9. Sockets

Sockets can be used to communicate between processes on different computer. The IP Address of a machine is the address of a machine that can be used to send messages to it over the internet. The port specifies which process should receive that message. A packet contains the address and the data. Port numbers range from `0` to `65535`. However, the ports `0` to `1023` are reserved for well-known services. The ports `1024` to `49151` should be registered for use for public services. Other ports can be freely used. `127.0.0.1` is the local host address which refers to the local machine.

Webpages are typically served at port 80 and secure web pages at port 443.

Stream sockets are built on the TCP Protocol and guarantee that no data will be lost and that the order in which the data is sent will be maintained.

System Calls and Library Functions

`socket`

The prototype of `socket` is `int socket(int domain, int type, int protocol);`

`socket` creates an endpoint for communication and returns a descriptor. `domain` defines the communication domain used for communication. This can be set to either of `PF_INET` or `AF_INET`. `type` defines the socket type. This can be set to `SOCK_STREAM` for stream sockets. `protocol` defines the protocol family used. This can be set to `0` to indicate that the default protocol, which is TCP, should be used.

A stream socket must be in a connected state before any data may be sent or received on it. Once connected, data may be transferred using `read` and `write` system calls. When a session has been completed, `close` may be called.

`-1` is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

`struct sockaddr_in`

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
```

```
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

`sin_family` should be set to `AF_INET` to b consistent with our `socket` call configurations. `sin_port` should store the port of the process. `htons` needs to be used to make sure the byte order is correct. `sin_addr.in_addr` should be set to `INADDR_ANY` so that the socket can accept connections from any address. The `sin_zero` field pads out the struct so that it has the same size as the `sockaddr` struct. `memset` should be used to set these bytes to `0`.

`bind`

The prototype of `bind` is `int bind(int socket, const struct sockaddr *address, socklen_t address_len);`

`bind` assigns a name to an unnamed socket. When a socket is created with `socket` it exists in a name space (address family) but has no name assigned. `bind` requests that address be assigned to the socket. Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using `unlink`).

`socket` is the file descriptor of the socket. `address` is the generic struct for all address families. The struct `sockaddr_in` needs to be used for the TCP Protocol. `addres_len` is the length of the address being passed. This should be `sizeof(struct sockaddr_in)`.

`0` is returned on success and `-1` is returned on failure and `errno` is set.

`listen`

The prototype of `listen` is `int listen(int socket, int backlog);`

`listen` specifies the willingness to accept incoming connections and a queue limit for incoming connections. `socket` defines the file descriptor of the socket and `backlog` defines the maximum length for the queue of pending connections. If a connection request arrives with the queue full, the client may receive an error. Alternatively, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

`listen` applies only to sockets of type `SOCK_STREAM`.

`0` is returned on success and `-1` is returned on failure and `errno` is set.

`accept`

The prototype of `accept` is `int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`

`socket` is a socket that has been created with `socket`, bound to an address with `bind`, and is listening for connections after a `listen`. `accept` extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of socket, and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `accept` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `accept` returns an error. The accepted socket may not be used to accept more connections. The original socket socket, remains open.

`address` is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. Only the `sin_family` parameter needs to be set.

`address_len` is a value-result parameter; it should initially contain the amount of space pointed to by address; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

It is possible to `select` a socket for the purposes of doing an `accept` by selecting it for read.

`listen` returns `-1` on error and sets `errno`. Otherwise, it returns the file descriptor for the accepted socket.

`connect` and `getaddrinfo`

The prototype of `getaddrinfo` is `int getaddrinfo(const char *hostname, const char *servname, const struct addrinfo *hints, struct addrinfo **res);`

`getaddrinfo` is used to get a list of IP addresses and port numbers for host `hostname` and service `servname`. `hostname` can be set to the string representing the name of the machine on which the code is running. The parameters `servname` and `hints` can be set to `NULL`. `res` is a linked list of structs contains information about the valid addresses. The `sin_addr` field of the `ai_addr` field of `res` can be used to initialise the `sin_addr` field of the `sockaddr_in` struct. `freeaddrinfo` should be called to deallocate the memory used by this linked list.

`0` is retuned on success and a non-zero error code on failure.

The prototype of `connect` is `int connect(int socket, const struct sockaddr *address, socklen_t address_len);`

`socket` is the file descriptor of the socket. `address` is the socket to which `connect` attempts to connect to. `address_len` is the size of the address.

`0` is returned on success and `-1` is returned on failure and `errno` is set.

`htons`, `htonl`, `ntohl`

The prototype for `htons` is `uint16_t htons(uint16_t hostshort);`
The prototype for `htonl` is `uint32_t htonl(uint32_t hostlong);`
The prototype for `ntohl` is `uint32_t ntohl(uint32_t netlong);`

These routines convert 16 bit, 32 bit, and 64 bit quantities between network byte order and host byte order. Network byte order is big endian, or most significant byte first. On machines which have a byte order which is the same as the network order, routines are defined as null macros.

Sample Code

Client

```
int main() {
    // create socket
    int soc = socket(AF_INET, SOCK_STREAM, 0);
    if (soc == -1) {
        perror("client: socket");
        exit(1);
```

```
    }

    //initialize server address
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(54321);
    memset(&server.sin_zero, 0, 8);

    struct addrinfo *ai;
    char * hostname = "teach.cs.toronto.edu";

    /* this call declares memory and populates ailist */
    getaddrinfo(hostname, NULL, NULL, &ai);
    server.sin_addr = ((struct sockaddr_in *) ai->ai_addr)->sin_addr;

    // free the memory that was allocated by getaddrinfo for this
list
    freeaddrinfo(ai);

    int ret = connect(soc, (struct sockaddr *)&server, sizeof(struct
sockaddr_in));
    if (ret == -1) {
        perror("client: connect");
        exit(1);
    }

    printf("Connect returned %d\n", ret);

    char buf[10];
    read(soc, buf, 7);
    buf[7] = '\0';
    printf("I read %s\n", buf);

    write(soc, "0123456789", 10);
    return 0;
}
```

Server

```
int main() {
    // create socket
    int listen_soc = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_soc == -1) {
        perror("server: socket");
```

```
        exit(1);
    }

    //initialize server address
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(54321);
    memset(&server.sin_zero, 0, 8);
    server.sin_addr.s_addr = INADDR_ANY;

    // bind socket to an address
    if (bind(listen_soc, (struct sockaddr *) &server, sizeof(struct
sockaddr_in)) == -1) {
      perror("server: bind");
      close(listen_soc);
      exit(1);
    }

    // Set up a queue in the kernel to hold pending connections.
    if (listen(listen_soc, 5) < 0) {
        // listen failed
        perror("listen");
        exit(1);
    }

    struct sockaddr_in client_addr;
    unsigned int client_len = sizeof(struct sockaddr_in);
    client_addr.sin_family = AF_INET;

    int client_socket = accept(listen_soc, (struct sockaddr
*)&client_addr, &client_len);
    if (client_socket == -1) {
        perror("accept");
        return -1;
    }

    write(client_socket, "hello\r\n", 7);

    char line[10];
    read(client_socket, line, 10);
    /* before we can use line in a printf statement, ensure it is a
string */
    line[9] = '\0';
    printf("I read %s\n", line);
```

```
    return 0;
}
```