Assignment 1 (100 points)

Anirudh Sivaraman

2024/02/02

General instructions

This assignment consists of two parts: the multiple choice section and the code section. Each part is worth an equal amount (even though the points on Gradescope are not equal). If you have not been added to the Gradescope for the class, please contact the course staff.

A quick reminder, this is an **individual assignment** (except for a small extra credit portion), and the course policies on collaboration and academic honesty apply.

1 Conceptual (50 points)

See Gradescope for the questions. The assignment is largely multiple choice, but also includes some short answer and some basic network experiments.

Following is some background information on some of the tools we ask you to use.

1.1 Background on wget

wget is a tool that uses HTTP (the same protocol your web browser uses) to retrieve web pages or other files on the Web. If wget is not installed on your machine, install it using MacPorts or Homebrew on Mac and apt-get or an equivalent package manager on Linux. (Or use the provided Google Cloud VM). For this assignment, we'll use wget to download a file available at a particular URL. wget also reports the current rate at which the file is being downloaded.

1.2 Background on ping

Ping is probably the first tool a network admin uses to measure latency on a network. Like most networking tools, ping has been around for a long time (since 1983). Ping sends a special type of packet (called an ICMP packet) and measures the *round-trip latency* from host A to host B and back to host A. The round-trip latency is simply twice the standard latency from host A to host B, assuming the forward and return paths are symmetric.

You can run the ping program by typing this on the terminal.

```
ping www.google.com
```

Terminating ping will report the min, max, average, and standard deviation of round-trip latencies observed by ping. The report looks like this.

```
--- www.google.com ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 9.739/19.444/43.388/13.887 ms
```

The way ping works is as follows. Ping first sends an ICMP request packet from your computer to www.google.com, which www.google.com responds to with an ICMP response packet. Your computer measures the time elapsed between the time the request was sent out and the response comes back and reports this as the round-trip latency.

Recall from lecture that latency is made up of four components: application-level delay; transmission delay of individual packets, which depends on the packet size and the link's capacity; the propagation delay due to the physical distance between two points; and queueing delay because some other application ends up sharing the same link on a packet-switched network.

In the questions, we'll ignore application-level and transmission delay because it takes very little time to generate a ping packet in the ping application and ping packets are very small, meaning that they have a small transmission delay.

1.3 Background on tcpdump

The final tool we'll experiment with in this assignment is `tcpdump`. Like `ping`, `tcpdump` is another widely used command line utility that's been around since the 1980s. Its purpose is to allow you to capture packets on your network and parse them into a (somewhat) human readable format. `tcpdump` also allows for flexible filtering of packets. There are nicer graphical tools built on top of tcpdump, namely Wireshark, that make capturing packets much more user friendly, but we will stick with `tcpdump` as we can run it on our Google Cloud VMs, which do not have a graphical interface. The reason we are using the Google Cloud VMs is because they are a more controlled environment; there is much less extraneous traffic such as web broswers or other services than your personal device likely has.

Specifically, here we will use `tcpdump` to explore the headers of each packet, which define the information needed by each layer. Note, normally the headers are packed into a fixed-length binary encoding to minimize overhead, but `tcpdump` will decode these for us. We use the header information to answer questions about certain application layer protocols.

2 Coding Assignment (50 points)

The goal of this portion of the assignment is to experience some hands-on socket programming and build up to to a "real" networked application of a chat server. One external resource that you may find useful for this portion is Beej's Guide to Network Programming. Although it gives examples in C rather than Python, because Python's socket module is a thin wrapper around the lower level C libraries, it it still useful! You should also familiarize yourself with the Python socket and select libraries.

Note we do not require you to adress a number of "real" issues including serialization of data, message framing in TCP, etc. Chapter 7 of Beej's guide discusses some of these issues, starting from Section 7.4. In a real implementation, you might also use something like protobuf. You are welcome to explore implementing these solutions yourselves! However, our automated grading systems expect plain (encoded) python strings.

Deployment and Development

In order to develop and test your code, we will take advantage of Google Cloud's education credit program. We will release a separate guide on how to setup the VMs in the cloud, and some tips and scripts for using them.

The advantage of using cloud VMs is that we can configure these VMs to have public IPs and loose accesss rules.[1] This way, you can try running your network applications over the actual internet, between your machines, rather than just on localhost. Extra credit (explained below) will be given for also running your solutions against a classmates!

[1] in other environments TCP and UDP ports might be blocked by a firewall

We will also run our own, persistent instances of each server application to serve as an example that you can test your client application(s) against. Information about the addresses and ports that these will run at will be included in a Piazza post (since they are subject to change). **We highly reccommend doing this, you may find some surprises :)**.

2.1 UDP RPC server (15 points)

We'll start with a simple a simple remote procedure call (RPC) system that allows a client to invoke methods remotely on a server machine. We'll support one RPC method where the client sends "prime(N)" and the server sends back yes or no depending on whether N is prime or not. You should use UDP to communicate between the client and the server. Use the provided starter code as a starting point, and submit the final versions of both your client and server programs.[2]

The starter code for the client is setup to take an IP address and then a port. So for example to run a client that sends requests through localhost to a server setup on the same machine, you would run

```
python3 rpc_client.py 127.0.0.1 9001
```

To have your client make requests to a different machine (either one of your own VMs, or someone else's), simply change the IP address.

The starter code for the server is setup to take a port. And will be run with the following commands

```
python3 rpc_server.py 9001
```

Note your server should accept both external and localhost traffic (*hint: what address do you bind to*). Because UDP is connectionless, it also needs to get the client address to send the reply to. (*hint: look in the socket documentation at variants of recv(). What do we use in the lecture notes to specify an addres to* **send to**?)

Important for autograder: The client sends 10 random RPC calls, and also contains an optional random seed argument. Do not modify these, or if you really want to, when run with the random seed, the client should make RPC calls with the first 10 random integers generated with that seed.

We will check that your client prints a `yes` or `no` based on the servers response to each call. For example, a single request in the client could generate the following output

```
sent: prime(82)
prime: no
```

We expect the server to print each number it receives.

Any other output will be ignored, so feel free to include your own prints for debugging or other purposes.

For testing aginst our server: our RPC server will expect messages of the form "prime(n)" where n is the integer argument.

2.2 TCP string concatenation server (15 points)

Next, we will implement a simple application that uses TCP. With TCP, we now also have to handle listening for an accepting connections from clients. Here, a client sends some data to the server, which concatenates another piece of data to that data and sends it back to the client. We have provided starter code for the client and server as a starting point for this assignment. Submit the final versions of your client and server programs.

The server and client will be run similarly to before, except now the server can also take a random seed to determine the random string it appends to the data.

[2] UDP is unreliable. So, an application that calls recv() on a UDP socket may deadlock waiting for lost data that never arrives. Any real UDP application needs a timeout so that it doesn't wait indefinitely. This timeout can be implemented using select (see lecture 3 notes)

```
    python3 concatenation_client.py 127.0.0.1 9001
```

and

```
    python3 concatenation_server.py 9001
```

We expect the client and server to print the data they receive.

For extra credit, you can make the concatenation server *persistent*; able to handle multiple clients, by listening for and accepting connections, and handling clients disconnecting.

2.3 TCP Message Server/Relay (20 points)

Now you that you are familiar with how sockets and TCP works, we will build a networked application that is close to a real application. Say we have two machines who need to communicate with each other. In an ideal world, we would have the following setup:

```
A <---> B
```

The double-headed arrow indicates that communication can happen in both directions between A and B: A can send to B and B can send to A through a direct TCP connection. However, if A and B are both end hosts residing in private networks, they have no way of establishing a connection with each other, since neither of them have a public IP. To remedy, this we can introduce a known server S with a public IP between A and B:

```
A <----> S <----> B
```

Now S establishes connections with both A and B, then takes bytes coming in from A and sends them to B and vice versa.

This arrangement is can also be thought of as S acting as a proxy server. Proxy servers can be used for much more besides relaying messages. For instance, S could require A to authenticate with S before communicating with B. S could be also account for how much traffic A sends out of the network. It could also be used to inspect the traffic between A and B to ensure it doesn't contain anything malicious.

So, we'll also give server S the additional requirement to detect any bad words in the strings being transmitted and replace them with corresponsing good words. The bad words are: $['virus', 'worm', 'malware']$ and their corresponding good words are: $['groot', 'hulk', 'ironman']$. Note that the length of the bad word and its corresponding good word is the same. This means after the relay replaces the bad words with the good words, the length of the string being transmitted does not change.

For this exercise, you'll be required to turn in two programs, one for the end hosts A or B (chat_client.py) and one for the server S (chat_server.py). They will be run the same way as before:

```
    python3 chat_client.py 127.0.0.1 9001
```

and

```
    python3 chat_server.py 9001
```

The 'chat_client.py' program should take input from the user through stdin, send the input to the server, and print any message it receives from the server to stdout. *(hint: use select()!)*. The server needs to receive messages from A, replace the bad words, and forward modified message to B (or vice versa). *(hint: use select() here too!)*

Again, extra credit will be offered for making your server persistent: with the ability to handle clients connecting and disconnecting. However, for this extra credit, your server should also be able to handle more than 2 clients, which should be a natural extension of making the server persistent.

3 Extra Credit

Following is a summary of the extra credit opportunities for this assignment.

- (1 point) Make your TCP concatenation server persistent and able to handle multiple concurrent clients.
- (1 point) Make your chat server persistent and able to handle more than 2 clients at once.
- (1 point) Explain some of the shortcomings of sending plain strings over sockets as we direct you to in this assignment.
- (6 points) Find a partner, and make a document together showing that your implementations of the 3 applications are interoperable. In other words, you should try running your *_client.py from your Google Cloud VM against their *_server.py on their Google Cloud VM, and vice versa. The document should show the commands you ran and have screenshots or the text output of each program. Finally, please it would be also be helpful for us if you make this document relatively organized, for example by describing the VMs you used or by adding additional output to your programs showing where packets/connections are coming from.

  A friendly reminder: besides this portion, the assignment is individual. Achieving interoperability by copying or sharing each other's code is not allowed. Instead, you should discuss your programs at the protocol level; what your respective clients sends and what your server expects.

3.1 Submission

Your code should be uploaded to Gradescope, and should contain:

```
rpc_server.py
rpc_client.py

concatenation_server.py
concatenation_client.py

chat_server.py
chat_client.py

extra_credit.pdf (optional)
```