Ans: The purpose of bind is to give a socket a networking address. This way, communication can be established between a client and a socket by referring to the socket's address. This procedure has traditionally been known as "assigning a name to a socket."

5. Explain the parameters of bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen) (1 mark)
   (a) int sockfd
   (b) struct *my_addr
   (c) socklen_t addrlen

Ans:

(a) int sockfd: socket descriptor

(b) struct *my_addr: points to a struct that represents an address on the network

(c) socklen_t addrlen: size in bytes of the struct pointed to by my_addr

6. Given that the `struct sockaddr` in the bind() method is designed to work for all networking connections i.e. outside of only just UDP and TCP, explain the significance of each field for `struct sockaddr` (1 mark)

   (a) sa_family_t sa_family

   (b) char sa_data[14]

```
struct sockaddr {
    sa_family_t sa_family; // unsigned short
    char sa_data[14]; // blob
}
```

Listing 1: struct sockaddr

Ans:

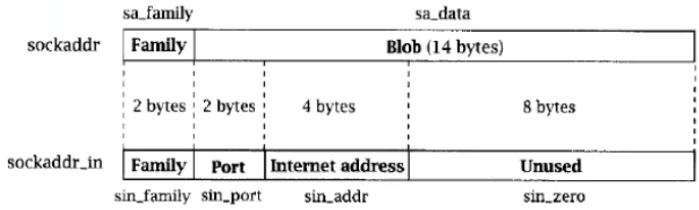
(a) sa_family_t sa_family: The address family (IPv4, IPv6, ...) used to bind a socket.

(b) char sa_data[14]: 14 bytes of data that is used to contain the more specific information of each individual networking protocol (TCP, UDP, ...) The purpose of this character array is to provide space.

7. Below is the syntax for delcaring a struct that holds information for constructing an IPv4 address. How can we pass this in as an argument for the aforementioned bind() function? (1 mark)

```
struct sockaddr_in {
    sa_family_t sin_family;   // unsigned short, address family:
   AF_INET
    in_port_t sin_port;       // unsigned short, port in network
   byte order
    struct in_addr sin_addir; // internet address
    char sin_zero[8];         // not used}
```

Listing 2: IPv4 sockaddr_in struct



Hint below!

Ans: We can cast the `struct sockaddr_in` (used in IPv4) as a `struct sockaddr` (struct to hold any networking protocol for bind() method). This is valid since they hold the same amount of bytes. This is a use of polymorphism in C.

8. If I gave you an 32-bit integer with value 1, write the C code to determine if the value is stored as little-endian or big-endian format. The integer is stored in a variable named `x`. (1 mark)

Ans:

```c
int main() {
    int x = 1;
    char *cp = (char *) &x; // cast is redundant, but used to show cp is of type char*
    if (*cp == 1) {
        printf("Little endian"); // first byte stores smallest integers
    } else {
        printf("Big endian"); // the 4 bytes are reversed in order
    }
    return 0;
}
```

Listing 3: little vs. big endian

9. Write C code to (2 marks)
   (a) convert little endian to big endian format for a short
   (b) convert big endian to little endian format for a short

Ans:

```
uint16_t little_to_big_endian(uint16_t little_endian) {
    return (little_endian >> 8) | (little_endian << 8)
}
```

Listing 4: little to big endian

```
uint16_t big_to_little_endian(uint16_t big_endian) {
    return (big_endian >> 8) | (big_endian << 8)
}
```

Listing 5: big to little endian

10. State whether network byte order and host byte order use big or little endian. (2 marks)

Ans:

(a) Network byte order uses **big-endian** format, where the most significant byte is stored first.

(b) Host byte order depends on the architecture. Most modern personal computers (e.g., x86 and x86-64 systems) use **little-endian** format, where the least significant byte is stored first.

11. State what the following functions are used for, and specify whether they will appear on server or client side code. (4 marks)

(a) `ntohs()`

(b) `htons()`

(c) `ntohl()`

(d) `htonl()`

Ans:

(a) `ntohs()` is used to convert a 16-bit value (short) from network byte order to host byte order. It typically appears on the **server side** when receiving port numbers or other 16-bit values from the client.

(b) `htons()` is used to convert a 16-bit value (short) from host byte order to network byte order. It usually appears on the **client side** when sending port numbers or preparing a socket structure before connecting.

(c) `ntohl()` is used to convert a 32-bit value (long) from network byte order to host byte order. It appears on the **server side** when receiving 4-byte values such as IPv4 addresses or other binary data.

(d) `htonl()` is used to convert a 32-bit value (long) from host byte order to network byte order. It appears on the **client side** when sending 32-bit values or setting fields like IP addresses before transmission.

12. What is the relationship between `htons()` and `noths()` in relation to big and little endianness? (1 mark)

Ans:

(a) `htons()` converts a short in host byte order (usually little endian) to network byte order (big endian)

(b) `ntohs()` converts a short in network byte order (big endian) to host byte order (usually little endian)

13. What function should go in the blank and why? (2 marks)

```
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
// Convert port to network byte order
server_addr.sin_port = __________;  // Port 8080
```

Listing 6: client side code

Ans:

(a) `htons(8080)`

(b) The bind() function expects the port number in network byte order, which is big-endian. Most systems (e.g., x86) are little-endian, so we convert with htons() (Host TO Network Short).

14. What function should go in the blank and why? (2 marks)

```
uint32_t net_val;
recv(sock, &net_val, sizeof(net_val), 0);  // received 4-byte
    integer
uint32_t host_val = __________;
```

Listing 7: server side code

Ans:

(a) `ntohl(net_val)`

(b) The 4-byte integer arrives in network byte order (big-endian), and we need to convert it to the host's native format. ntohl() stands for Network TO Host Long

15. Do you need to use a byte order conversion here? (2 marks)

```
uint32_t ip_host = inet_addr("192.168.0.1");  // host order
uint32_t ip_net = __________;
send(sock, &ip_net, sizeof(ip_net), 0);
```

Listing 8: client side code

Ans:

(a) No additional conversion needed.

(b) `inet_addr()` already returns the IP address in network byte order. If you call htonl() again, you will corrupt the order.

16. You're writing server code that receives a 2-byte port number from a client (sent in network byte order). What function should you use?

```
uint16_t port_net;
recv(sock, &port_net, sizeof(port_net), 0);
uint16_t port_host = __________;
```

Listing 9: server side code

Ans:

(a) `ntohs(port_net)`

(b) The port number sent over the network is in network byte order (big-endian). To use it on your local machine, convert it back using Network TO Host Short (ntohs()).

17. What does `connect()` do and explain each parameter field in `connect()` (2 marks)

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

(a) `sockfd`

(b) `addr`

(c) `addrlen`

Ans:

(a) The `connect()` system call is used by a client to initiate a connection to a server. It is commonly used in socket programming to establish a connection to a remote host.

(b) `sockfd`: The file descriptor of the socket, returned by `socket()`. It identifies the local endpoint of the connection.

(c) `addr`: A pointer to a `sockaddr` structure that specifies the address (IP and port) of the remote host to connect to. It is usually cast from a more specific structure like `struct sockaddr_in`.

(d) `addrlen`: The size (in bytes) of the address structure pointed to by `addr`.

If the connection succeeds, `connect()` returns 0. On failure, it returns -1 and sets `errno` to indicate the error.

```
1    int sock;
2     struct sockaddr_in echoserver;
3     sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
4     memset(&echoserver, 0, sizeof(echoserver)); /* Clear struct
     */
5     echoserver.sin_family = AF_INET; /* Internet/IP */
6     echoserver.sin_addr.s_addr = inet_addr(argv[1]); /* IP
     address */
7     echoserver.sin_port = htons(atoi(argv[3])); /* server port */
8     connect(sock, (struct sockaddr *)&echoserver, sizeof(
     echoserver));
```

Listing 10: connect() setup

18. What does `listen()` do and explain each parameter field in `listen()` (2 marks)

`int listen(int sockfd, int backlog)`

(a) sockfd

(b) backlog

Ans:

(a) `sockfd`: The file descriptor of a socket that has been created with `socket()` and bound to an address with `bind()`. This socket will be marked as a passive socket, meaning it will be used to accept incoming connection requests using `accept()`.

(b) `backlog`: Defines the maximum number of pending connections that can be queued up before the kernel starts rejecting new ones. If too many connection requests arrive when the queue is full, the client may receive a connection refusal.

19. What does `accept()` do and explain each parameter field in `accept()` (2 marks)

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

(a) sockfd

(b) addr

(c) addrlen

Ans:

(a) `sockfd`: The file descriptor of a socket that has been set up with `socket()`, `bind()`, and `listen()`. This socket must be in a listening state. `accept()` will block until a client initiates a connection to this socket.

(b) `addr`: A pointer to a `sockaddr` structure where the address information of the connecting client will be stored. This allows the server to learn the client's IP address and port.

(c) `addrlen` A pointer to a variable that initially contains the size of the `addr` structure. When `accept()` returns, this variable will be updated with the actual size of the address returned.

20. What does `recv()` do and explain each parameter field in `recv()`? (2 marks)

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

(a) `sockfd`

(b) `buf`

(c) `len`

(d) `flags`

Ans:

(a) The `recv()` system call is used to receive data from a connected socket. It reads incoming data from the socket's receive buffer into a specified memory buffer.

(b) `sockfd`: The file descriptor of the connected socket from which data is to be received.

(c) `buf`: A pointer to a memory buffer where the received data will be stored.

(d) `len`: The maximum number of bytes to read into the buffer.

(e) `flags`: Optional flags to modify the behavior of the call (e.g., `MSG_PEEK`, `MSG_WAITALL`).

On success, `recv()` returns the number of bytes received. If the connection is closed, it returns 0. On error, it returns -1 and sets `errno`.

21. What does `listen()` do and explain each parameter field in `listen()` (2 marks)

```
int listen(int sockfd, int backlog)
```

(a) sockfd

(b) backlog

Ans:

(a) `sockfd`: The file descriptor of a socket that has been created with `socket()` and bound to an address with `bind()`. This socket will be marked as a passive socket, meaning it will be used to accept incoming connection requests using `accept()`.

(b) `backlog`: Defines the maximum number of pending connections that can be queued up before the kernel starts rejecting new ones. If too many connection requests arrive when the queue is full, the client may receive a connection refusal.

22. What does `select()` do and explain each parameter field (4 marks)

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout)
```

(a) int nfds

(b) fd_set *readfds

(c) struct timeval *timeout

Ans:

(a) What `select()` does: monitors activity for list of file descriptors (blocking call) and returns the list of file descriptors that have activity (by return, the function modifies the input set of file descriptors to watch for, and returns -1 on error)

(b) int nfds: the max value + 1 for the list of file descriptors you want to check (usually set to macro FD_SETSIZE)

(c) fd_set *readfds: set of file descriptors to watch for any read() activity.

(d) struct timeval *timeout: information containing how long select() should block process. If set to `NULL`, then blocks forever.

23. Why is `select()` considered a *destructive* function? (1 mark)

Ans: `select()` modifies the fd_set structures that are passed into the function, only returning the file descriptors that had activity on them. For example, if only sock1 had `read()` activity but sock2 does not,

```
int main() {
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(sock1, &readfds);
    FD_SET(sock2, &readfds);
    select(maxfd + 1, &readfds, NULL, NULL, NULL);
}
```

Listing 11: select() destructive

then the fd_set readfds will only contain sock1 after it has executed, removing sock2 from the set.

Thus in an indefinite while loop, you should reset the fd_set passed into `select()`

```
int main() {
    fd_set readfds, readfds_copy;
    FD_ZERO(&readfds);
    FD_SET(sock1, &readfds);
    FD_SET(sock2, &readfds);
  while (1) {
    FD_ZERO(&readfds_copy);
    readfds_copy = readfds;
```

```
9       select(maxfd + 1, &readfds_copy, NULL, NULL, NULL);
10      }
11  }
```

Listing 12: select() destructive

24. Write a complete C code block that performs the following steps in a TCP client:

    - Creates a socket using IPv4 and TCP.
    - Initializes a `sockaddr_in` struct with an IP address and port from `argv[1]` and `argv[2]`.
    - Connects the socket to the server.
    - Sends the message `argv[3]` to the server.
    - Receives a response into a buffer.

    Assume `BUFFSIZE` is predefined. You may use `inet_addr()`, `htons()`, `strlen()`, and standard system calls like `socket()`, `connect()`, `send()`, and `recv()`. Do not include the full `main()` function or headers.

Ans:

```c
int main(int argc, char** argv) {
    int client_sock;
    struct sockaddr_in server_addr;
    char recv_buffer[BUFFSIZE];
    unsigned int msg_len;
    int bytes_received = 0;

    client_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(argv[1]);
    server_addr.sin_port = htons(atoi(argv[2]));

    connect(client_sock, (struct sockaddr *)&server_addr, sizeof(
   server_addr));

    msg_len = strlen(argv[3]);
    send(client_sock, argv[3], msg_len, 0);

    recv(client_sock, recv_buffer, BUFFSIZE - 1, 0);
}
```

Listing 13: client code

25. Write a complete C code block that performs the following steps in a TCP server:

    - Creates a socket using IPv4 and TCP.
    - Binds the socket to port `argv[1]` on all local interfaces.
    - Listens for incoming client connections with a maximum pending queue.
    - Accepts a client connection.
    - Receives data into a buffer and echoes it back to the client.

    Assume `BUFFSIZE` and `MAXPENDING` are predefined. You may use system calls such as `socket()`, `bind()`, `listen()`, `accept()`, `recv()`, and `send()`, along with `htonl()`, `htons()`, and `memset()`. Do not include the full `main()` function or headers.

Ans:

```c
int main(int argc, char** argv) {
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    char buffer[BUFFSIZE];
    socklen_t client_len = sizeof(client_addr);
    int received;

    server_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(atoi(argv[1]));

    bind(server_sock, (struct sockaddr *)&server_addr, sizeof(
   server_addr));
    listen(server_sock, MAXPENDING);

    client_sock = accept(server_sock, (struct sockaddr *)&
   client_addr, &client_len);

    received = recv(client_sock, buffer, BUFFSIZE, 0);
    send(client_sock, buffer, received, 0);
}
```

Listing 14: server code

26. Write a C code block that uses `select()` to implement a concurrent TCP server. Your code should:

    - Loop through all file descriptors to check for incoming connections or data.
    - Accept new connections on the listening socket and add them to the active file descriptor set.
    - Handle input on already-connected sockets using a function `read_from_client(int fd)`.
    - Remove a socket from the set and close it if the client disconnects.

    Assume the sets `active_fd_set` and `read_fd_set` are already declared, and that `sock` is the listening socket. The function `read_from_client()` is already defined. Do not write the full `main()` function or include headers.

Ans:

```c
int main(int argc, char** argv) {
    for (int i = 0; i < FD_SETSIZE; ++i) {
        if (FD_ISSET(i, &read_fd_set)) {
            if (i == sock) {
                // New connection on listening socket
                int new_fd;
                struct sockaddr_in client_addr;
                socklen_t size = sizeof(client_addr);

                new_fd = accept(sock, (struct sockaddr *)&client_addr, &size);
                if (new_fd < 0) {
                    perror("accept");
                    exit(EXIT_FAILURE);
                }

                fprintf(stderr, "Server: connect from host %s, port %d.\n",
                        inet_ntoa(client_addr.sin_addr),
                        ntohs(client_addr.sin_port));

                FD_SET(new_fd, &active_fd_set);
            } else {
                // Existing client sent data
                if (read_from_client(i) < 0) {
                    close(i);
                    FD_CLR(i, &active_fd_set);
                }
```

```
                }
            }
}
}
```

Listing 15: concurrent server code