Netlog Mining for Speed Test-Based Internet Performance Measurement

Aishwarya Joshi and Anwesha Pradhananga

Calvin University

December 19, 2024

1 Background

Ensuring a satisfactory and guaranteed service-of-quality of the Internet is important for almost all business sectors and individual users nowadays. Users can utilize the available commercial speed test platforms to monitor internet throughput and latency. Governments also set up programs, such as Measuring Broadband America, which was developed in collaboration with SamKnows, which helps to create a standard methodology to measure internet performance globally. [1] The Federal Communications Commission (FCC) no longer partners with SamKnows on fixed broadband performance data collection and is currently running the program internally. [2]

In the past, various approaches have been used to gauge Internet performance, broadly classified into passive and active measurements. Most Internet performance platforms today are based on the active measurement approach, which sends measurement probes from one or more vantage points to measure latency, throughput, packet loss, packet reordering, jitter, and many other metrics derived. The major speed test platforms, such as Ookla, Xfinity, Fast.com, Speedof.me, Cloudflare, and M-Lab, also use the active measurement approach. These commercial speed test services based on router deployment are essentially a closed system that does not share their measurement data and other technical details with the public, making it difficult for researchers to fully understand the performance of the network.

In this project, we use the commercial speed test platforms mentioned above to perform on-demand or continuous network measurements. We make use of CARROT (Configurable And RepRoducible Client for Open Speed Test), a novel measurement tool to run speed tests on these speed test platforms with user-specified parameters developed by CAIDA (Center for Applied Internet Data Analysis). The main advantage of this approach is bypassing the expensive task of deploying and maintaining yet another measurement platform. These existing platforms, particularly Ookla, are growing rapidly and cover more geographical areas and ISPs. However, the main disadvantage is that they use different methodologies and configurations to obtain the measurement data. As a result, it is very difficult to compare the measurement results obtained from different platforms. It is also impossible to change their configurations according to different measurement objectives.

2 CARROT (Configurable And RepRoducible client for Open speed Test)

To address these cross-speed test-platform issues, CARROT utilizes the major speed test platforms to conduct configurable Internet measurements. The basic idea is to use a headless browser to mimic a manual execution of speed test measurement. The toolkit also allows an experimenter to configure the measurement parameters, such as the number of measurement flows and packet size. CARROT is designed to achieve configurability, interoperability, and interpretability by allowing researchers to customize and reproduce their experiments, using multiple speed test platforms to expand the scope of single-platform measurement and reporting measurement data in a standardized format.

This toolkit captures all the Netlog data, a log file that records network-level events and states on the HTTP level for a Chromium browser, between the toolkit and the remote speed test server. The main challenge is to filter the relevant Netlog events from a large amount of Netlog data for computing delay and throughput. This task is particularly challenging because there is very little documentation available for understanding all the details in the data. Thus, we need to make sure that we wrangle the data by filtering we need to measure the throughput for the given netlog file.

3 Project Objective

The overall goal of this senior-year project is to develop

1. a Netlog analytic engine that will accurately and efficiently glean from the vast amount of Netlog data obtained from the CARROT toolkit for computing Internet performance metrics and standardize the output format,

2. use the standardized outputs to do network calculations including throughput using different measurement approaches,

3. analyze the differences in throughput based on different configurations of measurement parameters.

The engine is able to glean all the relevant network events from the six major speed test platforms: Ookla, Xfinity, Fast.com, SpeedOf.Me, Cloudflare, and M-Lab. The first five platforms use the HTTP for measurement, whereas M-Lab deploys its own measurement protocol. The toolkit allows us to configure the city, server, number of flows, the object size and whether we want to conduct a download or upload measurement through the input parameters. The output is a netlog file consisting of those events in a standardized data format which can be used by the research community to analyze the data according to their objectives.

4 Research Questions

For the purpose of this project, we are analyzing four main research questions.

1. How do we use the huge netlog data perform our own calculations?

2. How does changing the packet size, and TCP flow change the throughput calculation?

3. How do download and upload throughput calculation differ?

5 Netlog Analytic Engine

5.1 Overview

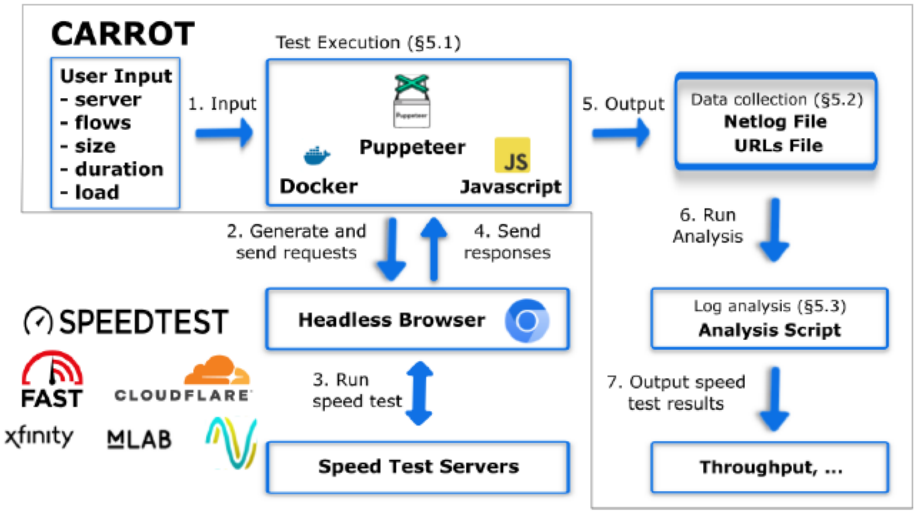

Figure 1: CARROT's design architecture (designed by Hemil, CAIDA)

CARROT utilizes NetLog [3], Chrome's network logging system, to capture all events from the browser, including network requests and responses, browser events, and timing information in a JSON-formatted trace file. CARROT uses Puppeteer, a Node.js library which provides high-level APIs for headless browser automation, allowing programmatic control over headless Chrome/Chromium browsers. Fig.1 shows the overall design architecture of CARROT. Step 1 in the figure depicts how the user provides specific configurations to run the toolkit, step 2, 3, and 4 show how the puppeteer library then runs the speedtest on the headless browser to conduct the speedtest measurements. That results in the generation of netlog file and URLs file as shown in step 5. The CARROT toolkit was capable of doing steps 1 to 5.

Our goal was to complete steps 6 and 7 of CARROT toolkit through the implementation of Netlog Analytic Engine. To this end, we use our analysis script to extract information from the netlog file using the URLs file and create two (for download) or three (for upload) JSON-formatted files: current position file, and latency file (step 6 in Fig. 7). After that, we run a script to calculate the throughput of the experiment conducted and plot the throughput values as a time series graph. We analyze these graphs to investigate our research questions.

5.2 Netlog File Data

The Netlog data that we collect after running the speed test has all the information about the experiment conducted. This information contains a chain of events related to an action, such as an HTTP transaction. This includes the start time and event type (as seen in Fig. 2).

rabbits-netlog > ookla > ookla.netlog


```
{"constants":{"activeFieldTrialGroups":[],"addressFamily":{"ADDRESS_FAMILY_IPV4":1,"ADDRESS_FAMILY_IPV6":2,"ADDRESS_FAMILY_UNSPEC
"URL_REQUEST_JOB_BYTES_READ":122,"URL_REQUEST_JOB_FILTERED_BYTES_READ":123,"URL_REQUEST_REDIRECTED":112,"URL_REQUEST_REDIRECT_JOB
"events": [
{"params":{"allow_dns_over_https_upgrade":true,"append_to_multi_label_name":true,"attempts":2,"can_use_insecure_dns_transactions"
{"phase":1,"source":{"id":5,"start_time":"34504203","type":17},"time":"34504203","type":404},
{"phase":2,"source":{"id":5,"start_time":"34504203","type":17},"time":"34504203","type":404},
{"params":{"persistent_store":false},"phase":1,"source":{"id":6,"start_time":"34504203","type":28},"time":"34504203","type":467},
{"params":{"persistent_store":true},"phase":1,"source":{"id":8,"start_time":"34504206","type":28},"time":"34504206","type":467},
{"params":{"persistence":true},"phase":0,"source":{"id":8,"start_time":"34504206","type":28},"time":"34504206","type":473},
{"phase":1,"source":{"id":9,"start_time":"34504206","type":26},"time":"34504206","type":456},
{"params":{"cors_preflight_policy":"consider_preflight","headers":"Origin: https://www.google.com\r\nContent-Type: application/x-
```


Figure 2: Netlog data from a test session of the Ookla Speed Test

The first line in Fig. 2 has a dictionary with the key "constants" where the value contains all the event types and their corresponding event type ids. For example, the event type "URL_REQUEST_JOB_BYTES_READ" has an event type id of 122 which is used in the netlog analytic engine. From second line onwards in Fig. 2, we have a list of events that each represent a specific event and its details like the event type, time, source, start_time, phase and params. For the purpose of the Netlog Analytic Engine, we will focus on the event type, time, and source which we will explain in the upcoming sections.

5.3 URLs File Data

rabbits-netlog > ookla > ooklaDownLoad_urls.txt


```
{
    "download":["https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/download?nocach
                "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/download?nocach
                "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/download?nocach
                "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/download?nocach
    "upload":[],
    "load":["https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc890
    "unload":[]}
```


Figure 3: URL file for download + load measurement

The URLs file returned by the CARROT toolkit is in a JSON format with four keys: "download", "upload", "load", and "unload" URL types. Each of these represents the kind of measurement that was performed while conducting the experiment. Download refers to conducting a download-specific measurement where the client is receiving the packets from the server and upload is when the client is sending packets to the server to congest and test the network. The load measurement is conducting the experiment

during a high network activity and unload measurement means conducting the experiment during a low network activity. Fig. 3 shows an example of a URLs file where the netlog file is download measurement with four HTTP transactions and is conducted using loaded measurement. Each URL in download refers to a unique HTTP transaction. These URLs help us to filter out the necessary events for the HTTP transactions.

5.4 Filtering the Netlog Data

The netlog analytic engine we developed is able to read the large netlog data generated by the speed test servers. There are multiple layers of filtering done to get the final byte time file, current position file, and latency file. Our analytic engine first goes through each event in the netlog file and looks for those events that have a "params" property that has an "object" with property "URL". The URL has to match with one of the URLs in the URLs file. The event should also have the event type 2 which refers to "WAITING_FOR_AVAILABLE_SOCKET". The events that match these criteria are stored in a separate list. Every event stored in this list has a source id, start time, and source type as one of its values.

```
// Check if the eventData meets certain conditions
if (
    eventData.hasOwnProperty('params') &&
    typeof (eventData.params) === 'object' &&
    eventData.params.hasOwnProperty('url') &&
    form.some(url => eventData.params.url.includes(url) &&
        eventData.type === 2
    )
) {
    if (eventData.source.hasOwnProperty('id')) {
        const id = eventData.source;
        results.push({ sourceID: id, index: index, dict: eventData });
    }
}
```

Figure 4: Filtering of the netlog data

During the same pass, a second layer of filtering is done. For this, we compare the source id of each event in the list from the first layer of the filter with the current event id in the netlog file and look for those that match. If it is a download experiment, it checks if the event is of type 122 (URL_REQUEST_JOB_BYTES_READ) or event type is 123 (URL_REQUEST_JOB_FILTERED_BYTES_READ). If that is true, and the event has a property "byte_count", that event is appended to the byte time file.

If it is an upload experiment, it checks if the event is of type 450 (UPLOAD_DATA_STREAM_READ) and it has a property "current_position", that event is appended to the current position file.

```
...
{"params":{"priority":"MEDIUM","traffic_annotation":101845102,"url":"https://speed.michwave.com.prod.hosts.ooklaserver.net:8080/download?nocache=f3db6ed0-7a80-4cee-8b41-3f9085c0820d%23&size=30000000&guid=1ecbbf5c-a9ba-4549-8f2f-08b155ad7cc9"},"phase":1,"source":{"id":5898,"start_time":"61310422","type":1},"time":"61310428","type":2},
...
{"params":{"byte_count":16384},"phase":0,"source":{"id":5898,"start_time":"61310422","type":1},"time":"61310435","type":123},
...
{"params":{"byte_count":16384},"phase":0,"source":{"id":5898,"start_time":"61310422","type":1},"time":"61310437","type":123},
...
{"params":{"byte_count":16384},"phase":0,"source":{"id":5898,"start_time":"61310422","type":1},"time":"61310439","type":123},
...
{"params":{"byte_count":16384},"phase":0,"source":{"id":5898,"start_time":"61310422","type":1},"time":"61310449","type":123},
...
```


Figure 5: An example of what events are filtered

Fig. 5 shows an example of the netlog file with only the filtered events shown. In line 2 of the figure, we can see that the event type is 2 and it has the URL listed in the URLs file. We store this event and compare its source id with all successive event's source id. Then for download measurements, the bytecount and time values of any matching successive event with event type 122 or 123 are stored in the byte time file. We do the same thing with upload measurements but by comparing with event type 450 and storing the current position and time in the current position file.

To get the latency file, we filter the netlog file to check if the particular event's URL matches the "load" or "unload" URLs in the URLs file. If the URL matches, we check for type 176 (HTTP_TRANSACTION_SEND_REQUEST_HEADERS) to add it to the latency file as the send time for that particular id, and check for type 181 (HTTP_TRANSACTION_READ_RESPONSE_HEADERS) to add it as the receive time.

5.5 Byte time file, current position file and latency file


```
[
  {
    "id": 5898,
    "type": "download",
    "progress": [
      {
        "bytecount": 16384,
        "time": "61310435"
      },
      {
        "bytecount": 16384,
        "time": "61310437"
      },
      {
        "bytecount": 16384,
        "time": "61310439"
      },
      // Manually removed the data here
      // for better visibility of other IDs
    ]
  },
  {
    "id": 5899,
    "type": "download",
    "progress": [...
    ]
  },
  {
    "id": 5900,
    "type": "download",
    "progress": [...
    ]
  }
]
```


Figure 6: Byte time file for download experiment

In Fig. 6, we see a list of dictionaries grouped by source ids. The id here corresponds to the source id from Fig. 5. The byte time file specifies the source id, experiment type (here: download) and a progress list that contains the bytecount and time. Each bytecount timestamp refers to how many bytes were sent at a given time. Each id is an HTTP transaction and the number of byte count-timestamps within each id depends on how big the packet size is and how the network sends or receives the packet as multiple HTTP requests or responses.

Figure 7: Current position file for upload experiment

In Fig. 7, we see a list of dictionaries grouped by source ids in the same way as byte time file. The difference is that the experiment type is upload and the progress list has a cumulative bytecount instead. Each of the items in the progress list is the total number of bytes that have been sent until the specified timestamp.

Figure 8: Latency file for download experiment

Fig. 8 shows how the latency values are stored. The sendTime refers to the time the measurement started and the packets were sent and the recvTime refers to the time it got a response back from the server. The ids in this file correspond to the ids in the byte time file.

The latency file and the byte time file or the current position file are significantly smaller than the raw netlog file. They were about 100 times smaller than the netlog file which makes them space efficient for researchers to use.

The network events from the JSON files are used to compute three main performance metrics: round-trip latency, download throughput, and upload throughput. The round-trip latency is performed under two scenarios: loaded and unloaded. The engine also supports different throughput computations, including those used by the speed test platforms. To calculate the throughput, we are using the average of byte counts for a given time period.

In conclusion, the network analytic engine takes in the netlog and url files generated by CARROT toolkit. It then generates the byte time file or current position file and latency file that can be used to calculate throughputs and plot the throughput charts.

6 Throughput Estimation

6.1 HTTP transaction flows

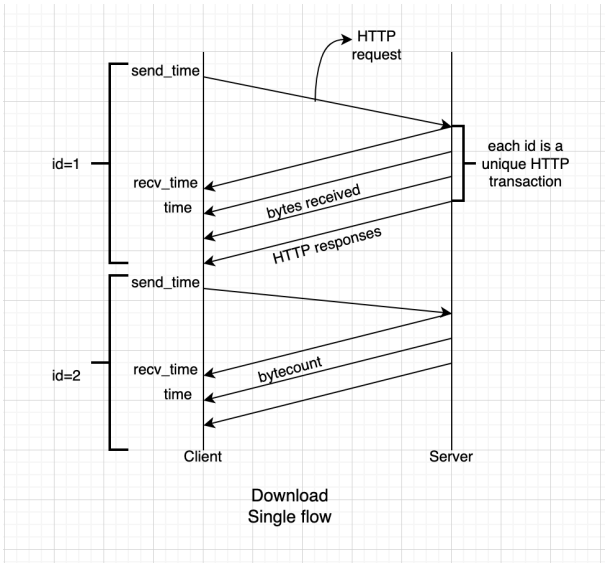

Figure 9: An example of HTTP transactions for download measurement

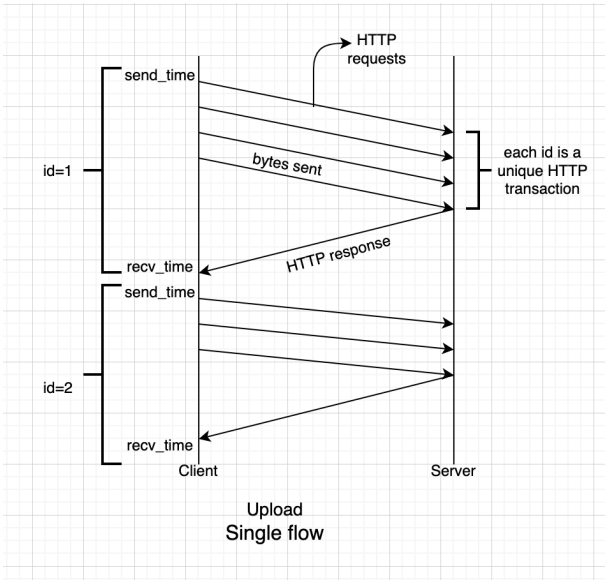

Figure 10: An example of HTTP transactions for upload measurement

Figs. 9 and 10 depict what the HTTP transactions between the speedtest server and the browser look like for download and upload a single flow. The diagrams here show how the packets flow differently for download and upload where, for download, the server sends all the packets after the recv_time, and for upload the browser sends the packets right after send_time and before the recv_time. Each id is a unique HTTP transaction and the arrows represent HTTP request and responses.

6.2 Single flow Throughput Calculation

In order to answer our research question, we calculate download throughput using the byte time file (shown in Fig. 6) and the latency file (shown in Fig. 8). Each "bytecount" and "time" value in the byte time file is used to calculate the transfer rate (bytes/time) over defined intervals. For upload throughput, the byte time file is empty, but we get the current position file (shown in Fig. 7) with the current position (the number of bytes that have been sent until that specific time) and time. This current position count is the cumulative number of bytes, so we converted it to a non-cumulative bytecount(similar to that in the byte time file). We do this by calculating the difference between consecutive current position values to get incremental bytes.

Throughput = "bytecount" at current time/(current time - previous time)

Since this method requires a time interval, the first "time" value in each id would be missing its previous "time" value. So, we use the receive time value (recv_time) from latency file for that particular id and prepend it to the list we have in byte time file. For all the other ones, we simply calculate time intervals using the difference between two consecutive time values. For upload data, we do not use the recv_time, since it is when the client receives the last response from the server in the case of upload (shown in Fig. 10).

For example, to calculate the throughput for the file in Fig.6, we use the latency file in Fig. 8. We see that the recv_time for source id 5898 is 61310432 (The time here is time in milliseconds). For throughput calculation, we look at the first bytecount in line 7 in Fig.6 where 16384 bytes were sent at time 61310435. So we calculate the throughput by 16384/((61310435 - 61310432)/1000). We divide the time duration by 1000 because we need to convert it to seconds. That gives us 5461 bytes per second which is equal to 0.044 Mbps. We do the same calculation for all the other bytecount - time pairs.

6.3 Multi flow Throughput Calculation

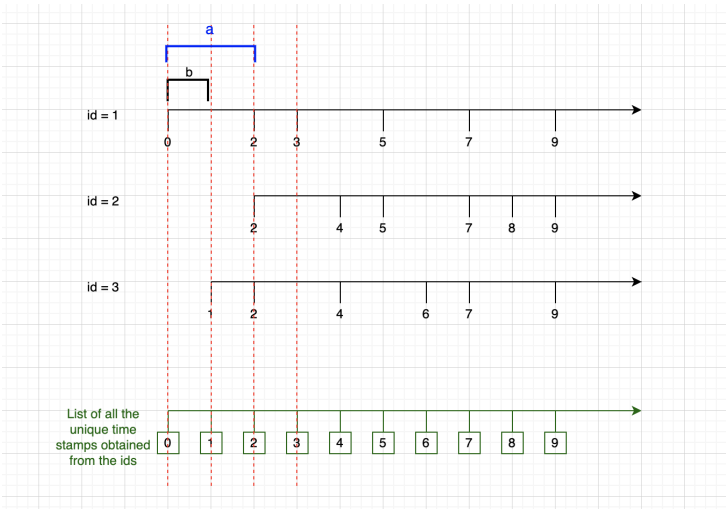

Figure 11: Multi-flow aggregation

We have a multi-step approach to calculate the multi flow experiment as there might be more than one HTTP transaction taking place at the same time. Thus, we need to aggregate the bytes across the different HTTP transactions. Initially we were assuming that the bytecount value for a given time is the exact bytecount send during that particular time. However, that idea was flawed because the bytecount was sent over the time interval (interval between the current time and the previous time) and not just the particular time stamp.

Our current approach is to go through all the HTTP transaction (ids) individually and add the time(if not already added) across those ids to a new list and sorts this final list. Fig. 11 is a mock-up example of a speed test experiment having at least 3 HTTP transactions. We can assume that this experiment has 3 TCP flows since there are three HTTP transactions overlapping with each other at the same time and there aren't any other transactions. In Fig 11, times across ids 1, 2, 3 are aggregated to get the list containing all unique timestamps. Once we have all the time stamps across ids, the code goes through each id and gets the bytecount for each time interval in the final time list. So, if multiple ids have bytecount values in the same time interval, they are added. For example, from Fig. 11, there were "a" bytes sent during time interval 0 to 2 for id 1. From the list of unique times, we know that the first time interval we should look at is 0 to 1. So, we take the ratio of time interval in the time list to the time interval of when bytes "a" was sent and calculate the same proportion of the ratio of bytes "b" to bytes "a".

This means that bytes "b":"a" = time interval (1 - 0) : (2 - 0), where "b" is the bytes sent during time interval 0 to 1 for id 1.

We calculate the bytes sent for all the times intervals for the first id and then move on to the next id. When we calculate the bytes sent for other ids, we check whether the interval already has a bytecount from previous ids showing that there is an overlap. If it does, then we add the bytes together for that particular time interval.

Our code is able to handle the case for single flow as well where the ids for single flow are also aggregated, but since these do not have overlapped times, it would be the same as combining all the ids into one big list.

Once all the ids are aggregated , we calculate the throughput using the same method we explained in the "Single flow Throughput Calculation" section.

When we have a list of throughput over time, we plot the final throughput for a given time by using Reverse Exponential Moving Average (Reverse EMA) to smoothen fluctuations and highlight trends so that a newly added throughput value does not have a huge effect on the trend. A reverse exponential moving average (REMA) is a variation of the exponential moving average (EMA) that works backward through a time series. It places a greater weight and significance on the previous data points. We calculated the REMA using Pandas DataFrame and plotted it using a time series chart. We used an alpha value of 0.1, which means that the throughput value for that particular time interval is calculated by assigning a weight of 0.1 to the new throughput and a weight of 0.9 to the throughput value that was previously calculated.

7 Analysis

7.1 Experiments Conducted

We ran the CARROT toolkit on our local machine using an Ethernet cable. We ran two experiments of each of the following in random order at two different times:

1. download_single_1_1000000
2. download_single_1_10000
3. download_single_1_30000000
4. download_multi_3_10000
5. download_multi_3_1000000
6. download_multi_3_30000000
7. download_multi_5_10000
8. download_multi_5_1000000
9. download_multi_5_30000000
10. upload_single_1_10000
11. upload_single_1_1000000
12. upload_single_1_30000000
13. upload_multi_3_10000
14. upload_multi_3_1000000
15. upload_multi_3_30000000
16. upload_multi_5_10000
17. upload_multi_5_1000000
18. upload_multi_5_30000000

The "download_single_1_1000000" experiment means that it is a download experiment with a single TCP flow with 1000000 bytes packet size.

The "upload_multi_5_30000000" experiment means it is a upload experiment with a multiple flow with 5 TCP flows and 30000000 bytes packet size.

7.2 Data Collected

7.2.1 Download Data

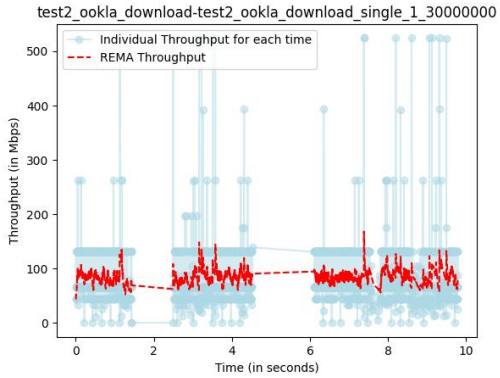

Figure 12: Single TCP Flow - 30M packet size

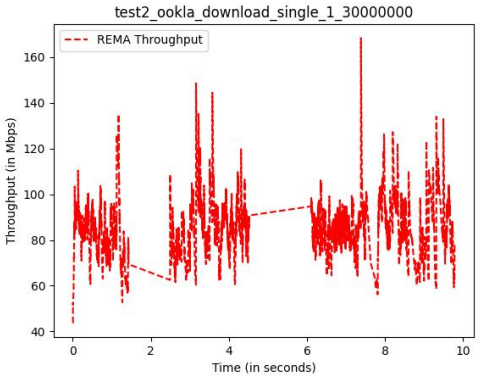

Figure 13: Single TCP Flow - 30M packet size

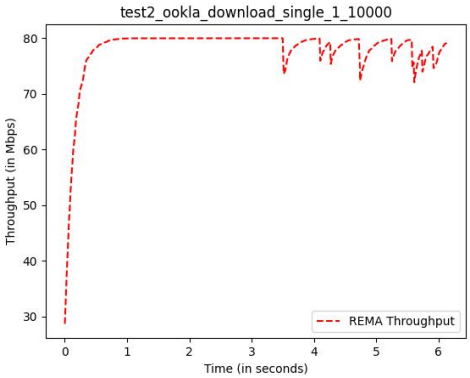

Figure 14: Single TCP Flow - 10k packet size

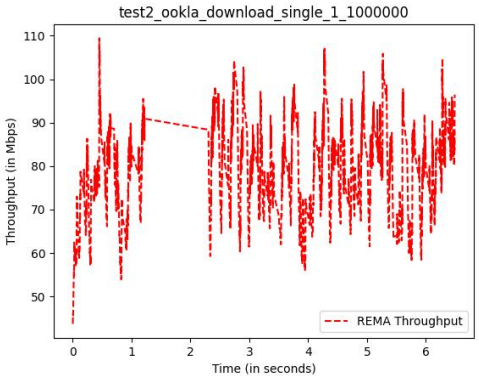

Figure 15: Single TCP Flow - 1M packet size

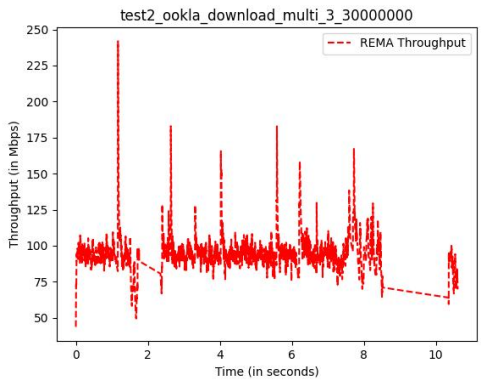

Figure 16: 3 TCP Flow - 30M packet size

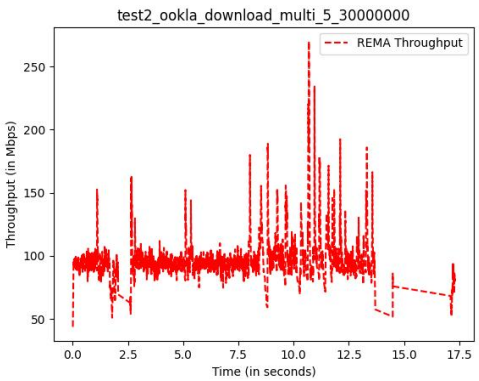

Figure 17: 5 TCP Flow - 30M packet size

7.2.2 Upload Data

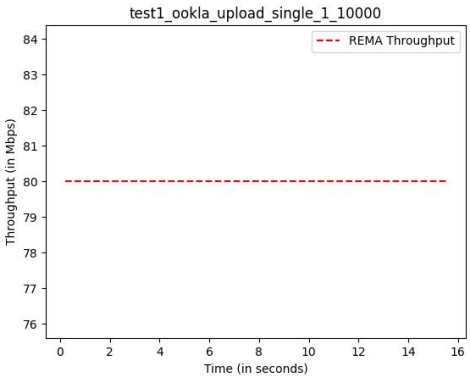

Figure 18: Single TCP Flow - 10k packet size

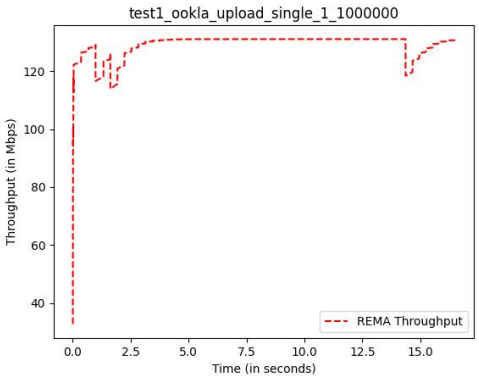

Figure 19: Single TCP Flow - 1M packet size

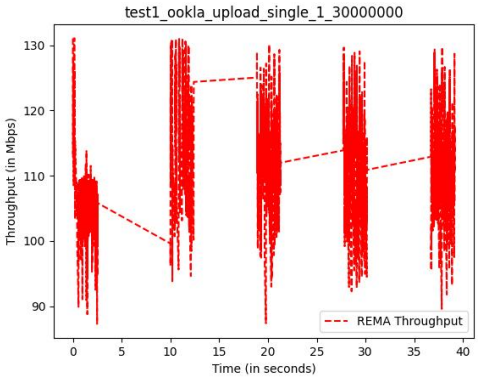

Figure 20: Single TCP Flow - 30M packet size

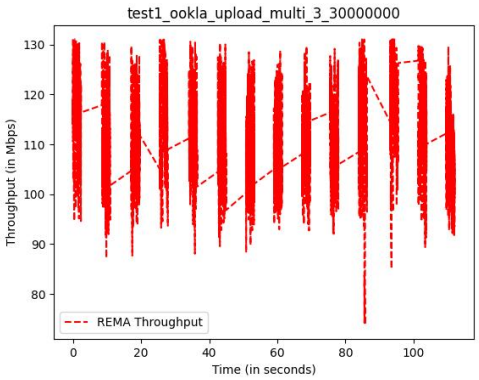

Figure 21: 3 TCP Flow - 30M packet size

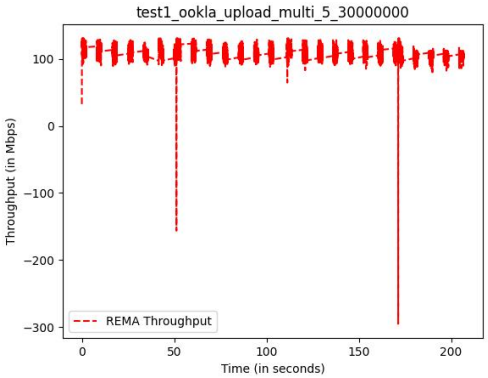

Figure 22: 5 TCP Flow - 30M packet size

7.3 Analysis of the Data Collected

We are only including selected graphs in this report in order to help answer our research questions. We have five graphs for download and five for upload experiments. There are three single TCP flow graphs with 10k, 1M, and 30M bytes packet size, and two graphs for multiple TCP flow (3 flows and 5 flows) with 30M as their packet size for both download and upload.

7.3.1 Graphical Representation

We used pandas plotting tool to plot time series charts with time (in seconds) on the x-axis and the throughput (in Mbps) on the y-axis. We used a continuous line graph to show the changes in throughput over different times. We tried two different approaches to represent the throughput data. One was plotting the individual throughput for each time interval along with the REMA throughput as shown in Fig. 12. The other was plotting only the REMA throughput as shown in Fig. 13. Despite being the same data and the REMA graph being the exact same on both charts, it looked different visually. We can tell more about the changes and fluctuations in throughput in the second approach. In the first approach, while we can see the occasional peaks and dips in throughput at each time, it is difficult to analyze the fluctuation in the REMA graph as the time scale has been zoomed out and the REMA graph looks flatter. Thus, for the purpose of this report, we will be using the graphs obtained using the second approach (only plotting the REMA graph) to analyze the collected data.

7.3.2 Comparing the Packet Sizes

When we look at the download graphs for single TCP flow, we can see how the throughput graph varies for different packet sizes. When the packet size was 10k (Fig. 14), the REMA throughput graph has less fluctuations and it increases until a certain value and levels after that. The leveling point is around 80Mbps. While there are some dips in the graph it still averages at around 80Mbps. We know that throughput was bytes sent over a certain time interval. We are sending 10k bytes at a time and it is generally sent within a millisecond. So, the throughput would be

$$(10000 * 8/10^6 Megabits)/(1/1000 seconds) = 80 Mbps$$

In the 1M packet size (Fig. 15), the throughput fluctuates between 60 to 100 Mbps. In 30M packet size (Fig. 13), the throughput fluctuates mostly between 70 to 110 Mbps. For both 1M and 30M packet sizes, the fluctuations were comparatively a lot higher than for 10k packet size. The peaks of the graph were also much higher than 80Mbps. This shows that the 10k packet size was not enough to congest the bottleneck of our Ethernet connection. The fluctuations arise because different amount of bytes is sent at each time and the time interval varies as well. With 10k packet size, the whole packet can be received as one or two HTTP responses. However, with 1M and 30M packet sizes, each packet has to be divided into multiple HTTP responses due to the limit on how many bytes can be sent through the network at once.

We can see a similar trend for the upload graphs for single TCP flow. Fig. 18 shows

that for 10k packet size the throughput is consistent at 80Mbps. Fig. 19 shows that with 1M packet size, the throughput is much higher at about 120 to 130 Mbps. Fig. 20 shows that for 30M packet size, the throughput fluctuates between 90 to 130 Mbps. We conjecture that we see 5 specific chunks in 30M graph because there were 5 HTTP transactions and there was delay between each of them. In each HTTP transaction the packet was split into multiple HTTP requests so that they can be sent over the network – hence the fluctuations.

This experiment shows that a higher packet size is important to get the actual throughput value by congesting the network.

7.3.3 Comparing the TCP Flows

To compare the TCP flows, we kept the packet size constant at 30M. We varied the TCP flows to be single, 3 flows and 5 flows. For download, the single flow shows fluctuations between 70 to 110 Mbps and has gaps between different chunks of data (the reason for this is the same as we mentioned earlier – multiple HTTP transaction with delays between them). For both 3 and 5 flows, there is much less gap between the chunks of throughput data but the peak is higher than with single flow with a maximum value of almost 250 Mbps. The reason for less gap is probably because in single flow, there are certain time intervals with no data flow in between different HTTP transaction because one has to end before the next one can start. However, for multi flow, there is a continuous data flow without many instances of no data flow because of overlaps of flows and also one might have data traffic even when others are idle. The higher peaks in 3 and 5 flows can be investigated more.

For upload, we can notice that as the number of TCP flow increases, the number of chunks in the graph also increases. The 30M packet size unexpectedly has a negative value which needs to be investigated more on.

This experiment suggests that multiple flows might give us a better idea of the throughput value since it has less idle time.

7.3.4 Comparing Download and Upload Experiments

When we compare the download and upload experiment graphs, we can see that download has a continuous flow while upload generally has throughput value in chunks. Both of them show a similar trend across different packet sizes and TCP flows.

8 Problems encountered

1. One of the problems we encountered was aggregating the multiple TCP flows. We were considering different ways we could use to aggregate the flows together and what made the most sense. We made a list of all timestamps. Then we aggregated all the bytecounts across the ids that are in the time interval and calculate the throughput for the multi flow.

2. The puppeteer version we are using to run the speedtest in the headless browser is not compatible with the arm architecture. Thus, we were unable to deploy our code in Raspberry pi and run measurements there.

3. There is a repository of speedtest measurement experiments conducted in fabric testbed portal. When we tried using the netlog files there, the throughput values were higher than expected. Thus, we conducted our own experiment in our local machine

9 Future Work

1. We have currently only done our experiment on Ookla server. The CARROT toolkit is able to generate the byte time and latency files for Xfinity, Fast.com, Speedof.me, Cloudflare (not M-Lab). However, the data we have collected is specific to Ookla and the next step is to extend our research questions to cover other speedtest platforms.

2. In the charts we have plotted, we see some time gaps between a few time intervals. The time gap is still present when we look at the pcap files we got from Wireshark. The next step is to investigate why the gap exists.

3. Currently, the plots show the exponential moving average of the aggregated throughput for each time interval. We would like to see how each flow affects the weight of the throughput at each time interval. We can do this by making a heatmap to show the overlaps at each time intervals to get a better estimate on how each flow affects the final throughput.

4. We would want to collect more data from different locations to answer our research questions a broader scale.

5. A console with data visualization can be developed to monitor the continuous speed test measurement from different vantage points and speed test platforms.

10 Acknowledgment

We would like to express our deepest gratitude to Dr. Ricky Mok from CAIDA for his invaluable guidance and support throughout this project. His expertise and insights were helpful in shaping the direction and outcomes of our work.

We are also immensely grateful to Hemil Panchiwala, a graduate student at UCSD, for his generous support and encouragement. His contributions were crucial to overcoming challenges and ensuring the success of this project.

Finally, we extend our heartfelt thanks to the Computer Science Department and especially to Professor Rocky Chang for providing the resources and opportunities that made this project possible.

References

[1] SamKnows. *SamKnows Agents*. `https://samknows.com/technology/agents`.

[2] Eric W. Burger, Padma Krishnaswamy, and Henning Schulzrinne. "Measuring Broadband America: A Retrospective on Origins, Achievements, and Challenges". In: *ACM SIGCOMM Computer Communication Review* 53.2 (Apr. 2023), pp. 11–21. ISSN: 0146-4833. DOI: 10.1145/3610381.3610384.

[3] The Chromium Projects. *NetLog: Chrome's network logging system*. https://www.chromium.org/developers/design-documents/network-stack/netlog.