

## A TCP CLIENT

---

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
  - i.e., send requests and await replies
  - we use here the application-level protocol
7. Close connection

## PEER IDENTIFICATION

---

- 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 can specify it directly as an integer
  - as command-line argument (read from hard disk, etc.)
    - \* we can 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 then `getservbyname` to obtain the actual port number
  - hardcoded
    - \* e.g., when doing custom client-server applications
  - as command-line argument (read from hard disk, etc.)
    - \* especially good for fully parameterized clients

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

## ALLOCATE A SOCKET

---

- We have to specify:
    - the protocol family
    - the socket type (TCP here)
- ```
#include <sys/types.h>
#include <sys/socket.h>

int sd = socket(PF_INET, SOCK_STREAM, 0);
```
- We end up with a **socket descriptor**.

## CHOOSING A LOCAL IP ADDRESS

---

- Why do we need the local IP address?

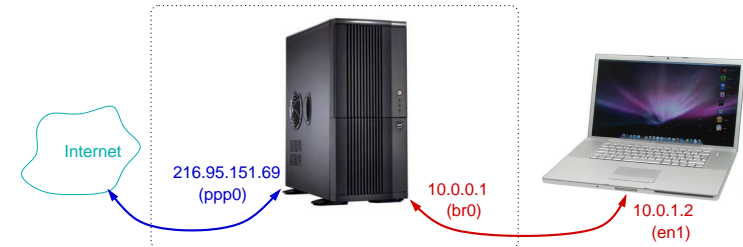
## CHOOSING A LOCAL IP ADDRESS

- 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?



## CHOOSING 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?



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

## CHOOSE A PORT

- We have to 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.
  - So the port number choice is again taken care of by the call to `connect`.

## CONNECT TO THE SERVER

- 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:

```
#include <errno.h>
extern int errno;
struct sockaddr_in sin;
int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
if (rc < 0) {
    perror("connect");
    exit(1);
}
```

## 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;

    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;

    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)htonl(hinfo->h_addr);

    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;
}
```

## COMMUNICATE WITH THE SERVER

---

- We send data using `send`.
- We receive responses using `recv`.
  - 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;
}
```

CSC414/CSC514, WINTER 2012

CLIENT SOFTWARE DESIGN/9

## COMMUNICATE WITH THE SERVER (CONT'D)

---

- What if we do not know how large the response is? For instance, the response may be:
  - One line of text, terminated by `'\n'`.
    - \* We could use `getline` to read the answer.
  - One line of text determines what comes after it.
    - \* Again, we use `getline` to read one line at a time, and decide what to do next.
  - As much as the server cares to send, no special end marker.
    - \* We read until there is no more data.
    - \* But how?

CSC414/CSC514, WINTER 2012

CLIENT SOFTWARE DESIGN/10

## COMMUNICATE WITH THE SERVER (CONT'D)

---

- We check whether we have any data 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.

CSC414/CSC514, WINTER 2012

CLIENT SOFTWARE DESIGN/11

## CLOSING THE CONNECTION

---

- `close` closes the connection and destroys the socket.
- Sometimes we want to shut down communication in **one direction only**:
  - The server receive a request and responds to it.
  - 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

CSC414/CSC514, WINTER 2012

CLIENT SOFTWARE DESIGN/12