CS Recitation 12

Gang Qiao

Q1: how does a client know the address and port number of a server:

A client uses ‘struct hostent* gethostbyname()’ to query

https://www.cs.rutgers.edu/~pxk/417/notes/sockets/index.html

https://www.cs.rutgers.edu/~pxk/417/notes/sockets/udp.html

Step 3a. Connect to a server from a client

If we're a client process, we need to establish a connection to the server. Now that we have a socket that knows where it's coming *from*, we need to tell it where it's going *to*. The *connect* system call accomplishes this.

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

int
connect(int socket, const struct sockaddr *address, socklen_t address_len);
```

The first parameter, `socket`, is the socket that was created with the *socket* system call and named via *bind*. The second parameter identifies the *remote*transport address using the same `sockaddr_in` structure that we used in *bind* to identify our local address. As with *bind*, the third parameter is simply the length of the structure in the second parameter: `sizeof(struct sockaddr_in)`.

The server's address will contain the IP address of the server machine as well as the port number that corresponds to a socket listening on that port on that machine. The IP address is a four-byte (32 bit) value in network byte order (see *htonl* above).

In most cases, you'll know the name of the machine but not its IP address. An easy way of getting the IP address is with the *gethostbyname* library (libc) function. *Gethostbyname* accepts a host name as a parameter and returns a `hostent` structure:

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
};
```

If all goes well, the `h_addr_list` will contain a list of IP addresses. There may be more than one IP addresses for a host. In practice, you should be able to use any of the addresses or you may want to pick one that matches a particular subnet. You may want to check that `(h_addrtype == AF_INET)` and `(h_length == 4)` to ensure that you have a 32-bit IPv4 address. We'll be lazy here and just use the first address in the list.

For example, suppose you want to find the addresses for google.com. The code will look like this:

```
#include <stdlib.h>
#include <stdio.h>
#include <netdb.h>

/* paddr: print the IP address in a standard decimal dotted format */
void
paddr(unsigned char *a)
{
        printf("%d.%d.%d.%d\n", a[0], a[1], a[2], a[3]);
}

main(int argc, char **argv) {
        struct hostent *hp;
        char *host = "google.com";
        int i;

        hp = gethostbyname(host);
        if (!hp) {
                fprintf(stderr, "could not obtain address of %s\n", host);
                return 0;
        }
        for (i=0; hp->h_addr_list[i] != 0; i++)
                paddr((unsigned char*) hp->h_addr_list[i]);
        exit(0);
}
```

Here's the code for establishing a connection to the address of a machine in `host`. The variable `fd` is the socket which was created with the *socket* system call.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>      /* for fprintf */
#include <string.h>     /* for memcpy */

struct hostent *hp;     /* host information */
struct sockaddr_in servaddr;    /* server address */

/* fill in the server's address and data */
memset((char*)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(port);

/* look up the address of the server given its name */
hp = gethostbyname(host);
if (!hp) {
        fprintf(stderr, "could not obtain address of %s\n", host);
        return 0;
}

/* put the host's address into the server address structure */
memcpy((void *)&servaddr.sin_addr, hp->h_addr_list[0], hp->h_length);

/* connect to server */
if (connect(fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
        perror("connect failed");
        return 0;
}
```

Q2: how does a server know the address of a client:

```
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
if (newsockfd < 0)
  error("ERROR on accept");
```

The `accept()` system call causes the process to block until a client connects to the server. Thus, it wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor. The second argument is a reference pointer to the address of the client on the other end of the connection, and the third argument is the size of this structure. The accept() man page has more information.

The argument *addr is a result parameter that is filled in* with the address of the connecting entity.

Hence, a server does not need to query explicitly the address of a client. When a connection is established, the client address is returned to (filled in) the second parameter of accept()

Step 3b. Accept connections on the server

Before a client can connect to a server, the server should have a socket that is prepared to accept the connections. The *listen* system call tells a socket that it should be capable of accepting incoming connections:

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

int
listen(int socket, int backlog);
```

The second parameter, *backlog*, defines the maximum number of pending connections that can be queued up before connections are refused.

The *accept* system call grabs the first connection request on the queue of pending connections (set up in *listen*) and creates a new socket for that connection. The original socket that was set up for listening is used *only* for accepting connections, not for exchanging data. By default, socket operations are synchronous, or blocking, and accept will block until a connection is present on the queue. The syntax of accept is:

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

int
accept(int socket, struct sockaddr *restrict address,
       socklen_t *restrict address_len);
```

The first parameter, *socket*, is the socket that was set for accepting connections with *listen*. The second parameter, *address*, is the address structure that gets filed in with the address of the client that is doing the *connect*. This allows us to examine the address and port number of the connecting socket if we want to. The third parameter is filled in with the length of the address structure.

Let's examine a simple server.
We'll create a socket, bind it to any available IP address on the machine but to a specific port number.
Then we'll set the socket up for listening and loop, accepting connections.

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

...
{
        int port = 1234;        /* port number */
        int rqst;       /* socket accepting the request */
        socklen_t alen;       /* length of address structure */
        struct sockaddr_in my_addr;    /* address of this service */
        struct sockaddr_in client_addr;  /* client's address */
        int sockoptval = 1;


        /* create a TCP/IP socket */
        if ((svc = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                perror("cannot create socket");
                exit(1);
        }

        /* allow immediate reuse of the port */
        setsockopt(svc, SOL_SOCKET, SO_REUSEADDR, &sockoptval, sizeof(int));

        /* bind the socket to our source address */
        memset((char*)&my_addr, 0, sizeof(my_addr));  /* 0 out the structure */
        my_addr.sin_family = AF_INET;   /* address family */
        my_addr.sin_port = htons(port);
        my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(svc, (struct sockaddr *)&my_addr, sizeof(my_addr)) < 0) {
                perror("bind failed");
                exit(1);
        }
```

```
        /* set the socket for listening (queue backlog of 5) */
        if (listen(svc, 5) < 0) {
                perror("listen failed");
                exit(1);
        }

        /* loop, accepting connection requests */
        for (;;) {
                while ((rqst = accept(svc, (struct sockaddr *)&client_addr, &alen)) < 0) {
                        /* we may break out of accept if the system call */
                        /* was interrupted. In this case, loop back and */
                        /* try again */
                        if ((errno != ECHILD) && (errno != ERESTART) && (errno != EINTR)) {
                                perror("accept failed");
                                exit(1);
                        }
                }
                /* the socket for this accepted connection is rqst */
                ...
        }
}
```