MPCS 54001, Winter 2021

Project 2[1]

Due: February 14 at 11:59 pm

Submit your assignment to your GitLab repo, in a folder named "project2". This work must be entirely your own. If you need help, post questions to Ed Discussion and/or visit the staff during office hours. As a reminder, if you make a public post on Ed Discussion, please don't give away the answer!

Task

The primary goal of this project is for you to implement a simplified file transfer server that supports reliable data transmission over UDP

The secondary goal of this project is for you to acquire the practical skills necessary to implement a networked application from a protocol specification (the protocol specification is split between this project description and RFC 1350).

Please be sure to study the protocol specification before asking for protocol clarifications on the project (based on previous iterations of the course, most questions have already been answered in the protocol specification).

Your project must be implemented in Python 3.x.

Trivial File Transfer Protocol (TFTP)

The Trivial File Transfer Protocol (TFTP) is a lightweight file transfer alternative to FTP which runs over UDP. TFTP was motivated by the need for storage-constrained devices to move files over the network.

RFC 1350 contains the full specification of TFTP. The online version of the TCP/IP Guide also contains a useful description of TFTP in this section.[2] Please feel free to take a look at those

[1] Adopted from a Networks project written by William Conner

[2] At just 10 pages, RFC 1350 is one of the most readable RFCs that actually describes a complete protocol specification.

resources, but keep in mind that you will implement a slightly modified version of TFTP for this project as described in the next section.

Echo Server

The server shall do the following:

- Support read (RRQ) and write (WRQ) requests from TFTP clients
- Accept server port number as the first positional command-line argument
- Accept retransmission timeout (in milliseconds) as the second positional command-line argument
- Use UDP as the underlying transport protocol
- Support 512-byte data blocks
- Support only binary mode (i.e., no need to explicitly support ASCII)
- Retransmit DATA and ACK messages after timeouts
- Support simultaneous file transfers (i.e., servers should be multi-threaded)
- Exit with informational message when ERROR messages received

The server shall not:

- Support TFTP options or option negotiation
- Attempt to recover from ERROR messages

You are not allowed to use off-the-shelf TFTP libraries, for obvious reasons. Please contact the course staff early if you have doubts about whether a library you want to use is appropriate or not.

Example usage:

- `python tftp_server.py <port> <timeout>`

Lock-Step Messaging

Since TFTP is implemented over UDP, your TFTP server needs to handle the possibility of lost DATA and/or ACK messages *from either the client or the server*. The basic strategy is that the sender will retransmit the same data block until that block is acknowledged. Conversely, the

receiver will retransmit the same acknowledgement until the next expected data block is received. A timer initialized with the specified retransmission timeout should be used to trigger retransmissions.

Since retransmissions could occur during the file transfer, there is now the possibility of duplicate DATA or ACK messages. You will need to handle duplicates appropriately.

Deliverables

Create a directory named `project2` in your individual GitLab repository. Upload your source code (server source files, README file) to that directory. You should verify that your files were submitted correctly using the web interface at https://mit.cs.uchicago.edu. Your README should contain instructions for using your code. You should also include useful code comments for the graders.

Grading

This project will be worth 15% of your overall grade (out of the 40% for projects) for the course.

We will run your code submissions and grade them using a TFTP client and a message-dropping UDP proxy in the middle.

General grading rubric for Project 2 is as follows (total 100 points):

- Starts/Runs and binds to a UDP port: **5 points**
- Sends file correctly during read requests: **15 points**
- Receives file correctly during write requests: **15 points**
- Uses separate ephemeral port for file transfers: **10 points**
- Retransmits data blocks after timeouts: **10 points**
- Retransmits acknowledgements after timeouts: **10 points**
- Handles duplicate data blocks: **5 points**
- Handles duplicate acknowledgements: **5 points**
- Handles multiple `RRQ/WRQ` requests in succession without restarting (loops around and accepts the connection again): **5 points**

- Either spawns new thread, forks new process, or asynchronously handles TFTP client requests to support multiple simultaneous file transfers (i.e., no serial servers): **5 points**
- Exits with informational message upon receiving error message: **5 points**
- Style points, grader discretion (efficiency, code correctness, README, comments, or other considerations): **10 points**

Testing

You can perform local testing on the same machine by using the TFTP client utility to talk to your TFTP server implementation.[3] An example of reading and writing a file with the TFTP client is included below. The following examples assume that your server is running on `localhost` at port 8080, so you can adjust the URL as necessary for your testing.

Example #1: Reading file from server

```
$ tftp
tftp> connect localhost 8080
tftp> binary
tftp> verbose
Verbose mode on.
tftp> get file.txt newfile.txt
getting from localhost:file.txt to newfile.txt [octet]
Received 10240 bytes in 0.1 seconds [819200 bits/sec]
tftp> quit
```

Example #2: Writing file to server

```
$ tftp
tftp> connect localhost 8080
tftp> binary
tftp> verbose
Verbose mode on.
tftp> put newfile.txt
putting newfile.txt to localhost:newfile.txt [octet]
Sent 10240 bytes in 0.1 seconds [819200 bits/sec]
tftp> quit
```

[3] Keep in mind that your submission might also be graded with the client and server running on different machines.

Since it is very unlikely that packets would be dropped during your local testing, a UDP proxy with a configurable packet drop rate is provided on Canvas with the assignment. You can run the UDP proxy with the following example command (note that you'll need to point your TFTP client at the proxy instead of your server).

Example #3: Running UDP proxy on port 8081 with a target drop rate of 10% and a TFTP server running at localhost:8080

```
$ java -jar udpproxy.jar 8081 localhost 8080 0.1
```

Example #4: Pointing TFTP client to UDP proxy

```
$ tftp
tftp> connect localhost 8081
tftp> binary
tftp> verbose
Verbose mode on.
tftp> put newfile.txt
putting newfile.txt to localhost:newfile.txt [octet]
Sent 10240 bytes in 10.3 seconds [7953 bits/sec]
```

I strongly suggest developing on the Linux machines, rather than using your own personal machines, since that is where your projects will be graded. Any differences between that environment and your personal computers are ultimately your responsibility.