Course Home
Home

Administration
Overview
Course email
Anonymous feedback

Assignments
Home virtual machines
Homework dropbox
Class GoPost forum
Class Gradebook

Calendar
Course calendar
HW list

Online Resources
bitcoin/Tor Links

CSE461 Project 0: UDP

Out: Monday January 6, 2013
Due: Monday January 13, 2013 by **11:59pm**.
Teams Allowed: No

Summary

Project 0 asks you to write code that can communicate using UDP, one of the two most commonly used Internet transport protocols (the other being TCP). Your code will send and receive packets containing text. When it receives a packet, it prints the text it contains. When the user types a line of input, you put that text into a packet and send it. By running two instances of your code, what you type at one should be printed by the other, even when they run on distinct machines.

To do this your code must do the following:

- Create a (UDP) socket and possibly bind it to a particular IP address and port.
- Read/write data from/to the socket.
- Create a thread that waits for incoming packets and handles them when they arrive, while using a different thread to deal with whatever the user may type.
- Shut down cleanly.

This is, intentionally, about the simplest conceivable networked application, because figuring out what to write to achieve these basic tasks can be more frustrating than what you're used to. Our goal is to work through those issues, whatever they might be.

As part of this, you need to choose an implementation language. You'll want to use the same language for all the projects, so it's worth making a careful choice. The language (and/or libraries available to programs written in it) must support the functionality described above. Additionally, we will soon need the ability to time out when waiting for an incoming network message, and eventually we will need support for public/private key encryption and associated functionality (like digital signatures). It's very hard to know just how well some language/system will support functions you haven't yet used, but you might at least check that there seem to be reasonable encryption libraries (for instance) for any language you consider.

Note: We'll grade your code on attu, so the language, build system, libraries, etc. you use must be available there.

I think the only real danger in choosing a language is if you are considering something very cutting edge, or thinly used for whatever reason, and find later in the course that you're writing code that users of other languages get for free from existing libraries. Because of that, I think any language that's reasonably commonly used in production environments should work for all the projects. Java and C++ will certainly work. There may be better choices, depending on what languages you know, or want to know. (I've implemented in perl, so decided to avoid Java and C++. Some things that would have been easy in Java/C++ (threading, in particular) ended up being surprisingly awkward in perl, though, so that's an example of the kind of surprise that can occur.)

Operational Specifications

You're writing a single program that will behave as a client or a server, depending on how it is invoked.

You don't need to deal with network errors in this project. (You'll do that in the next one.) You should do standard checking of return codes, but you don't need to worry about the case that a sent packet doesn't arrive or that the other end of the communication misbehaves.

Server Behavior

1. Invoke with command `./run <portnum>`, where `<portnum>` is a port number. Code should create a datagram (UDP) socket bound to that port.
   Note: `run` is probably a shell script. See the Grading section.
2. Print to `stdout` a useful IPv4 address for the host you're running on and the port of the just created socket.
   Note: The host you're on probably has multiple IP addresses. Finding a useful one can be difficult. One approach is to get the host's name (keyword: `hostname`) and then use a library routine to resolve the name to an IPv4 address (keyword: `getaddrinfo`). Exactly how you do this will depend on the language you're using.
3. Wait for an incoming packet.
4. Print the source IP address and port of the just arrived packet.
5. Print the character data contained in the just arrived packet.
6. At this point the output should look like this (except for the particular addresses):

```
$ ./run 46199
128.208.2.42 46199
```

```
[Contact from 76.28.236.179:33120]
192.168.0.105 33120
```

7. Send back to the source of the incoming packet a packet containing your IPv4 address and port, separated by a single space, and without any trailing '\0', '\n', or the like.
8. Loop reading `stdin` and sending any lines of text read to the UDP port that contacted you.
   Note: You can assume that each read line fits in a single UDP packet.
9. At the same time, read the data from any incoming packets and print it to `stdout`.
   Note: Use threads to concurrently read `stdin` and incoming packets.
10. Terminate when you see EOF on `stdin`.
    Note: On Linux you can generate EOF by typing ctrl-d.

Client Behavior

1. Invoke with command `./run <hostname> <portnum>`. The `hostname` can be a domain name (e.g., `attu1.cs.washington.edu`) or an IPv4 address (e.g., `128.208.1.137`).
2. Create a UDP socket that can be used to send packets to the destination given by the command line arguments.
3. Determine a useful IPv4 address for the host you're running on, and the port your socket is bound to.
4. Send to the destination a packet containing your IPv4 address and port, separated by a single space, and without any trailing '\0', '\n', or the like.
5. Loop reading `stdin` and sending any lines of text read to the UDP port that contacted you.
6. At the same time, read the data from any incoming packets and print it to `stdout`.
7. Terminate when you see EOF on `stdin`.

Execution and Debugging

- The standard Linux program `nc` can be used as a working implementation of (essentially) Project 0 while you are debugging your implementation. To start it as a server, issue a command like `nc -ul -p 38119`. To start it as a client, issue a command like `nc -u attu1.cs.washington.edu 38119`. Using `nc` gives you a working server side while debugging the client side, and vice versa. Your should be able to run two instances of your own code, though, when it's finished.
- `wireshark` is a tool that can display network packets sent from or received by your machine. It's a bit difficult to use, and requires privilege you cannot get on a CSE machine, but it could be useful if running on your own machine. It is most helpful if you think your code is sending packets but there's no sign that anything is being received. `wireshark` can tell you whether or not packets are actually being sent.
- IP addresses can have "limited scope." For instance, the address `127.0.0.1`, which corresponds to the name `localhost`, always means the host on which the name is used. Some IP addresses, like `192.168.0.1`, are meaningful only on the local network, but are not meaningful when issued on some other network. Finally, some addresses are "global" and can be used anywhere. All of this means you can have various kinds of trouble connecting a client and server that are on different machines or different networks. Running two instances on a single machine should always work. Running two instances on two CSE machines should also always work (so long as you avoid the `localhost` issue). Running one instance at home and one at CSE may or my not work; you might find that you can send packets to CSE from home, for instance, but can't send from CSE to your house.
- `attu` is actually four machines, whose names are `attu1` through `attu4`. If you try to connect your client to your server using the name `attu`, it will connect at random to any one of the four actual machines. Use specific host names, rather than `attu`, to avoid problems.

Grading and Required Script

We'd like to be able to write a script that runs all submissions. For that to be possible we need all submissions to support a single way to build and launch them. For that reason, you **must** submit a script, `run`, that will perform any required build of your code (e.g., compiling and linking a C++ program) and then launch it, passing whatever command line arguments are given when invoking `run`. That is, when we write our script we'll be able to pretend that everyone has submitted an executable named `run`, with the executable following the specifications noted earlier.

To actually grade, we will run on `attu`. You should verify that your program and the `run` script work there.

Turn in

When you're ready to turn in your assignment, do the following:

1. The files you submit should be placed in a directory named `proj0`. There should be no other files in that directory.
2. Create a README.TXT file that contains your name, student number, and UW email address.
3. Put the README.TXT file, your project 0 solution source code, and your `run` script in the `proj0`directory.
4. While in the directory that is the parent of `proj0/`, issue the command `tar czf proj0.tar.gz proj0`.
5. Verify that the tar file contains the files you intend to submit: `tar tf proj0.tar.gz`.
6. Submit the `proj0.tar.gz` file to the course dropbox. (There's a link to the dropbox in the navigation section of all course pages.)
7. Fill out the following two question survey to give us feedback on this assignment, which will help us tune future projects:

   https://catalyst.uw.edu/webq/survey/zahorjan/222449

   Any grading associated with the survey is limited to whether or not you took it; what you say cannot affect your grade.