Netlog Mining for speed test-based Internet Performance Measurement

Aishwarya Joshi, Anwesha Pradhananga

December 2024

1 Background

Ensuring 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 their 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 derived metrics. 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 the 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 network performance.

In this project, we use the above-mentioned commercial speed test platforms 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 they 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 all the existing 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, utilizing 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 for a 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 output to compute throughput using different measurement approaches, and 3) analyze the differences in throughput based on different configurations.

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

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. As we run the code, we get the netlog data that has information about the metadata grouped by IDs, each containing a chain of events related to an action, such as an HTTP transaction. This includes the start time and event type (as seen in Figure 1). The netlog analytic engine we developed is able to read the large netlog data generated by the speed test servers.

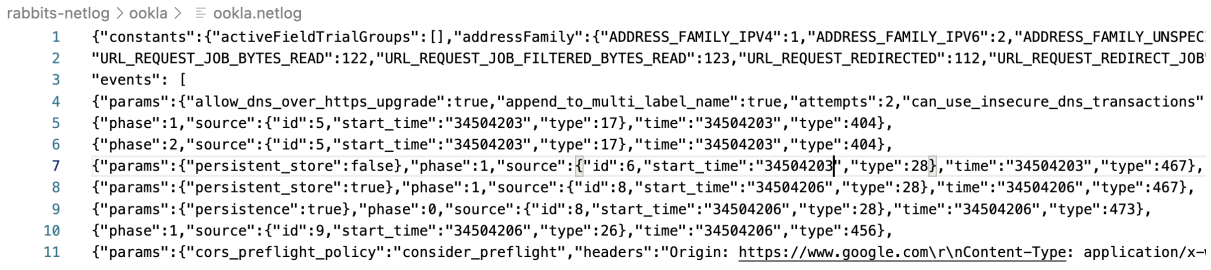

Figure 1: Netlog data from Ookla Server

In the figure above, we can see that the first line has an object that has a key "constants". This object allows us to know what number is associated with each value. For example, "URL_REQUEST_JOB_BYTES_READ" has an id of 122 which is used in the netlog analytic engine.

In our netlog analytic engine, the first step is to read the URL file which has the URLs that correspond to the particular HTTP transaction that we need for our analysis. To get those URLs, we monitor the actual requests that are sent from the browser when we run the speed test using their websites (we do this by checking the Network tab in Inspect of the browser). We find the relevant requests for download, upload and then update the unique identifiers, object size, etc. in the URL as per our requirement.

These URLs have a URL type that indicates whether the event is download or upload, and load (when the measurement is ongoing) or unload (when there is no measurement). Figure 2 is an example of a URL file where the netlog file was a download measurement with 4 TCP flows with a load event type. Our netlog analytic engine uses these URLs to filter out the necessary events.
Then, we use this URL file and some other criteria such as if it has a property called "params" and if "params" is an object with another property "URL" to filter the event data.
The engine uses the URL file and filters out certain events whether the event has a "params" property that has an "object" with "URL". Using the information in the "constants" object, we found that we need events with type 122 (URL_REQUEST_JOB_BYTES_READ)

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=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
            "https://speedtest.spacelink.com.prod.hosts.ooklaserver.net:8080/hello?nocache=bc89
    "unload":[]}
```

Figure 2: URL file for upload

and type 123 (URL_REQUEST_JOB_FILTERED_BYTES_READ) for download data and type 450 (UPLOAD_DATA_STREAM_READ) for upload data.
It then stores the id, index, and entire event data in the array 'results' for further filtering. This filtered data is stored in byte_time_list.json, which is significantly smaller than the raw netlog file. The netlog file was about 35-55MB while the byte_time_list.json file was reduced to about 60-300KB.

```
// 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 3: Filtering eventData by comparing with certain criteria

All the events in the 'results' array with the current selection for 'urltype' is stored in a JSON file that has the format shown in Figure 4. This is a JSON file where we have the information about how many bytes were sent at a given time and it is grouped by IDs (unique HTTP transactions). Using the filtered events, we also create another JSON file where we store the sendTime and recvTime of each event. This is the round trip time (RTT) of the when the handshake happened. 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. As we can see in Fig. 4, each id in the byte_time_list.json corresponds to the ids in latency.json.

The network events from the JSON file above are used to compute three main perfor-

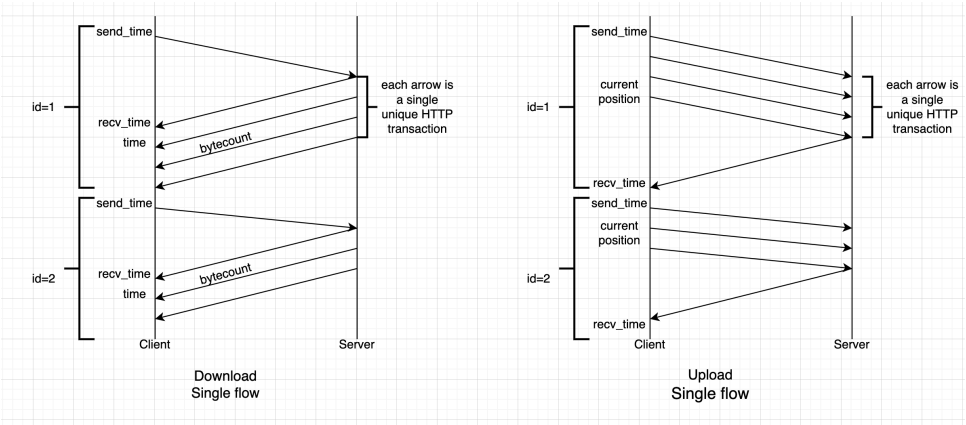

Figure 4: Filtering eventData by comparing with certain criteria

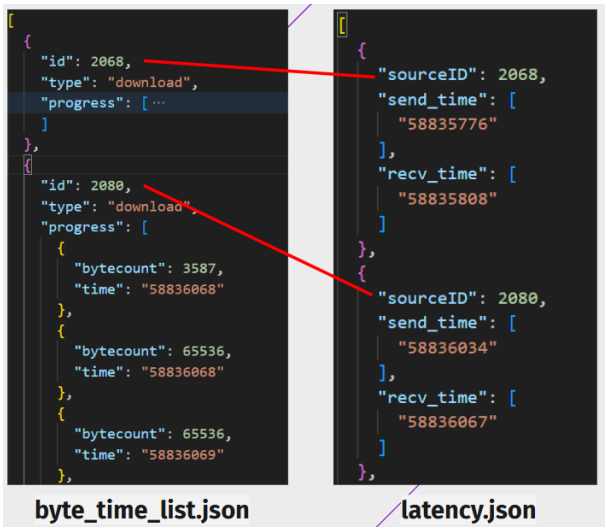

Figure 5: The two JSON files grouped by each HTTP transaction

mance 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.

This engine is seamlessly integrated with the CARROT toolkit to perform large-scale measurements from different cloud platforms and network measurement platforms. Therefore, the engine must be able to produce the measurement metrics very efficiently in terms of the speed and use of memory.

6 Throughput Calculation

In order to answer our research question, we calculate download throughput using the byte_time_list.json file (shown in Fig 5) and the latency.json file (shown in Fig 5). Each "bytecount" and "time" value is used to calculate the transfer rate (bytes/time) over defined intervals. For upload throughput, the byte_time_list.json file is empty, but we get

a file with the current position (the number of bytes that have been sent until the current time) and time. This current position count is the cumulative number of bytes, so we converted it to a non-cumulative byte count (similar to that in the byte_time_list.json file).

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 from latency.json for that particular id and prepend it to the list we have in byte_time_list.json 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 receive time, since the receive time is when the client receives the last response from the server in the case of upload (shown in Fig. 4).

To calculate the throughput, our python code in 'throughput.py' firstly goes through all the ids and adds the time (if not already added) in a new list and sorts this final list. 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 byte count values in the same time interval, they are added. The ids for single flow is also aggregated, but these do not have overlapped times so it would be the same as combining all the ids into one big list. Once all the ids are aggregated, we calculate the throughput. 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.

7 Analysis

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 pupeeter 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. Extending our research questions to cover other speedtest. platforms
2. Analyse the data that we collect from different physical locations by deploying our code in multiple machines across different places.
3. A console with data visualization can be developed to monitor the continuous speed test measurement from different vantage points and speed test platforms.

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.