Lab 3 – Custom TCP/UDP File Transfer Utility over Python

Authors: Angela Zhou (azhou@hmc.edu), Bruce Yan (byan@hmc.edu)

Overview

Using the native unix application, `scp`, to transfer files may be reliable but still limited to the limitations of `ssh`. We have decided to use Python for the implementation of socket transfer. Our server sends and receive data using both TCP and UDP protocols. TCP is primarily used for transferring of metadata and UDP is primarily used for transferring of data chunks (pieces of files).

Approach

We have decided to transfer the files using two separate streams, transferring the metadata using TCP and transferring the actual file pieces using UDP protocol. Since we know that metadata about the file, such as file size and checksum, need to be preserved; we've decided to send that through the TCP protocol. We decided to use Python's file read function to read a certain blocksize and transfer the amount that's read.

A summary of the metadata information is shown below:

TCP - metadata		UDP - data	
Client	**Server**	**Client**	**Server**
File name	Resend data information	Send data chunks	Receive data chunks
Total # of blocks		Block Index (marks which block # is being transferred)	
Checksum of file			
Checksum of each block			
Total Size of file			
Size of each block			

Modules/Libraries

In our project, we made use of the following public python libraries

- `socket` – this is the low-level networking interface for socket servers and clients
- `os` – contains portable tools to help the user use operating system dependent functionality, such as getting the size of the file
- `hashlib` – we used md5 checksum function in this security interface to ensure that files sent and received matched the same hashed md5 checksum
- `sys` – we used sys to grab the arguments (such as file name) that were being passed into the system
- `ast` – we used ast to help convert string lists between strings and literals between streams

Metadata Transfer (TCP)

The idea behind our TCP protocol implementation is to ensure the successful transferring and receiving of important information, such as blocksizes and checksums. To store all the metadata, we've decided to use an array of arrays to send the data over. We chose arrays because they are relatively cheap in cost and quite versatile in terms of functionality.

Data Transfer (UDP)

The idea behind our UDP protocol implementation is to ensure the sending of packets as fast as we can, and send as many packets as possible. If packets are dropped along the way due to either delay, link loss, or an inconsistent connection, our TCP implementation will come in hand if we drop packets along the way due to delay and link loss, we can use our TCP implementation to confirm and re-transmit the missing pieces. Instead of sending and checking the checksum at the same time, we propose to send all data using UDP first, then verify using TCP at the end

Performance

The following table illustrates our program's performance and throughput. As shown, the throughput goes down as the delay and loss increases. We ran through each of the runs multiple times using 20Mb, 50Mb, and 100Mb files. The Avg Throughput of 3 runs is recorded.

A few attempts was made for 1GB file; however, it often took over 1.5 hours to transfer the entire file and we feel due to the single threaded nature of things, it wasn't very practical. It might have been faster if the 1GB data was copied to a portal HDD, then FedEx'd over from location A to location B.

Bandwidth (Mb/s)	RTT Delay (ms)	Loss (%)	Avg Throughput (Mb/s)
100	10	1	.21
100	20	5	.13
100	60	10	.12
100	100	20	.14
100	200	20	.14

Table 1: Data collected for bandwidth testing on Deterlab

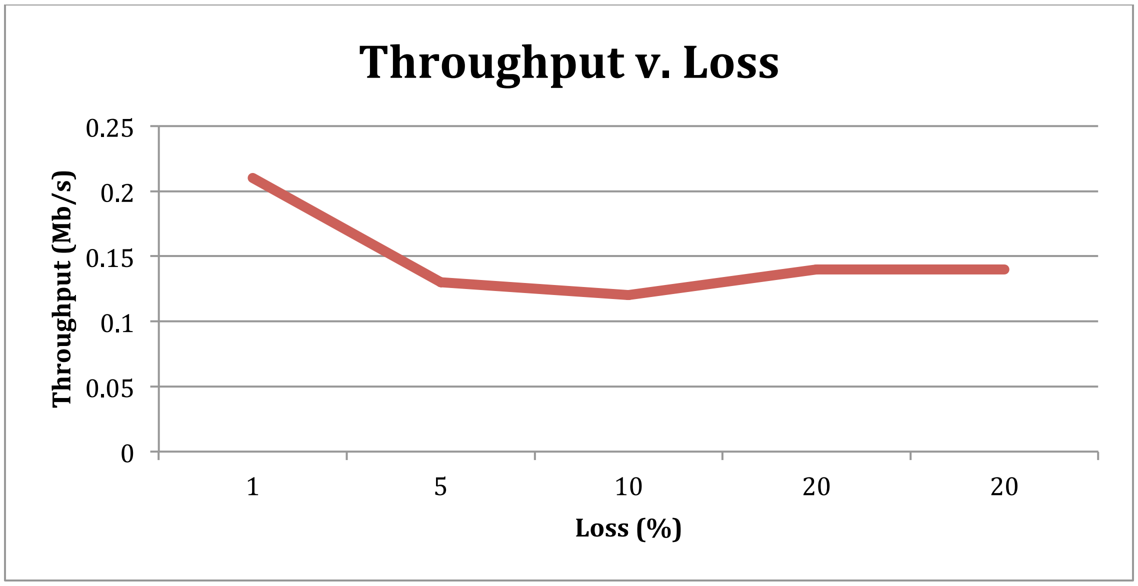

Figure 1: Graph of throughput vs loss for custom file transfer utility over Python.

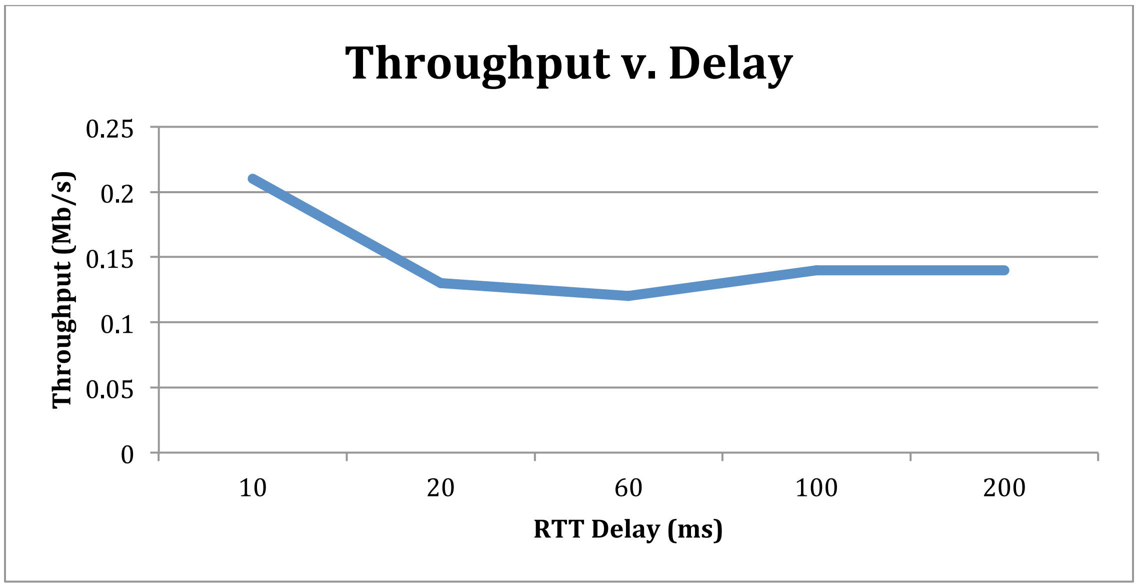

Figure 2: Graph of throughput vs RTT delay for custom file transfer utility over Python.

Initial Thoughts

Our single-threaded implementation of a TCP/UDP socket server is very limited. We noticed that because data was being serialized, we had to process the data in chunks. Since we only had 1 worker thread operating on the data, we could theoretically improve the performance by multi-threading it in the future. However, this potential improvement has an upper bound. As we maximize the bandwidth usage, we will eventually be limited by the delay and the loss; thus increasing threading will not improve our throughput especially knowing that TCP connections are extremely compromised by large delays and loss.

Appendix – Presentation

Our presentation is hosted on Google Slides (requires HMC login):
https://docs.google.com/a/g.hmc.edu/presentation/d/13wsIZ2vtJckYLAp41PUPLskLWEJXtkDF7xMN3odpDWM/edit?usp=sharing