

Assignments

[Click here to exit full screen mode.](#)

Assignment #1

Assignment has been submitted on Sep 28, 2021 9:50 PM

Draft - In progress

Submitted

Returned

Title

Assignment #1

Due

Sep 28, 2021 11:55 PM

Number of resubmissions allowed

Unlimited

Accept Resubmission Until

Sep 30, 2021 11:55 PM

Status

Submitted Sep 28, 2021 9:50 PM

Grade Scale

Points (max 100.00)

Modified by instructor

Sep 17, 2021 10:56 AM

History

Tue, 28 Sep 2021 21:49:33 -0400 Rishabh Jain (rjain57) saved draft

Tue, 28 Sep 2021 21:50:39 -0400 Rishabh Jain (rjain57) submitted

Instructions

CS3357 Assignment #1

Fall Session 2021

Purpose of the Assignment

The general purpose of this assignment is to explore some simple network concepts and programs using a simple network simulation tool. This assignment is designed to give you experience in:

- various network tools and protocols
- the must tool for UDP network simulation
- using simple Python network programs
- network reliability, or unreliability, as the case may be

Assigned

Saturday, September 11, 2021 (please check the main [course website](#) regularly for any updates or revisions)

Due

The assignment is due Tuesday, September 28th, 2021 by 11:55pm (midnight-ish) through an electronic submission through the [OWL site](#). If you require assistance, help is available online through [OWL](#).

Late Penalty

Late assignments will be accepted for up to two days after the due date, with weekends counting as a single day; the late penalty is 20% of the available marks per day. Lateness is based on the time the assignment is submitted.

Individual Effort

Your assignment is expected to be an individual effort. Feel free to discuss ideas with others in the class; however, your assignment submission must be your own work. If it is determined that you are guilty of cheating on the assignment, you could receive a grade of zero with a notice of this offence submitted to the Dean of your home faculty for inclusion in your academic record.

What to Hand in

Your assignment submission, as noted above, will be electronically through [OWL](#). You are to submit all requested scripts / screen shots / images through OWL. (See below for details.)

Assignment Task

Believe it or not, the Internet is a pretty unreliable network by design. An IP network can lose packets, have packets arrive out of order, and in some cases even corrupt packets in transit. On top of this, protocols like TCP run that give us stability and reliable data transfers, but underneath it all is still an unreliable network at its core.



In this assignment, you will be exploring network reliability and unreliability using a simple network simulation tool called must. Why is it called must you ask? Well, it's an acronym that stands for Mike's UDP Simulator Tool. Yes, I wrote this ... around 15 years or so ago. Despite its age, it still works like a champ,

though! In essence, must acts as a proxy between two communicating UDP applications, forwarding packets back and forth. By providing different options to must, however, you can create a variety of suboptimal network conditions, inducing packet loss, corrupted packets, delays, and out-of-order packet delivery. These things can and do happen in the Internet at large, but must forces them to happen, making it a great tool for testing out UDP applications under a variety of conditions.

Using must and a simple Python client-server application, you will be exploring first hand what happens when the network misbehaves. Again, this happens all the time, but our use of higher layer protocols like TCP insulates and protects us from these issues. Following through the exercises below, you will get a good handle of what can happen in an unreliable network ... and learn how to use must, which could come in handy for future assignments.

Getting An Environment Set Up

For this assignment, you will need access to a Unix-like environment of some kind. The options at your disposal, depend on your computer and operating system:

- If you are running Linux on your computer already, that should suffice as it is. You'll just need the compiler tools and [Python 3](#), and those things are quite likely installed by default, depending on your distribution.
- If you are running MacOS, you are also in pretty good shape, and should be able to run things from a Terminal or using [XQuartz](#). You will need the command line compiler tools installed, which might depend on [Xcode](#), as well as [Python 3](#), which may also be installed by default. (This largely depends on the version of MacOS that you are running.) If you have virtualization software (like [VMWare](#) or [VirtualBox](#)) with a Linux VM handy, that should work just as well too.
- If you are running Windows, it is likely best to virtualize in some fashion to get Linux up and running. Virtualization software (like [VMWare](#) or [VirtualBox](#)) could work for this, but these days, my preferred approach is to use the [Windows Subsystem for Linux or WSL](#). You can get a number of Linux distributions running under Windows quite readily with this, assuming that your computer supports things well. Once it's up and running, you just need the compiler tools and [Python 3](#) installed and you're ready to go. (And those should be there by default, depending on your distribution.) You could always use a live CD or live USB key to boot into Linux, but that's often not the most convenient way of doing things.
- If you are using a Chromebook, you should be able to [get Linux on it](#), depending on the age of your device. Again, compiler tools and [Python 3](#), and you're set.
- If none of these are an option or work for you (for example because you have a Windows computer that doesn't support virtualization), we have a Linux server that you can connect into and use. Simply use ssh to connect into compute.gaul.csd.uwo.ca with your Western credentials and you should be all set. You'll be doing things remotely, but it should still work fine for this assignment.

Getting The Software You Need

Now you just need a copy of the Python UDP test app that you'll be using and a copy of must. You can find each of these in a .zip file attached to this assignment.

For the UDP test app, simply unzip the app.zip file, and you'll have a client.py and a server.py. These are loosely based off of the sample UDP code from the Kurose & Ross text. To use this app, simply first run the server as in:

```
python3 server.py
```

Depending on your installation of [Python 3](#), simply running python might suffice as well instead of python3. The server will then report which port number it's waiting on for messages from the client. Make a note of that and then run your client as:

```
python3 client.py host port
```

In this case, `host` is the name of the host the server is running on and `port` is the port number reported when you ran the server. If the client and server are on the same host (likely the case unless you are using multiple computers for this), `localhost` will likely suffice for the host name. So, if the server reported that it is running on port 12345 and is on the same host as the client, you can run the client as:

```
python3 client.py localhost 12345
```

Doing that, anything you type at the client will be sent over to the server line by line and printed there. Pretty snazzy. The cleanest way to stop the client is with a Ctrl-d, indicating the end-of-file on standard input, and the client will shut down gracefully. (You can also hit Ctrl-c, but the client will throw a message in the process.) You can shut down the server with a Ctrl-c and it will handle that gracefully too. One more cool thing you can do is something like this:

```
python3 client.py localhost 12345 < file.txt
```

If `file.txt` is a plain text file, the client will read this line by line and send it over to the server, shutting itself down when done. Nothing too magical is going on here, this is just a standard shell redirection going on, substituting the contents of the file in place of standard input when things run. But, it's a nice way of saving some typing when you run things!

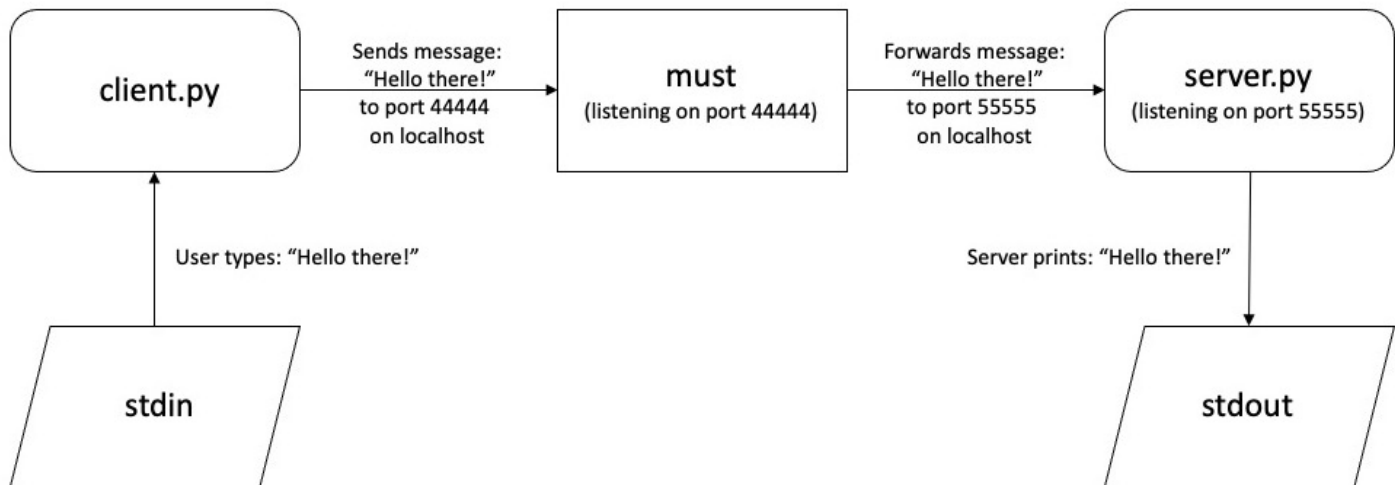
For must, simply unzip the must.zip file and use make to build it. Instructions on how to use it can be found in the included README file or by simply running must with no parameters. As noted above, must acts as a proxy between the client and server forwarding packets back and forth as necessary. Let's suppose that your client, server, and must are all on the same host (localhost) and that your server is waiting for messages on port 55555. You would then run must, asking it to forward received messages to port 55555 on the localhost, so that anything it receives is resent to the server. You would do this like:

```
./must -h localhost -f 55555
```

When run, must reports it is listening on port 44444, so anything received at that port will get forwarded to the server on localhost at port 55555. You can then run your client as:

```
python3 client.py localhost 44444
```

Anything typed at the client is sent to must, as it is waiting for messages at port 44444 and must forwards these packets to the server on port 55555. To the client and server, they are communicating with one another, but must is actually sitting between them. By default, must does nothing to the messages and simply forwards them along ... you can see a little more of what's going on by using must's verbose (-v) option when running it. In the end, the flow of messages looks like this:



Now that you have UDP messages flowing through must normally, you can start looking at having some fun by adding some unreliability to the mix.

Getting Down to Work

Now you have an environment to work in and some familiarity with must and the UDP app used in this assignment. What to do next? In essence, this assignment involves you running the UDP client and server in a number of scenarios, with data forwarded through must and must injecting various types of issues into the packet stream to simulate an unreliable network. To demonstrate that you are doing all of this correctly, you are to capture the output of this and submit that for this assignment.

So, how to capture the output? Well, you're in luck. Unix-like systems come with a handy program called `script` by default. When `script` is run and given a file name, it will create a new shell. You can then run a command at this new shell and `script` will capture the output of this and save it in the named file.

When you're done running commands and want `script` to save its output, simply type `exit` at the shell prompt, and `script` will terminate and save its collected output.

To make things more straightforward and consistent, you should find a text file for the client to send to the server and use this same file in all your scenarios. Doesn't really matter what it is ... you just want there to be a decent number of lines to it (10+) so you can see things happening. Instead of using the above redirection trick to send your file via the client, you should use a slightly different pipe trick. Consider this command:

```
cat -n file.txt | python3 client.py localhost 12345
```

Instead of simply redirecting your file into standard input for the client for transmission, `cat -n` will number each line of your file and then pipe it into standard input for your client. This way, your server will receive numbered lines from your file instead of the lines themselves, making it much easier to spot missing lines or lines that are out of order from transmission. Pretty slick! Shell hacks are the best!

For each scenario listed below, you will save script output of the client, server, and must (in verbose mode) in separate files, calling the files:

- `scenario-x-client.txt`: Saved client output for scenario x, where x is a number between 1 and 3.
- `scenario-x-server.txt`: Saved server output for scenario x, where x is a number between 1 and 3.
- `scenario-x-must.txt`: Saved must verbose output for scenario x, where x is a number between 1 and 3.

Consequently, across all 3 scenarios, you will be submitting a total of 9 script output files for this assignment. You should also submit the original text file used across all your scenarios, so with that, your submission should be a total of 10 files. (They can be submitted as individual files or bundled together as a .zip file. Be sure to test the .zip file before submitting it though!)

And now for the scenarios ...

Scenario 1 - Lost Packets

The first scenario is pretty straightforward. You want must to lose packets. Try a packet loss rate of around 50% and record what happens. This isn't likely to happen in practice if all is well in the network, but in a highly congested network, hitting a loss rate of 50% or higher is certainly a possibility.

Note that must won't necessarily lose exactly 50% of your packets as it's choosing things randomly. Over time, with enough packets, it should average out though. Also note that packets might get lost along the way without must doing anything ... when using UDP that can happen on its own. It's not likely to happen, but certainly a possibility!

Scenario 2 - Corrupted Packets



The second scenario is also pretty straightforward. Now, instead of losing packets, you want must to inject some bit errors into your packets. You can play with things to find a number that works for you, but don't set it too high. For example, with a 50% bit error rate, it is quite likely that every byte of your file will be corrupted and that's just being unnecessarily cruel to your poor data. Play around with things a bit to see what works well for creating some problems!

Note that bit errors can cause different sorts of issues here. If you're lucky, they might just cause characters in your text to change to other characters. If you're unlucky, Python on the receiving end might not be able to properly decode the text (as it doesn't see the characters as being valid any more) and that could cause an error to be thrown. Both situations are good demonstrations of the effects of corruption. Again, also note that this corruption in theory could still happen on its own, but UDP's checksum mechanism should stop this from happening. (But in such an event would cause the packet to be lost ... oh well!)

Scenario 3 - Our of Order Packets

The third scenario has you use must to deliver packets out of order. This requires a bit more thought as must doesn't have a "jumble up the order of my packets" option. Think about what it means for packets to be out of order and you'll figure it out. (Hint: It has something to do with delays.) You might have to experiment with things a bit to get this to happen, but it shouldn't take too much tinkering around. With your lines of text being numbered, it should be pretty easy to see things put out of order!

Additional resources for assignment

-  [must.zip](#) (5 KB; Aug 27, 2021 3:38 pm)
 -  [app.zip](#) (1 KB; Aug 27, 2021 3:38 pm)
-

Submission

Assignment Text

This assignment allows submissions using both the text box below and attached documents. Type your submission in the box below and/or use the Browse button or the "select files" button to include other documents. **Save frequently while working.**

Source

Styles



Format



Font






Size



Words: 0, Characters (with HTML): 0/1000000

Attachments

-  [scenario-1-server.txt](#) (2 KB; Sep 28, 2021 9:44 pm) [Remove](#)
-  [scenario-2-client.txt](#) (1 KB; Sep 28, 2021 9:44 pm) [Remove](#)
-  [scenario-3-must.txt](#) (2 KB; Sep 28, 2021 9:45 pm) [Remove](#)

-  [scenario-3-server.txt](#) (3 KB; Sep 28, 2021 9:45 pm) [Remove](#)
-  [scenario-2-server.txt](#) (3 KB; Sep 28, 2021 9:45 pm) [Remove](#)
-  [scenario-3-client.txt](#) (1 KB; Sep 28, 2021 9:45 pm) [Remove](#)
-  [scenario-1-must.txt](#) (2 KB; Sep 28, 2021 9:44 pm) [Remove](#)
-  [scenario-1-client.txt](#) (1 KB; Sep 28, 2021 9:44 pm) [Remove](#)
-  [file.txt](#) (1 KB; Sep 28, 2021 9:44 pm) [Remove](#)
-  [scenario-2-must.txt](#) (2 KB; Sep 28, 2021 9:45 pm) [Remove](#)

Select more files from computer no file selected


or select more files from 'Home' or site

Proceed

Preview

Save Draft

Cancel

 Don't forget to save or proceed!