

Client software design

Stefan D. Bruda

CS 464/564, Fall 2023



- 1 Get the IP address and port number of the peer
- 2 Allocate a socket
- 3 Choose a local IP address
- 4 Allow TCP to choose an arbitrary, unused port number
- 5 Connect the socket to the server
- 6 Communicate with the server
 - Exchange messages
 - Often the client sends requests and the server replies, but **this is not always the case**
 - **The message exchange happens according to the application-level protocol**
- 7 Close connection



- Depending on the actual application, the IP address of the peer (i.e., server) can be specified in more than one ways, including:
 - Hardcoded (rarely)
 - We specify it directly as an integer
 - As command-line argument (read from configuration file, etc.)
 - We use `gethostbyname` to get the actual address (i.e., number)
 - Use a separate protocol (broadcast or multicast) to find a server
- Ports can also be specified in many ways, including:
 - It is a well-known port
 - We use `getservbyname` to obtain the actual port number
 - Hardcoded
 - Possibly suitable for custom client-server applications
 - As command-line argument (read from configuration file, etc.)
 - Especially useful for parameterized clients

```
telnet linux.ubishops.ca 22
my-client linux.ubishops.ca ssh
```



- We need to specify at allocation time:
 - The protocol family
 - The socket type (TCP for the time being)

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
int sd = socket(PF_INET, SOCK_STREAM, 0);
```

- We end up with a **socket descriptor**



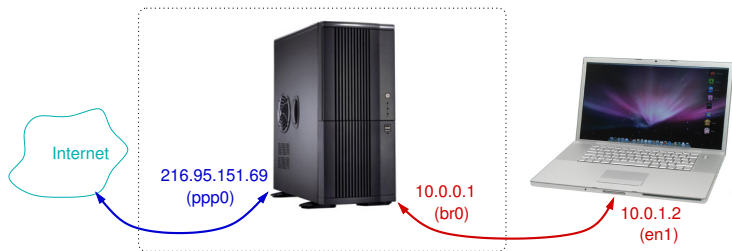
- Why do we need the local IP address?



- Why do we need the local IP address?
 - Because a connection is specified by **two** endpoints
- Why is it a problem to choose a local IP address?



- Why do we need the local IP address?
 - Because a connection is specified by **two** endpoints
- Why is it a problem to choose a local IP address?
 - Because a machine might have **multiple addresses**



- The appropriate address must be chosen so that IP is able to route packets in the right direction
- Choosing the right IP address is done after a dialogue with IP
- The system call `connect` does it for us



- We must specify a local port number for the same reasons we have to specify a local address
- The choice of port number does not matter as long as:
 - It does not conflict with the port assigned to a well-know service
 - It is not in use by another process
- We could try at random until we get a free port. . .
 - . . . However, the system keeps track of port usage anyway, so this would be overkill
 - Thus the port number choice is again taken care of by the call to `connect`



- In all, we obtain the local coordinates (IP address, port) and we connect in one step:

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

- Something like this:

```
struct sockaddr_in sin;  
int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));  
if (rc < 0) {  
    perror("connect");  
    exit(1);  
}
```



```
int connectbyport(int(const char* host, const unsigned short port) {
    struct hostent *hinfo;
    struct sockaddr_in sin;
    const int type = SOCK_STREAM;
    int sd;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL) return err_host;
    sin.sin_addr=(unsigned int)hinfo->h_addr;

    sin.sin_port = port;

    sd = socket(PF_INET, type, 0);
    if ( sd < 0 ) return err_sock;

    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```



FILLING IN THE SERVER ADDRESS

```
int connectbyport(int(const char* host, const unsigned short port) {
    struct hostent *hinfo;
    struct sockaddr_in sin;
    const int type = SOCK_STREAM;
    int sd;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL) return err_host;
    sin.sin_addr=(unsigned int)hinfo->h_addr; // only if you are lucky

    sin.sin_port = port; // only if you are lucky

    sd = socket(PF_INET, type, 0);
    if ( sd < 0 ) return err_sock;

    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```



FILLING IN THE SERVER ADDRESS

```
int connectbyport(int(const char* host, const unsigned short port) {
    struct hostent *hinfo;
    struct sockaddr_in sin;
    const int type = SOCK_STREAM;
    int sd;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL) return err_host;
    sin.sin_addr=(unsigned int)htonl(hinfo->h_addr); // assumes a bit too much

    sin.sin_port = (unsigned short)htons(port);

    sd = socket(PF_INET, type, 0);
    if ( sd < 0 ) return err_sock;

    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```



FILLING IN THE SERVER ADDRESS

```
int connectbyport(int(const char* host, const unsigned short port) {
    struct hostent *hinfo;
    struct sockaddr_in sin;
    const int type = SOCK_STREAM;
    int sd;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL) return err_host;
    memcpy(&sin.sin_addr, hinfo->h_addr, hinfo->h_length);

    sin.sin_port = (unsigned short)htons(port);

    sd = socket(PF_INET, type, 0);
    if ( sd < 0 ) return err_sock;

    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```



- We send data using `send` (or `write`)
- We receive responses using `recv` (or `read`)
 - Note that the response could come **in pieces**, even if the server answers back in large chunks
 - You should be prepared to accept data a few bytes at a time

```
const int ALEN = 128;
char* req = "some sort of request";
char ans[ALEN];
char* ans_ptr = ans;
int ans_to_go = ALEN, n = 0;

send(sd, req, strlen(req), 0);

while ( ( n = recv(sd, ans_ptr, ans_to_go, 0) ) > 0 ) {
    ans_ptr += n;
    ans_to_go -= n;
}
```



- We do not necessarily know how long is the response
- The shape of the response varies according to the application-level protocol and may be:
 - One line of text (terminated by `'\n'`)
 - We use `readline` (or equivalent) to read the answer
 - One line of text determines what comes after it
 - Again, we use `readline` to read one line at a time, and then decide what to do next based on what we just read
 - As much as the server cares to send, with no special end marker
 - We read until there is no more data
 - But how?



- We check whether we have any more data coming on our way
- Communication is not instantaneous, so we have to give some time for the data to arrive

```
const int recv_nodata = -2;
```

```
int recv_nonblock (int sd, char* buf, size_t max, int timeout) {  
    struct pollfd pollrec;  
    pollrec.fd = sd;  
    pollrec.events = POLLIN;  
  
    int polled = poll(&pollrec,1,timeout);  
    if (polled == 0) return recv_nodata;  
    if (polled == -1) return -1;  
    return recv(sd,buf,max,0);  
}
```

- Outcomes:

- **-2**: no more data available within the given **timeout**
- **0**: end of file (when the server closes connection on us)
- **$n > 0$** : n characters have been read.



- `close` closes the connection and destroys the socket
- Sometimes we want to shut down communication in **one direction only**
 - Reason: the server receive a request and responds to it
 - But what does it do now with the connection?
 - If the client has in fact more requests, the connection should stay open
 - If this is the last request, the connection should be closed
- A client (or server) can **partially close** a connection, to let the server know that it is finished.

```
int err = shutdown(sd, SHUT_WR);
```

- The server (client) will then receive and end of file
- The second argument of `shutdown` can be
 - `SHUT_RD (0)`: further receives will be disallowed
 - `SHUT_WR (1)`: further sends will be disallowed
 - `SHUT_RDWR (2)`: neither receives, nor sends will be allowed