

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 receive 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]`

```
1 struct sockaddr {  
2     sa_family_t sa_family; // unsigned short  
3     char sa_data[14]; // blob  
4 }
```

Listing 1: struct sockaddr

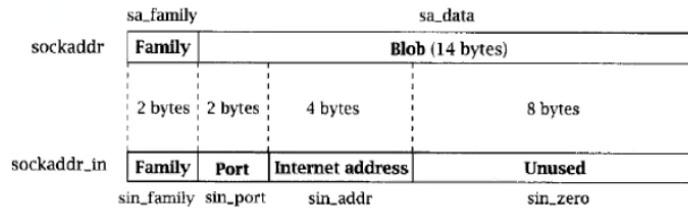
Ans:

- (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 declaring 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)

```
1 struct sockaddr_in {  
2     sa_family_t sin_family; // unsigned short, address family:  
3     AF_INET  
4     in_port_t sin_port; // unsigned short, port in network  
5     byte order  
6     struct in_addr sin_addr; // internet address  
7     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:

```

1 int main() {
2     int x = 1;
3     char *cp = (char *) &x; // cast is redundant, but used to
4     show cp is of type char*
5     if (*cp == 1) {
6         printf("Little endian"); // first byte stores smallest
7         integers
8     } else {
9         printf("Big endian"); // the 4 bytes are reversed in
10        order
11    }
12    return 0;
13 }
```

Listing 3: little vs. big endian

9. Write C code to (2 marks)
- convert little endian to big endian format
 - convert big endian to little endian format

Ans:

```
1  uint16_t little_to_big_endian(uint16_t little_endian) {  
2      return (little_endian >> 8) | (little_endian << 8)  
3  }
```

Listing 4: little to big endian

```
1  uint16_t big_to_little_endian(uint16_t big_endian) {  
2      return (big_endian >> 8) | (big_endian << 8)  
3  }
```

Listing 5: big to little endian

10. What is `htons()` and `noths()` used for?

(2 marks)

Ans:

Ans: `htons()` is used to convert a port number to a 2 byte short so that a socket client can connect to a specific port on an address. For example, if I wanted to connect to port 80, I would first need to convert 80 into a 2 byte short using `htons(80)` which would return a big endian IP address.

`ntohs()` is the opposite function in that it takes a big endian IP address and spits out the port number in decimal 10 format. For example, `ntohs(168.192.101.100)` could be converted into 40.

11. What is the relationship between `htons()` and `noths()` in relation to bit and little endianness?

(1 mark)

12. 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  */  
6  echoserver.sin_family = AF_INET; /* Internet/IP */  
7  echoserver.sin_addr.s_addr = inet_addr(argv[1]); /* IP  
8  address */  
9  echoserver.sin_port = htons(atoi(argv[3])); /* server port */  
10 connect(sock, (struct sockaddr *)&echoserver, sizeof(  
11 echoserver));
```

Listing 6: `connect()` setup

13. 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.

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

```
int accept(int sockfd, struct sockaddr *addr, socklen_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.

15. 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**.

16. 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.

17. 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.

18. 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,

```
1 int main() {
2     fd_set readfds;
3     FD_ZERO(&readfds);
4     FD_SET(sock1, &readfds);
5     FD_SET(sock2, &readfds);
6     select(maxfd + 1, &readfds, NULL, NULL, NULL);
7 }
```

Listing 7: `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()`

```
1 int main() {
2     fd_set readfds, readfds_copy;
3     FD_ZERO(&readfds);
4     FD_SET(sock1, &readfds);
5     FD_SET(sock2, &readfds);
6     while (1) {
7         FD_ZERO(&readfds_copy);
8         readfds_copy = readfds;
9         select(maxfd + 1, &readfds_copy, NULL, NULL, NULL);
10    }
```

```
11 }
```

Listing 8: select() destructive

19. 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 `BUFSIZE` 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:

```
1 int main(int argc, char** argv) {
2     int client_sock;
3     struct sockaddr_in server_addr;
4     char recv_buffer[BUFSIZE];
5     unsigned int msg_len;
6     int bytes_received = 0;
7
8     client_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
9
10    memset(&server_addr, 0, sizeof(server_addr));
11    server_addr.sin_family = AF_INET;
12    server_addr.sin_addr.s_addr = inet_addr(argv[1]);
13    server_addr.sin_port = htons(atoi(argv[2]));
14
15    connect(client_sock, (struct sockaddr *)&server_addr, sizeof(
server_addr));
16
17    msg_len = strlen(argv[3]);
18    send(client_sock, argv[3], msg_len, 0);
19
20    recv(client_sock, recv_buffer, BUFSIZE - 1, 0);
21 }
```

Listing 9: client code

20. 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 `BUFSIZE` 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:

```
1 int main(int argc, char** argv) {
2     int server_sock, client_sock;
3     struct sockaddr_in server_addr, client_addr;
4     char buffer[BUFSIZE];
5     socklen_t client_len = sizeof(client_addr);
6     int received;
7
8     server_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
9
10    memset(&server_addr, 0, sizeof(server_addr));
11    server_addr.sin_family = AF_INET;
12    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
13    server_addr.sin_port = htons(atoi(argv[1]));
14
15    bind(server_sock, (struct sockaddr *)&server_addr, sizeof(
server_addr));
16    listen(server_sock, MAXPENDING);
17
18    client_sock = accept(server_sock, (struct sockaddr *)&
client_addr, &client_len);
19
20    received = recv(client_sock, buffer, BUFSIZE, 0);
21    send(client_sock, buffer, received, 0);
22 }
```

Listing 10: server code

21. 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:

```

1 int main(int argc, char** argv) {
2     for (int i = 0; i < FD_SETSIZE; ++i) {
3         if (FD_ISSET(i, &read_fd_set)) {
4             if (i == sock) {
5                 // New connection on listening socket
6                 int new_fd;
7                 struct sockaddr_in client_addr;
8                 socklen_t size = sizeof(client_addr);
9
10                new_fd = accept(sock, (struct sockaddr *)&
client_addr, &size);
11                if (new_fd < 0) {
12                    perror("accept");
13                    exit(EXIT_FAILURE);
14                }
15
16                fprintf(stderr, "Server: connect from host %s,
port %d.\n",
17                        inet_ntoa(client_addr.sin_addr),
18                        ntohs(client_addr.sin_port));
19
20                FD_SET(new_fd, &active_fd_set);
21            } else {
22                // Existing client sent data
23                if (read_from_client(i) < 0) {
24                    close(i);
25                    FD_CLR(i, &active_fd_set);
26                }
27            }
28        }
29    }

```

```
30 }
```

Listing 11: concurrent server code