CSE461 Project 0: Network Programming / UDP

Out: Monday April 1, 2019
Due: Friday April 12, 2019 by **11:59pm**.
Teams: Required: pairs

Project Goals

Project 0 asks you to implement what might be described as a client-server data transfer application. We don't actually need dozens of implementations of data transfer, though, so the real point is the experience of implementing communication protocols. While you won't directly re-use your Project 0 code in later projects, you will have to implement very similar functionality in all of them. This is a chance to explore implementation approaches, because your choice could have a big effect on how much time those later projects will take.

The operation of communication protocols always involves asynchrony, which can be an implementation challenge. Project 0 gives you experience implementing protocols using two approaches:

- **Thread-based**. Here we use threads to provide concurrency. By carefully choosing when threads are created and what they do, we can often find a simple solution to asynchrony. The distinguishing feature is that each blocked thread is waiting for only a single thing (e.g., keyboard input, or network input). You probably have had some exposure to threads in previous courses, so this approach should be at least somewhat familiar. Additionally, threads have a long history, and so mature support for them exists in most languages/systems. C/C++, Java, and Python all have completely sane support for threads.
- **Non-blocking IO** or **asynchronous IO** or **event-loop based**. The basic idea of these alternative approaches is for a single thread to wait on more than one "event source" at a time. For example, a

thread might wait for *either* keyboard input *or* a packet to arrive. Depending on the language/package you use, you might need to combine this idea with multi-threading (e.g., when using Java NIO). At the other extreme (e.g., node.js), the entire program is single-threaded, the major advantage of which is that you have no self-inflicted race conditions.

C++ has Boost Asio (http://www.boost.org/doc/libs/1_57_0/doc/html/boost_asio.html), an asynchronous IO package. Java has NIO (http://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio-pdf.pdf), a non-blocking IO interface. Python has the pyuv package (http://pyuv.readthedocs.org/en/v1.x/), an event loop package. node.js (http://nodejs.org/) is a javascript-based language designed around the event-loop model. You can use any of those. If you want to use something else, please contact the course instructional staff.

To achieve the goals of this project, you must work in pairs. That is, there are two different developers involved. Project 0 has distinct client and server sides, so there are two distinct pieces of code involved in a full solution. We take advantage of that by dividing the work in this way:

	Client	**Server**
Thread-based	`Partner A`	`Partner B`
Not thread-based	`Partner B`	`Partner A`

This schedule allows you to (a) experience both implementation approaches, (b) make progress independently of your partner (since your own client and your server should interoperate, even if implemented in different languages), and (c) experience having your code interoperate with code written by someone else. When you're done, all four combinations of client and server choices should work together.

P0P Messages

P0P, the Project 0 Protocol, defines what messages are sent between the client and server, and how they are encoded. (There is a mail protocol, POP, with a similar-looking name, but the two have nothing to do with each other.) P0P is designed to support the Project 0 application. The Project 0 application transfers lines of input from the client's `stdin` to the server, which then prints them on its `stdout`.

Protocol Headers

P0P is much more realistic than the protocols shown in sections and class, in large part because it defines a message as a header plus data (rather than just data). Without a header, you can have only one kind of message, and so can't do simple things you almost certainly need to do, even if you don't realize it yet. (One example is returning an error indication, for instance, even though we don't do that in this project.) Protocols you design should always include headers in message encodings.

Protocol Sessions

P0P supports the notion of a *session*. A session is a related sequence of messages coming from a single client. Sessions allow the server to maintain state about each individual client. For instance, the server could, in theory, print out how many messages it has received in each session, for instance, or it could maintain a shopping cart for each session. (We don't actually implement either of those.)

Unlike TCP (which has "connections"), UDP doesn't have any notion related to sessions, so we build them as part of our protocol.

Protocol Messages and Format

P0P defines four message types: HELLO, DATA, ALIVE, and GOODBYE. All message encodings include a header. The header is filled with binary values. The header bytes are the initial bytes of the message. They look like this, with fields sent in order from left to right:

magic	version	command	sequence number	session id

16 bits 8 bits 8 bits 32 bits 32 bits

- `magic` is the value 0xC461 (i.e., decimal 50273), whentaken as an unsigned 16-bit integer in network byte order (big endian). An arriving packet whose first two bytes do not match the magic number is silently discarded.
- `version` is a one byte protocol version, and is the value 1.
- `command` is a one byte value: 0 for `HELLO`, 1 for `DATA`, 2 for `ALIVE`, and 3 for `GOODBYE`.
- `sequence numbers` in messages sent by the client are 0 for the first packet of the session, 1 for the next packet, 2 for the one after that, etc. Server sequence numbers simply count up each time the server sends something (in any session). Sequence numbers are four bytes.
- `session id` is an arbitrary 32-bit integer, in networy byte order. The client chooses its value at random when it starts. Both the client and the server repeat the session id in all messages that are part of that session.

Multi-byte values are sent big-endian (which is often called "network byte order").
In DATA messages, the header is followed by arbitrary data; the other messages do not have any data. The receiver can determine the amount of data that was sent by subtracting the known header length from the length of the UDP packet, something the language/package you use will provide some way of obtaining.

Only one P0P message may be sent in a single UDP packet, and all P0P messages must fit in a single UDP packet. `P0P` itself does not define either maximum or minimum DATA payload sizes. It expects that all reasonable implementations will accept data payloads that are considerably larger than a typical single line of typed input.

P0P Message Processing

Server

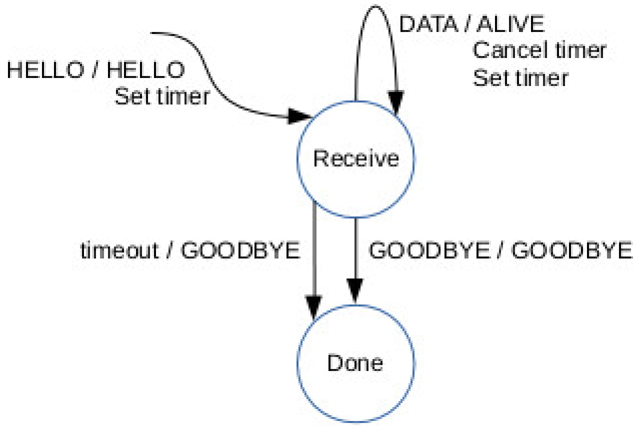
The server sits waiting for input from the network or from `stdin`. Execution of the server ends when either end of file is detected on `stdin` or the input line is "q". When the server quits, it sends `GOODBYE` messages to all sessions (described shortly) thought to be currently active.

When a new packet arrives, the magic number and version are checked. The packet is silently discarded unless those fields match the expected values.

Next, the server examines the session id field. If a session with that id has already been established, it hands the packet to that session. Otherwise, it creates a new session and hands the packet to it.

Server Session

Server sessions operate according to the following FSA diagram:



Here transition labels are of the form *event / action(s)*, meaning *"when an event of the specified type occurs while in the source state, take the actions specified and transition to the destination state."* For instance, the transition `HELLO / HELLO; Set timer` means that when a `HELLO` message arrives, reply with a `HELLO` message and also set a timer that will raise an event at some later time (unless it is canceled), and then transition to state `Receive`.

Sessions are created (by the server) after receiving a message with a new session id. The newly created session checks that this initial message is a `HELLO`, and terminates if not. Because the client may never send a `GOODBYE`, sessions garbage collect themselves by setting an inactivity timer. If no message is received from the client for too long, a `GOODBYE` is sent and the session terminates.

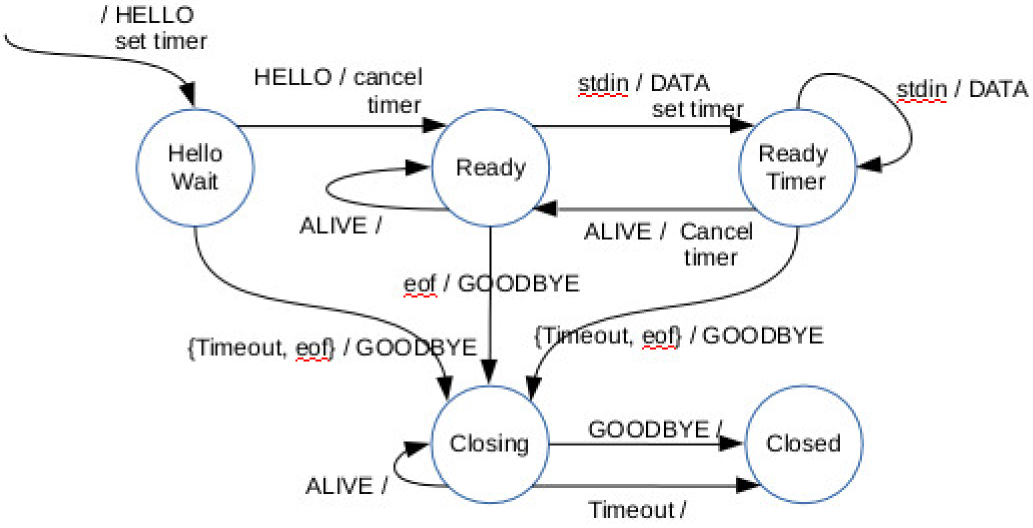
When a `DATA` message is received from the client, its data payload is printed to `stdout`. To give the client a way to determine that the session is still up, the session sends an `ALIVE` message in response to each `DATA` message it receives.

The server session should keep track of the client sequence number it expects next. That is, if it has just processed a packet with sequence number 10, it should remember that the next sequence number expected is 11. If the next packet received has a sequence number greater than the expected number, a "lost packet" message should be printed for each missing sequence number. If the next packet's sequence number repeats the last received packet number, a "duplicate packet" message is printed and the packet is discarded. If the next packet's sequence number is less than the last packet's sequence number, we assume it is caused by a protocol error and the session closes: it sends a `GOODBYE` and transitions to `DONE`. (Sequence numbers "from the past" can be caused by the Internet delivering packets "out of order." That can occur, and more realistic protocols would want to deal with it more gracefully.)

If the server session receives a message for which there is no transition in its current state, it is a protocol error. In that case, it closes. For instance, receiving a `HELLO` while in `Receive` is an indication something is seriously wrong, and the only option is to close.

Client Behavior

The client follows the following FSA:



GOODBYE/ in any state transitions to state CLOSED

Basically, the client sends lines of input to the server. If no packets were ever lost, the client would receive a packet back for every one it sends: a `HELLO` in response to a sent `HELLO`, an `ALIVE` in response to a `DATA`, and a `GOODBYE` in response to a `GOODBYE`. If no response is received within a timeout, the client takes that as an indication the server is not running, and so the client terminates. (Note that this is not a very good assumption as the problem could just be a single dropped packet. We'll look at how to do a better job of guessing whether or not the server is really there later in the course.)

Session termination normally starts with the client, and involves a `GOODBYE` message in each direction. However, if the client receives a `GOODBYE`, it believes that the server is gone, no matter what the client's current state, and so it transitions immediately to the `Closed` state.

The client shuts down when it detects end of file on `stdin` and the input is coming from a tty, or when the the input line is 'q' and input is from a tty. If `stdin` is connected to a file and eof is reached, the client should try to shut down after all outgoing messages have been put on the network. (Whether or not you can do that might depend on the implementation language/package you're using.)

Operational Details

To make it easier, or maybe even possible, to run your implementations we have to have a standard way to invoke your code and to process its output. You must conform exactly the specifications shown here, for both invocation and output. (Sorry.)

Invocation

Because how you launch a program can depend on its implementation language, you may need to write a shell script to conform to these instructions. Whatever you do, when we download your files and issue the commands shown below, what is described in the instructions should happen. You should check, on `attu`, that it does, before submitting.

- **Server**: `./server <portnum>`
  `<portnum>` is the port number the server should bind to.
- **Client**: `./client <hostname> <portnum>`.
  `<hostname>` and `<portnum>` give the location of the server. The `hostname` can be a domain name (e.g., `attu1.cs.washington.edu`) or an IPv4 address (e.g., `128.208.1.137`).

Output: Basic Typed Input

Suppose you run two client instances, back to back, like this:

```
$ ./client localhost 1234
one
two
three
eof
$ ./client localhost 1234
foo
bar
eof
```

Here `eof` is printed by the client program to indicate that end of file has occurred on `stdin`. (On Linux, type ctrl-d.) The other lines are what the user typed.

The server's output should look like this:

```
$ ./server 1234
Waiting on port 1234...
0x736f0b1f [0] Session created
0x736f0b1f [1] one
0x736f0b1f [2] two
0x736f0b1f [3] three
0x736f0b1f [4] GOODBYE from client.
0x736f0b1f Session closed
0x545537a9 [0] Session created
0x545537a9 [1] foo
0x545537a9 [2] bar
0x545537a9 [3] GOODBYE from client.
0x545537a9 Session closed
```

The first value on each interesting line is the session id. The number in square brackets is the sequence number of the packet that caused this output line.

Output: Input Redirected from File

Redirecting `stdin` to read from a file makes it possible to offer so much input so fast that packets are lost. For example:

```
$ ./client localhost 1234 <Dostoyevsky.txt (Dostoyevsky.txt)
eof
```

produced this server output:

```
Waiting on port 1234...
0x149780c3 [0] Session created
0x149780c3 [1] The Project Gutenberg EBook of The Brothers Karamazov by Fyodor
0x149780c3 [2] Dostoyevsky
0x149780c3 [3]
0x149780c3 [4]
0x149780c3 [5]
0x149780c3 [6] This eBook is for the use of anyone anywhere at no cost and with
0x149780c3 [7] restrictions whatsoever. You may copy it, give it away or re-use
0x149780c3 [8] the terms of the Project Gutenberg License included with this eB
0x149780c3 [9] online at http://www.gutenberg.org/license
0x149780c3 [10]
0x149780c3 [11]
0x149780c3 [12]
0x149780c3 [13] Title: The Brothers Karamazov
0x149780c3 [14]
0x149780c3 [15] Author: Fyodor Dostoyevsky
0x149780c3 [16]
...eliding many lines...
0x149780c3 [456] unhappy young woman, kept in terror from her childhood, fell i
0x149780c3 [457] kind of nervous disease which is most frequently found in peasa
0x149780c3 [458] who are said to be “possessed by devils.” At times after terri
0x149780c3 [459] hysterics she even lost her reason. Yet she bore Fyodor Pavlov
0x149780c3 [460] sons, Ivan and Alexey, the eldest in the first year of marriag
0x149780c3 [461] Lost packet!
0x149780c3 [462] Lost packet!
0x149780c3 [463] Lost packet!
0x149780c3 [464] Lost packet!
```

```
0x149780c3 [465] Lost packet!
0x149780c3 [466] looked after by the same Grigory and lived in his cottage, whe
0x149780c3 [467] Lost packet!
0x149780c3 [468] Lost packet!
0x149780c3 [469] Lost packet!
...eliding many lines...
```

Exactly which packets are dropped is non-deterministic. Your output should match the above, except for differences caused by the non-determinism.

Output: Concurrent Clients

You can cause two clients to be concurrently active with a single server using a shell script like this one:

```
#!/bin/bash
./client localhost 1234 <Dostoyevsky.txt >dual-c1.out 2>&1 &
./client localhost 1234 <Dostoyevsky.txt >dual-c2.out 2>&1 &
```

Start the server, redirecting its output to some file, and then execute the script.

Doing that should produce server output that shows the two clients are running concurrently, as in this example output file. (dual-server.out) (This output is also non-deterministic.)

Execution and Debugging Details

- The standard Linux program nc can help in early debugging of your implementation. As a client, nc will send input typed at the keyboard to some server. As a server, it will print whatever is sent to it.

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`. `nc` will never understand the P0P message header, so the most you can get out of it is an easy way to verify that something is being sent or received.

- `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 privileges you cannot get on a CSE machine, but it could be useful if running on your own machine. It is most helpful when `nc` indicates your code is sending packets, but there's some problem with the receiver understanding them: `wireshark` will show you the bits that are put on the wire, and so lets you determine whether you have an encoding (sender-side) or decoding (receiver-side) bug. (Note that if you run the client and server on the same machine, `wireshark` is unlikely to be able to see any packets traveling between them, even if they are 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 may 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 seven machines, whose names are `attu1` through `attu7`. If you try to connect your client to your server using the name `attu`, it will connect at random to any one of the seven actual machines. Use specific host names, rather than `attu`, to avoid problems.

Grading Procedure

For the testing component of grading, we will run on `attu`. You should verify that your program can be invoked there in the fashion described above, and that it runs successfully. **You should verify that your submission works across the Internet** by running client and server on different machines (e.g., `attu1` and `attu2`).

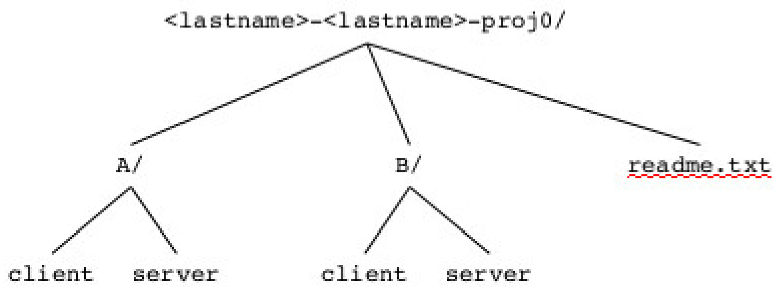
Note that it is very hard to verify that your code will run for us on `attu`. It may be that it works for you, because you have some environment variable setting you code relies on, but fails for us, because we don't. We suggest testing by launching a shell that has a minimal environment and running there:

```
$ env -i bash   # launch a shell with a minimalm environment
$ ./server 1234 # now use that shell..
...             # ctrl-d to terminate the minimal environment shell
```

Turn in

[This is perhaps outdated. It's likely we will switch to git turn in. However, we'll want the directory structure shown here even if we do.]

There should be **only one submission per team**. You should turn in a `.tar.gz` file that, when unpacked, creates the directory structure shown here:



- The `.tar.gz` file you submit should have a name that is the last names of the two partners and the string "proj0", all separated by hyphens. For example, `guy-zahorjan-proj0.tar.gz` is a plausible name for the file to be turned in.
- Expanding that tar file should create a directory whose name is like `guy-zahorjan-proj0`.
- That directory should contain subdirectories, named `A` and `B`, and a `readme.txt` file. (In this an all subsequent projects, pay attention to the specified file case.)
- The first two lines of the readme file should be formatted like this:

  ```
      A: Nat Guy
      B: John Zahorjan
  ```

  Those lines may be followed by any number of lines giving additional comments or information.
- Subdirectories `A` and `B` should contain executable files named `client` and `server`, as well as any other files required to run your code. The files in `A` are the ones written by partner A, etc.

We'll do things like this:

```
$ tar xf guy-zahorjan-proj0.tar.gz
$ cd guy-zahorjan-proj0
$ cat readme.txt
$ cd A
$ ./server 5432
```

All those commands should succeed. (They won't if you don't follow the details of this section.)