David Han

# Live Throughput Capture Per Second Using Tcpdump and String Operations

## INTRODUCTION

For my research this semester, my goal was to calculate and record the throughput of a machine for every second. What made this project not as easy as it looked was that the program I needed to write would be under the constraints of a router. This router had very minimal software packages, so it could not support the power of networking library functions like pcap.

I began this project writing the program in C using pcap.h library functions. The plan consisted of using well-established library functions like pcap_open_live, pcap_compile, and pcap_setfilter to essentially open a channel to receive packets and filter them on the restrictions that we wanted only UDP and TCP packets. However, what I learned was that the the compiled programs I wrote in either C or Golang (which is the language I ended up using at the end) would not run on the router due to mismatch in processors known as cross-compilation.

Thus, I turned to an alternative of using tcpdump output to a program that would scan each line representing a packet, and analyze the packet for the source to destination tuple and the packet length in bytes. This approach ultimately worked and I was able to implement this with accuracies around +/- 5%.

## IMPLEMENTATION

To implement, I wrote this program in Golang. We want to be able to print the aggregated throughput every second as well as add to a packet statistic store for every packet received from tcpdump. Tcpdump along with filters for only UDP and TCP packets was used to pipe its output to this program called nstdin.go. This gave us the source for getting the live packets. In order to print the aggregated throughput every second and keep a temporary packet store for statistics as long as a second, two different Golang channels with accompanying go-routines were used to implement this system.

The first channel called “input” was used to watch over the incoming input from tcpdump. Tcpdump would be immediately outputting packets it recorded. The go-routine “getInput” would use the NewScanner function from bufio library that was given input from stdin (os.Stdin) to get a line of output from tcpdump separated by newline character. This line was fed into the “input” channel. A infinite loop was placed around this code so that the go-routing would go on forever.

The second channel called “timeChannel” was used to notify my program that a second had passed and the aggregated throughput statistics needed to be printed out. The accompanying go-routing was called “timeout” and it used the sleep function from time to sleep for one second, then announce in the “timeChannel” that one second had passed.

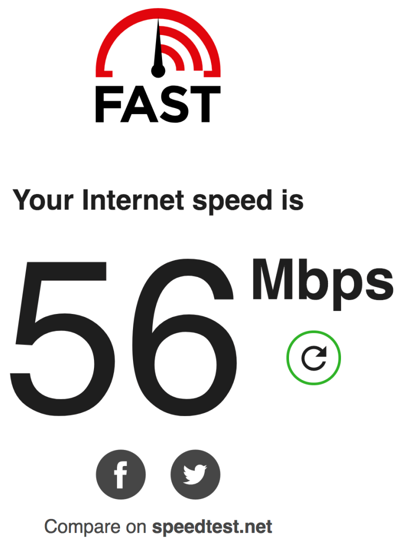
In the main function, I also allocated a hash map with a key being a string consisting of the source IP address, “greater than” character, and destination IP address, and value being the packet length. This hash map called “state” would hold the packet statistics for every second, then flush the map to record the new statistics for the next second.

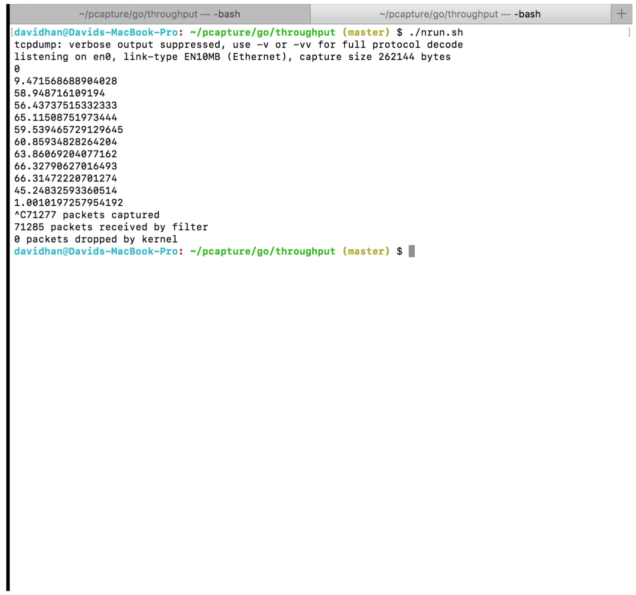
The select protocol was used to orchestrate the process between the two go channels. For the “input” channel, string operations such as the split function that delimits a string by space was used to take a line of packet output and extract the source IP, destination IP, and length in bytes. These values were used to update the “state” hash-map allocated earlier. For the “timeChannel,” it would print out and flush the “state” hash map. In addition, it would aggregate and add up all of the packet lengths from the “state” hash-map and print out the throughput. At the beginning of the case “timeChannel” in select, we also begin a timer that ends when the program is done with the aggregation. This recorded amount of time is considered a latency and it is subtracted from 1 second, and the time difference is used to calculate the throughput. To implement priorities between these two cases of go channels, the “timeChannel” was put down as a regular case, while the “input” channel is placed in lower priority by inserting it into a default block. We want to implement priorities because the program should immediately place precedence in the timeout go-routine.

Finally, in order to test the accuracy of this program, I had a shell script called nrun.sh that piped the tcpdump to the input of my program called nstdin.go:

*sudo tcpdump -l -n -i en0 udp or tcp | ./nStdin*

And using a website called **fast.com**, which record the internet speed, I can compare its value with an average of all of the throughput values from this program. Here is what happened:




The first value of 0 represented the lag that took when I started my program **first**, then started the **fast.com** function. The last value of 1.001019725… represented the extra second after the function in **fast.com** ended. With these values:

Table 1

| |
|---|
| 9.471568688904028 |
| 58.94871610919 |
| 56.43737515332333 |
| 65.11508751973444 |
| 59.539465729129645 |
| 60.85934828264204 |
| 63.86069204077162 |
| 66.32790627016493 |
| 66.31472220701274 |
| 45.24832593360514 |

The average of these values was 58.948716109194. With an expected value of 56, the percentage error was 0.05265564481 or 5.265564481%.

The future applications of this program could be used to be run on a router in the middle of Columbia in order to analyze throughput statistics throughout the day. This could reveal interesting observations like which website does Columbia uses the most or specifically which website at particular times in the day are accessed the most. This program is currently tracked on a Github.