📚

README

Status	In progress

ABOUT: CSEE 4119 Simplified TCP over UDP
Project Files
How To Run 🏃
    Required Libraries 📚
    architecture
    Installing and Running `newudpl` 🔗
    Running `tcpclient.py` 💻
    Running `tcpserver.py` 💻
Bugs and Features ⭐
    Bugs
    Features
    Additional Features 💫
        Argument Checking
        Fast retransmit 🚄
        Verbose Logging 💬
Testing Environment 🧪

ABOUT: CSEE 4119 Simplified TCP over UDP

- Erin Liang, ell2147
- Slip days requested: 3 (using all of them up I think!)

📕 Implementation of a simple version of TCP that runs over UDP where a client establishes a connection to the server via a three-way handshake, transfers a user-specified file to the server, and terminates the connection when the maximum amount of retransmissions are reached at any given point in the connection or when the server is done receiving the file. To simulate an unreliable network, `newudpl` is used as a proxy to forward the file sender's (`SimplexTCPClient`) messages to the receiver (`SimplexTCP server`). In practice, TCP offers additional guarantees that were not implemented, including congestion and flow control.

This project was assigned in Professor Misra's CSEE 4119 Computer Networks Columbia University course during the Spring 2023 semester.

Project Files

📔 All submitted files and a short description of each

```
├── proj3
   ├── README.md <-- You're here now!
   ├── DESIGN.md
   ├── utils.py
   ├── tcpserver.log
   ├── tcpclient.log
   ├── tcpserver_debug.log
   ├── tcpclient_debug.log
   ├── tcpclient.py
   └── tcpserver.py

| Filename                        | Description
|--------------------------------|-----------------------------------------------------------------------------------------                                                            |
| `README.pdf`                   | Describes program, program usage, known bugs, etc.
| `DESIGN.pdf`                   | Describes the internal workings of the program and design tradeoffs considered
| `utils.py`                     | Helpful functions/params used in both tcpclient and tcpserver, e.g.
|                                | tcp header creation, checksum calculations, timeout EWMA formula weights
| `tcpserver.log`,`tcpclient.log`| Screendump of a typical client-server interaction
| `tcpserver_debug.log`          | See above, but with more robust logging
| `tcpclient_debug.log`          | See above.
| `tcpclient.py`                 | Contains all the code for client functionality
| `tcpserver.py`                 | Contains all the code for server functionality                                                   |
```

How To Run 🏃

Required Libraries 📚

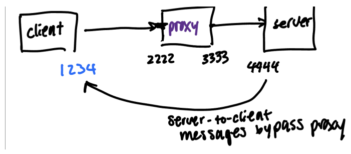
📔 In order to run the program, the following Python libraries must be installed. There are some other ones (`random`, `sys`, `os`, `time`) that are required but should be provided natively by your installation of Python.

I would've added a script to do this for you and make grading easier, but I'm running out of time to study for the final.

- logging
- ipaddress — for ipaddress cli argument validation
- struct — for packing/unpacking the tcp header values into bytes
- argparse — for cli argument handling
- traceback — for debugging statements while I was coding. This import and its invocations can be commented out.

architecture

- Below are commands that recreate this setup:



Installing and Running `newudpl` 🔗

- `newudpl` is a network emulator with a UDP link. To install and build `newudpl`:

```
# unzip the tar file
tar -xf newudpl-1.7.tar
cd newudpl-1.7.tar
./configure
make
```

- Example `newudpl` command.
    - Running `newudpl` that listens for client's messages on port 2222 and forwards messages to port 4444. Both the client and server program are run on the local machine. The probability of packets being dropped is 50%. Oof.

```
./newudpl -p 2222:3333 -i 127.0.0.1:1234 -o 127.0.0.1:4444 -vv -L50
```

- You can use `-p [receive port]:[send port]` to specify which port `newudpl` listens for udp packets from source host and from which port to send udp packets to destination host.
    - The client needs to send packets to the `receive port` of `newudpl`. This port is specified in the client

Running `tcpclient.py` 💻

- To run the client, an example command is:

```
python3 tcpclient.py ./testfiles/screendump.txt 0.0.0.0  2222 1152 1234
```

- This will start running a tcpclient that transfers `screendump.txt` listening on port 1234 for ACKs. The tcpclient will forward all segments it needs to send to the server to `newudpl` (the proxy), which is listening on address (0.0.0.0, 2222). The tcp `windowsize` for pipelining segments is set to 1152.
- Generically, the usage of `tcpclient` is:

```
usage: tcpclient.py [-h] file address_of_udpl port_number_of_udpl windowsize ack_port_number
```

- Note that the `windowsize` must be a multiple the MSS set in `utils.py`, but the program will yell at you if you don't provide the correct arguments. The MSS can be adjusted in utils.py to test performance of smaller segments. I set MSS to 576 in my submission because Wikipedia says that that's the typical TCP MSS size.

Running `tcpserver.py` 💻

- To run the client, an example command is:

```
python3 tcpserver.py ./testfiles/screendump.txt  4444 127.0.0.1 1234
```

- This will start running a tcpserver listening on port 4444 locally. It will receive `screendump.txt` and acknowledge receipt of tcp segments by sending ACKs to the client's address (port 1234 locally)
- Generically, the usage of `tcpserver` is:
    - note that (address_for_acks, port_for_acks) is the client's address. We assume that the client

```
usage: python3 tcpserver.py [-h] file listening_port address_for_acks port_for_acks
```

Bugs and Features ⭐

Bugs

The code works as it is and as the assignment spec specifies. Specifically, it implements:

- Connection establishment via the three-way handshake
- Reliable transmission of a file
- Connection teardown via sending a FIN (client and server initiated)
- Retransmission timer adjustment
- Logging — different levels of logging. See the additional features section.

This isn't really a bug, but the program is quite slow when running `newudpl` on the 50% packet loss setting for really large window sizes, although it does terminate correctly and gracefully. This is probably because so many packets are timing out… I've seen timeouts get to 1273.4688541185 seconds when a lot of retries are allowed and the `TIMEOUT_MULTIPLIER` is pretty high. The behavior is discussed later in the design doc and in the features blurb below:

Features

You can adjust some of the TCP variables set in `utils.py` to test what yields the best performance. These variables are:

- `MSS`, the maximum amount of data that can be carried in a single TCP segment.
- `MAX_RETRIES` , the maximum amount of retries the programs try to send a segment before initiating the connection teardown sequence (i.e. sending a FIN to to the client/server)
- the retransmission and timeout constants: `INITIAL_TIMEOUT`, `ALPHA`, `BETA`, `TIME_WAIT`,and `TIMEOUT_MULTIPLIER`. These values are discussed in the design doc

Note that adjusting these variables comes at your own risk. No matter what, the connection is guaranteed to terminate gracefully (but not always efficiently). For example, if MAX_RETRIES is too low and there is significant packet loss, the client will probably hit the threshold for retransmissions, only send over a portion of the data, and send a FIN to terminate the connection. This is reasonable behavior because a client should probably wait for the network conditions to get better before trying to transmit a file.

Additional Features 💫

📕 Below are features that I spent a good deal of time implementing that I'm hoping will be considered for extra credit :)

Argument Checking

- To ensure a smooth run of the program, a help option (`-h`) is added to both the client and server programs. Additionally, command line arguments are validated for both the client and the server, primarily using the `argparsing` module. Intuitive usage errors are outputted in the case of incorrect argument values, arg counts, etc.
- Validations are done in `validate_args` in `utils.py` . They include:
  - Making sure the file to be transferred exists on the system. This is primarily so the connection doesn't get set up only to not be able to find the file on the system.
  - port numbers are integers within the 1024-65535 range
  - IP addresses are valid IPv4 or IPv6 addresses in dotted decimal notation.
  - Ensuring that windowsize is a multiple of MSS
- Invoking the help options for the server.

```
$ python3 tcpserver.py -h
usage: tcpserver.py [-h] file listening_port address_for_acks port_for_acks

Bootleg TCP implementation over UDP

positional arguments:
  file              file to send over TCP
  listening_port    port to listen on
  address_for_acks  address to send ACKs to
  port_for_acks     port to send ACKs to

optional arguments:
  -h, --help        show this help message and exit
```

- Invoking the help options for the client:

```
$ python3 tcpclient.py -h
usage: tcpclient.py [-h] file address_of_udpl port_number_of_udpl windowsize ack_port_number

Bootleg TCP implementation over UDP

positional arguments:
  file                 file that client reads data from
  address_of_udpl      emulator's address
  port_number_of_udpl  emulator's port number
  windowsize           window size in bytes
  ack_port_number      port number for ACKs

optional arguments:
  -h, --help           show this help message and exit
```

- Example prints when a file doesn't exist

```
$ python3 tcpclient.py file 0.0.0.0 2222 576 1234
TCPClient - INFO - ============================
TCPClient - INFO - TCPClient Parameters:
TCPClient - INFO - file: file
TCPClient - INFO - address_of_udpl: 0.0.0.0
TCPClient - INFO - port_number_of_udpl: 2222
TCPClient - INFO - windowsize: 576
TCPClient - INFO - ack_port_number: 1234
TCPClient - INFO - =============================
UTILS     - ERROR - File does not exist.
TCPClient - ERROR - Invalid arguments. Aborting...
```

- Similar print statements occur when an invalid port number is specified, etc.

Fast retransmit 🚄

- When window sizes get bigger and `newudpl` has a lot of packet loss, lots of duplicate ACKs are sent and it can take a while for packets to be retransmitted since the client has to wait for the timer to expire.
- To try to detect packet losses earlier, I implemented a basic version of fast retransmit. Upon receipt of 3 duplicate ACKs, we resend the first segment in the window, as it was probably lost.
- The implementation is in `SimplexTCPClient.send_file_gbn()`:

```
# Part of fast Retransmit. If we receive 3 duplicate ACKs, then resend the segment
# with the lowest sequence number in the window.
if num_dup_acks == 3:
  logger.info(
    "Received 3 duplicate ACKs. Resending segment with seq_num {send_base}"
  )
  segment, num_retries = window[0]
  window[0] = (segment, 0)
  self.socket.sendto(segment, self.proxy_address)
  num_dup_acks = 0
```

Verbose Logging 💬

- To get more verbose logging, you can set `LOGGING_LEVEL` in `utils.py` to logging.DEBUG. This may come at the expense of performance if the file is large and especially because the logs are being written to multiple streams.
- To get less verbose logging, you can set LOGGING_LEVEL to `logging.INFO` , `logging.WARNING`, `logging.ERROR`, etc. Higher levels will include all the logs from the levels below. For example, INFO will include all the logging.error logs. See the documentation for Python's logging module for more details.
  - `DEBUG` — lots of information: every packet and its ack_num, seq_num, payload, and flags, when a new sample RTT is measured / discarded when the packet being measured is a retransmission, window moving forward and which segment is being removed from it, the expected ack_num when a duplicate ack is received, etc.
  - `INFO` — timeout updates, states for connection teardown and establishment (e.g. CLOSE_WAIT, SYN_SENT, etc), file transfer messages, out of order segments, etc.
  - `WARNING` — maximum retransmissions reached —> beginning to terminate program, handling unexpected exceptions (haven't seen this one be invoked yet)
  - `ERROR` — checksum verification failures, unexpected flags in received segments, incorrect ack numbers during the handshake sequence, invalid arguments,
- For convenience, logs are written to `tcpclient.log` , `tcpserver.log` , as well as `stdout`

Testing Environment 🧪

- This program works for my 2020 Mac (Intel) running Python 3.9.13, so it should work for Linux environments. I'm on macOS Big Sur v11.76.