Sockets Reference Sheet

Stefan Guna, guna@disi.unitn.it

TCP sockets

The sequence for establishing a TCP connection is depicted in Figure 1. We explain each step from the picture.

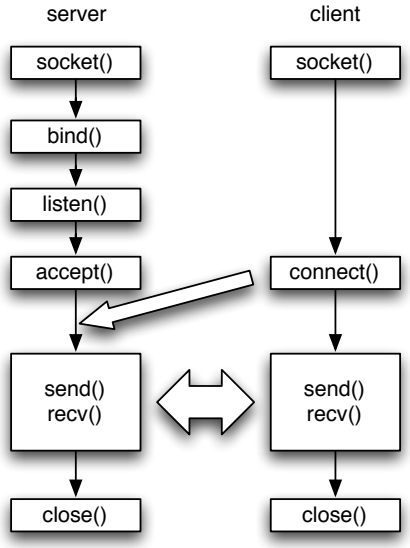

Figure 1

Creating the socket

This step is common for both client and server. First, the client/server must create a TCP socket using the following function call:

```
int socket_fd =
    socket(AF_INET, SOCK_STREAM,
           IPPROTO_TCP);
```

The first parameter (AF_INET) denotes that the socket is used for an Internet protocol. Other types of protocols exists, but are rarely used.

The second parameter (SOCK_STREAM) indicates that a stream will flow through the socket. Please note sockets can be used for lower level protocols (SOCK_RAW).

The last parameter (IPPROTO_TCP) denotes the IP protocol that the socket uses.

The socket address

The client can obtain the IP address of the server by performing a DNS lookup. The function is:

```
struct hostent *server_host = gethostbyname("www.google.com");
```

After the IP address is obtained, the client must fill in a structure representing a IP socket address: a pair (IP address,port). The following is the declaration of the structure:

```
struct sockaddr_in {
    short           sin_family;   // e.g. AF_INET
    unsigned short  sin_port;     // e.g. htons(3490)
    struct in_addr  sin_addr;     // see struct in_addr, below
    char            sin_zero[8];  // not used
};

struct in_addr {
    unsigned long s_addr;
};
```

The first member (sin_family) must be set to AF_INET, which denotes that the structure denotes an IPv4 address format. IPv6 and Ethernet addresses can be used.

The second member (sin_port) represents the port. Please note that **the port must be specified in network byte order**. To convert a 2 byte integer (i.e. short) to network byte order, `htons` can be used.

The last member (sin_addr) is a structure containing one field. This must be filled with the IP address (also in **network byte order**):

- If the IP address is found from a lookup using `gethostbyname`. In this case `server_host->h_addr` can be used. No conversion to the network byte order is required.
- If no lookup is performed and you know the IP address, use `inet_addr` which converts a string (i.e. "127.0.0.1") to a format that can be used directly in the sockaddr structure.

Connecting to a server

The client connects to a server using the following function call:

```
connect(socket_fd, struct sockaddr_in *your_structure,
    sizeof(struct sockaddr_in));
```

Server waiting for connection

First, the server must setup a port to listen on. `sockaddr_in` must be used to specify the address and the port number to use. Example:

```
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(2000); // remember, in network byte order
server_addr.sin_port = htonl(INADDR_ANY); // htonl instead of htnos
```

Then, the socket must be bind to this address and the listening operation can be started:

```
bind(socket_fd, (struct sockaddr *) &server_addr, sizeof(server_addr));
listen(socket_fd, 10);
```

All clients who are trying to connect are put in a waiting queue. The size of the queue is configured by the second parameter of listen. To remove a client form the waiting queue and establish the connection, use the following call:

```
struct sockaddr_in client_addr;
int client_addr_len = sizeof(client_addr);
int new_client =
    accept(socket_fd, (struct sockaddr *) &client_addr, &client_addr_len);
```

The function blocks until a client connects. A new socket is created for each incoming client. The function returns the client address in the second parameter (client_addr) and the size of the address in the third parameter (client_addr_len). The size is required because different protocols can have different address sizes. The size of the structure client_addr must be passed to the function.

Exchanging data

The following blocking functions can be used to exchange messages by both the server and client:

```
ssize_t send(int socket, const void *buffer, size_t length, int flags);
ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

Use 0 for the last parameter (flags).

As alternative, you can use `write` and `read` to send and receive data through stream sockets as you were using a file.

UDP sockets

Differently from TCP sockets, the UDP protocol does not require a connection to be established prior to sending any data. Hence, as you can see from Figure 2, the calls to listen, accept, connect are no longer used.

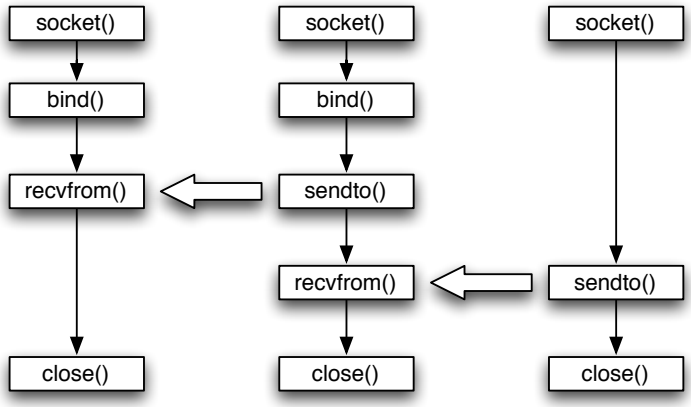

Figure 2

Creating the socket

Similarly to TCP, UDP sockets are created using the following function call:

```
int socket_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

Enabling a socket to receive

As already mentioned, UDP is a connection-less protocol. Hence, to listen / accept / connect is required. However, when sending data, one still needs to know the IP address and the UDP port of the recipient; furthermore the recipient must know that he is receiving data on a certain port.

For this, one must call `bind`, just in the case of TCP sockets.

Exchanging data

Because there is no connection established over UDP sockets, one must specify the recipient of every packet sent. The function used to send UDP packets (called datagrams) is the following:

```
int sendto(int socket, void *buffer, size_t length, int flags,
       struct sockaddr *dest_addr, socklen_t dest_len);
```

where `flags` should be set to 0, `dest_addr` must be set to the address of the recipient (see "The socket address") and `dest_len` must be set to `sizeof(struct sockaddr_in)`.

One can receive datagrams using the following function:

```
int recvfrom(int socket, void *buffer, size_t length, int flags,
       struct sockaddr *address, socklen_t *address_len);
```

where the last two parameters are filled in with the address of the sender.