CSPB 3753

Client/Server

Our networking crash course provides most of the code needed to create a Client/Server environment that supports both TCP and UDP, which we describe below.

1 Client Program

The client should read input from the command line as follows:

1. `-x <data>` where `<data>` must be a *32-bit unsigned integer*.
2. `-t udp` or `tcp`; for `tcp` the client opens a TCP connection to the server; for `udp` the client sends data to the server using UDP.
3. `-s <ip>` where `<ip>` is the IP address of the server.
4. `-p <number>` where `<number>` is the port being used by the server.

If any of the above command-line arguments are missing, your program is required to print an error message and exit. The command arguments may appear in any order (see `man getopt`). You must determine if the user specified an IPv4 address or a hostname for the `-s` option. Your program is also required to sanity check all of the inputs to be sure each is valid. In particular, if the user provides an invalid port number or a invalid IP address, then your program should report this error and gracefully exit.

After reading the command-line arguments, the client should create a message (see Section 3), then open either a UDP or TCP socket, then send the message to the server. After sending the message, the client must write a status message, for example,

```
sent <data> to server <ip>:<number> via TCP...
```

and then wait for a REPLY message (see Section 3) from the server. If a REPLY message is received from the server, the client should print `success!`, then exit.

If no REPLY message is received from the server after 3 seconds, the client should print an error message and exit.

2 Server Program

Your server will be listening for messages from the client program. Your server program should read input from the command line as follows:

1. `-t udp or tcp`; for `tcp` the server listens for incoming TCP connections; for `udp` the server waits for UDP messages from the client.

2. `-p <number>` where `<number>` is the port to listen for messages.

If any of the above command-line arguments are missing, your program is required to print an error message and exit. The command arguments may appear in any order. Your program is also required to sanity check all of the inputs to be sure each is valid.

Upon receiving a message (see the message format section below), the server must print

```
the sent number is: X
```

where `X` is computed based on the packet received. The server must then send a REPLY message to the client. Unlike the client program, the server must *not* exit, as it will continue to listen for more incoming message from other potential clients.

3 Message Formats

If you do not follow the message formatting described below (e.g., you elect to send `<data>` as a string rather than in a portable binary format), then you will receive at most 85% on the assignment.

3.1 Data Message Format

The message that the client sends consists of

1. A 1 byte version field whose value must be 1.

2. A 4 byte number that is the unsigned integer entered by the user.

You must use a structure to represent this message and you must pack your structure.

3.2 REPLY Message Format

The message that the server sends consists of

1. A 1 byte version field whose value must be 1.

You must use a structure to represent this message and you must pack your structure.

See the "Serialization – How to Pack Data" section of Beej's guide to better understand how to pack structs to send them over the network. You will certainly find the following functions helpful for this assignment

```
uint32_t htonl(uint32_t hostlong);
uint32_t ntohl(uint32_t netlong);
```

which you can think of as "encode" and "decode" functions respectively.

4 Hints

1. Use `getopt()` for reading in the command-line arguments.
2. After a `send()`, it is possible that the client (or server) might not have sent everything it was supposed to, that is, we might have a *partial send*. See the "Slightly Advanced" section of Beej's guide to get around this.
3. For the 3 second timeout, see the "Common Questions" section of Beej's guide.
4. For `uint8_t`, there are no "endian issues", hence network byte order is a not an issue.
5. Do not worry about IPv6 for this assignment.

5 Testing

If you haven't dont this already, you'll need to open two virtual machines and have them communicate with each other in order to test your client and server code. See the Networking Lab for more instructions, or follow the directions below.

Use the virtual network editor in VMware Workstation (Edit > Virtual Network Editor) to configure a Host-only private network as follows:

1. Open the Virtual Network Editor
2. Select VMnet2
3. Select the Host-only radio button
4. Configure a subnet IP (ex. 192.168.110.0)
5. Configure a subnet mask (ex. 255.255.255.0)

6. Select the "Use local DHCP service" option in the Virtual Network Editor to take care of automatic addressing. Use the DHCP Options button to configure the address range. In this example, you'd use Start IP 192.168.110.1 and End IP 192.168.110.254.
7. In each VM's settings, configure its network adapter to use VMnet2 as its custom network.
8. Power on the virtual machines. The command `ip addr` lists all the IP addresses assigned to your (virtual) machine.

If this solution does not work for you, then visit StackOverflow, as there are other posts that walkthrough this setup.

Finally, be sure that you are using a port number that is not already in use by the OS. Higher port numbers tend to be vacant. If you want to be sure that the port number you are using is not already in use, run the command

```
grep -w <your port number> /etc/services
```

If that comes up dry, then you are good to go.

Submission Instructions

Commit and push your final submisson to your GitHub repo (including your code and a makefile). Running `make all` should produce two executables: `client` and `server`.

Bonus (up to 40 pts)

Extend your client/server program to a basic bulletin board system (BBS) that supports a single message board (click here for more information on BBSes – they were the precusor to the world wide web). The client should be able to perform three tasks:

1. `post <title> @ <msg>`: append `<msg>` to message board with the title `<title>`.
2. `read` display all of the messages on the bulletin board to `stdout`.
3. `remove <title>` removes the post labeled `<title>` from the message board.

A client shouldn't be able to delete other clients' posts. You may assume each client has a distinct IP address. The rest of the implementation details are up to you.