Observing TCP Round Trip Times with FlowScope

Christian Wahl
Advisor: Simon Bauer, M. Sc.
Seminar Innovative Internet Technologies and Mobile Communications SS2018
Chair of Network Architectures and Services
Department of Informatics, Technical University of Munich
Email: wahlc@in.tum.de

ABSTRACT

Most of our network software implementations depend on the Transmission Control Protocol (TCP). Thus, the performance and reliability of these solutions depend on a reliable protocol. A fast connection requires a timely acknowledgement of a received data packet. We want to measure this timespan, between the sending of the packet by the source and the arrival of the acknowledge by the destination. This timespan is called round trip time (RTT). Therefore, we present a solution to observe this timespan by employing FlowScope to measure the round trip time with the help of a suited extension module. Furthermore, we evaluate with this proof-of-concept whether FlowScope [4] is a appropriate tool for this purpose and it can collect suitable data for further network focused research.

Keywords

TCP, FlowScope, RTT, Round Trip Time, Network Monitoring

1. INTRODUCTION

The Transmission Control Protocol (TCP) is used to transfer data over networks which could be deficient and as a consequence lose data packets. TCP [12] ensures a successful transmission by awaiting the confirmation before sending the next packet. TCP is the most used network transmission protocol, accounting for around 90 percent of the internet traffic [5, 8].Thus, weaknesses of the transport route will affect a majority of internet users. One of this weaknesses could be a link between the source and destination with a raised latency which leads to increased response times of the connection or in extreme cases, i.e. in case of a failure to timeouts of the session due to packet loss. To monitor the time between sending a data packet and the acknowledgement of the same in (near) real time, we present a solution based on FlowScope [4], extended by a module created for this purpose.

We structure the paper as follows: First, we introduce relevant terms and definitions in Section 2. Second, we discuss related work in Section 3. After that, we show the implementation in Sections 2 and 4. Then, we look at the results of the implementation in Section 5 and look at another way to mitigate the shortcomings found during the evaluation in Section 6. Furthermore, we propose possible improvements to the solution in Section 7. Before concluding the paper in Section 9, we encourage our readers to reproduce our research on different input data in Section 8.

2. TECHNICAL BACKGROUND

In this section we give an overview over employed tools, define commonly used terms and describe the investigated protocol.

2.1 Network Flow

A network *flow* is defined by its unique five-tuple. This tuple is formed by the *source ip address* and *port*, *destination ip address* and *port*, and by the used protocol (in our case TCP):

$$\text{flow tuple} = (\text{src}_{\text{IP}}, \text{src}_{\text{port}}, \text{dst}_{\text{IP}}, \text{dst}_{\text{port}}, \text{protocol})$$

2.2 Round Trip Time

This term defines the timespan between the point when the sender transmitted his packet and when it receives the acknowledgement that the packet has been received. This timespan is called round trip time (RTT). The authors of [7] discuss the different causes for the delay. The delay might be caused due to the transport medium (i.e., based on the physical properties of light or electrons), network (caused by queuing and congestion) or processing delay, etc.

Formally, we define it as the difference between the point in time where the packet with sequence number x arrives and the point in time when the packet arrives which acknowledges packet x (refer to Section 2.3 and to [12] for an explanation)

$$t_{\text{seq}} = \text{time stamp of packet with sequence number } x$$
$$t_{\text{ack}} = \text{time stamp of packet}$$
$$\text{with acknowledge number } x + 1$$

$$RTT = t_{\text{seq}} - t_{\text{ack}}$$

2.3 Protocol Background

TCP is embedded into the protocol stack at Layer 4 of the OSI Model on host computers [12]. It assumes to be able to get data-segments from the layer below and to forward received data to the layer above. It is packet oriented and sends and receives the packets from the surrounding layers. TCP is a versatile protocol and has a number of optional header extensions. The protocol is defined in RFC 793 [12]. However, as we only want to analyse the round trip time we

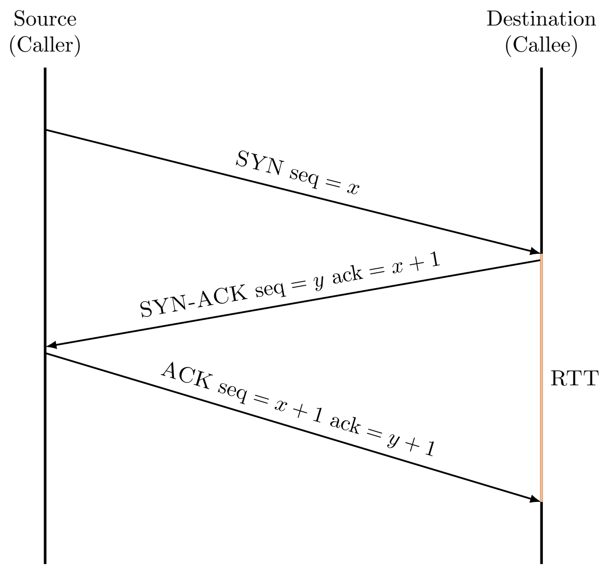

Figure 1: The TCP Three-Way Handshake [12] with Round Trip Time

focus on parts of the protocol that are important for this analysis.

A TCP packet includes the SYN-bit that is set (= 1) when the caller wants to initiate a new connection, the ACK-bit is set if the sender acknowledges a packet. TCP uses a procedure called "Forward Acknowledge", this means the acknowledge number is advanced by one compared to the acknowledged sequence: When a packet with acknowledge number x is received, then the packet with sequence number $x - 1$ is acknowledged. Furthermore, a TCP connection is initiated with the Three-Way Handshake depicted in Figure 1 and ended by the two parties sending a packet with the FIN-bit set and both acknowledging it.

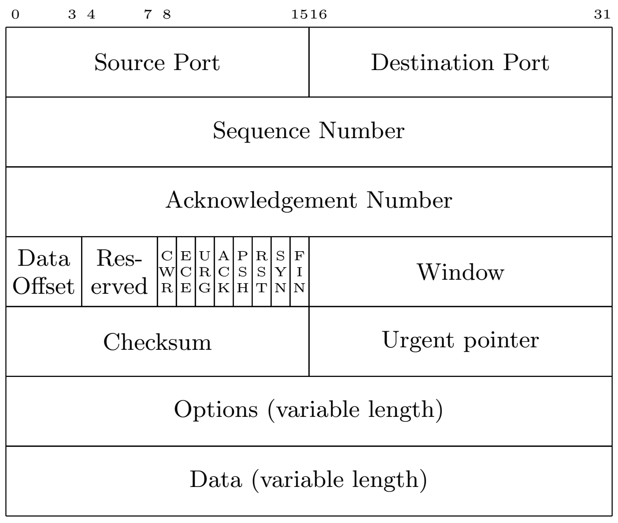

Figure 2: TCP header [12, 15]

2.4 Employed Software

During the development and test of the FlowScope module we used MoonGen [3] to replay a stored packet trace containing a SSL-scan and FlowScope [4] to capture and analyse the replayed traffic. Both tools are based on the *libmoon* [2] framework which encapsulates the *Data Plane Development Kit (DPDK)* [17]. The *libmoon* library extends the *luajit* [11] *just-in-time* compiler with interfaces to *DPDK* and a multi-threaded runtime model.

2.4.1 MoonGen

MoonGen is a customizable high-speed packet-generator [3]. It was used to generate test traffic that includes real TCP session interactions (including sequence numbers with the appropriate acknowledge numbers). This was accomplished by utilizing the included script to replay a captured *pcap* (refer to Section 8 for more information).

2.4.2 FlowScope

To capture and analyse the traffic generated by MoonGen, FlowScope was employed. It is designed for networks with a link rate exceeding 10 Gbit/s and according to the authors has been successfully tested with rates above 100 Gbit/s [4]. Most of the high-level functionality is written in *lua*. FlowScope is multi-threaded and due to the limitation of the *luajit* [11] each thread runs a separate instance of the *luajit* environment. With the means provided by *libmoon* it is possible to communicate over thread-boundaries. Each thread of FlowScope has a defined purpose [4, 14]:

Inserter This thread reads the packets from the NIC and stores it in a ring buffer.

Analyzer This thread runs the user module on a packet and stores the packet in a hash map according to the result of the user module. The required functions are shown in Section 4.1.

Checker The checker runs in configurable intervals and checks which flows are outdated and therefore can be removed or need to be preserved.

Dumper The dumper thread writes the captured packets to a *pcap*-file. The content of this file is determined by a filter expression written in the *packet-filter language* (similar to expressions used by `tcpdump`) and will be compiled to a *lua* function with pflua [20].

3. RELATED WORK

Observing the round trip time of TCP flows has been the topic of many research reports [5, 7, 16] and product of tool development [9, 21]. Two of these software pieces analyse the round trip time as an input value for further analysis of tcp flows. However, in this paper we focus on round trip time calculation. In [21] Zhang et al. present *T-RAT*, a tool to analyse the TCP flow rates. *T-RAT* takes traces of TCP connections and tries to infer the limiting factor on the flow rate. Integral to this analysis is the calculation of the round trip time (RTT): The RTT is calculated based on groups of packets forming a flight (as illustrated in [16]) based on the inter-arrival times. It then classifies the flight into one of the TCP phases (e.g., *slow start*, *congestion avoidance*, etc.). Shakkottai et al. report though, that this approach is controversial and not well defined [16]. Following the analysis of the TCP flow rates, Mellia et al. present in [9] another software to analyse and create different types of statistics on the IP and TCP level. The tool needs captured network traces (i.e., it only works offline) and will then extract the attributes and features of the flows needed for the statistic (on commodity hardware and retroactively). One of these attributes is the round trip time, where the authors gather the minimum, maximum and average RTT with respect to lost and retransmitted packets, also called the "Karn's algorithm" [6]. For the computation of the round trip time multiple models exist: Some only take the RTT during the Three-Way Handshake (illustrated in Figure 1) into account [5], while others evaluate if the RTTs are different during the phases of a flow [7]. Additionally, Fraleigh et al. point out in [5], that at the start of this century, it might have been more difficult to acquire a suitable dataset as it is now, due to limitations on storage space and available transmission capacities. More information about FlowScope can be found in [4], about MoonGen in [3].

4. IMPLEMENTATION

In this section we describe the requirements that FlowScope imposes on a user module, discuss the algorithm and environment we used to test our solution.

4.1 Required Attributes by FlowScope

FlowScope has the ability to be extended by a user module. This module needs to provide a minimum set of functions and attributes in order to use the flow tracking that FlowScope offers. See the Github repository [18] for the implementation. The lua module needs to export all attributes and functions that are listed below, only attributes marked with **optional** can be left out. In order to retain a clear structure, the required attributes and functions are sorted by the thread they belong to (this information was extracted from the example modules and by reading the source code in [14], especially the `exampleUserModule.lua` [13]). In order to implement a user module that is capable of measuring the RTT of TCP-flows, we implemented the following functions:

4.1.1 Flow Tracking

The *Analyzer* thread carries out the tracking of the flows. After a packet is inserted by an *Inserter*, the `flow key` needs to be extracted in the `extractFlowKey` function. Next, the `handlePacket` function is called with this packet.
C-structure types need to be defined with `ffi.cdef` function of the *luajit ffi* library [10] before they are known to the library. If a parameter needs to be set to a name of a *C-structure* the variable needs to include the whole name, i.e., "`struct ip`".

`mode` Either "`qq`" (to use the Queue of Queues data structure [4] / the ring buffer) or "`direct`" (to directly access the *NIC* without buffering)

`stateType` The type of the *C-structure* used to carry the state of this flow. Due to the implementation of the hash map in the background, this structure is limited to 128 B in size. With a change to the definition of the hash map in FlowScope this could also be enlarged.

`defaultState` This is an **optional** map datatype. The `stateType` will be initialized with the values defined here. Before applying the default values, the memory space designated for this structure is filled with zeros. Then, the default values are applied (see the description of `ffi.new` in [10] for an extensive description). If this map is undefined or empty, the values are set to 0.

`flowKeys` A `flowKey` is a *C-Structure* that defines how and on which attribute the packets should be associated. In case of a IP-flow, the five-tuple outlined in Section 2.1 constitutes a suitable flow key. The `flowKeys` attribute expects a map that includes all *C-Structure* definitions for possible flows. The size of a flow key is limited to 64 B.

`function extractFlowKey(buffer, flowKeyBuffer)` The function is used to extract the features of a flow from the `buffer` (which contains the whole packet including all headers from layer 2 and above) and store it in the `flowKeyBuffer`. Therefore the `flowKeyBuffer` is a pointer to a memory location that is large enough to store the largest `flow key`. The function returns **false** if the packet should be ignored and not stored at all or (**true**, i) if it should be stored, where i indicates which flow key was used (based on the `flowKeys` map).

`function handlePacket(flowKey, state, buffer, isFirstPacket)` The `handlePacket` function extracts the information from a packet (contained in `flowKey` and in the packet itself, in the `buffer`) and stores it into the flow `state`. The `state` variable is initialized with the contents of `defaultState` if `isFirstPacket` = **true** or it contains the state of the flow as it was changed by the last `handlePacket` invocation. If the first packet of the flow is handled, `isFirstPacket` is **true**. The function must return either **true** if this flow should be archived after it has expired (see Section 4.1.2) or **false** if this flow should not be archived.

4.1.2 Checker Thread Configuration

The *Checker* thread observes the Queue of Queues buffer for expired flows and marks flows that are expired (i.e., flows that are seen as finished) for the dumper thread (refer to the next section for a explanation). All of the attributes of the *Checker* thread are **optional**.

`checkInterval` This attribute defines the interval in which the *Checker* thread runs (in seconds). If the variable `checkInterval` is not defined, the *Checker* thread is disabled and will not run, in this case none of the functions prefixed with "`check`" are called.

`checkState` This *lua* map defines the internal state of this *Checker* thread.

`checkExpiry(flowKey, state, checkState)` This function checks whether the flow with `flowKey` and `state` is expired. It also has access to the *checker*'s state via the `checkState` variable. The `checkExpiry` function should return **false** if this flow is still active. If the flow is expired, the expiry-timestamp ts is smaller than the current time, the function should return a tuple with $(\mathbf{true}, ts)$ and is, as a consequence, removed from the hash table and from the rules of the dumper threads.

`checkInitializer(checkState)` The `checkInitializer` function is called after the state of the checker thread(s) has been initialized by FlowScope. It can be used to initialize the state of the *Checker* thread (`checkState`) or to initialize libraries once per thread.

`checkFinalizer(checkState, keptFlows, purgedFlows)` `checkFinalizer` is handled similarly to `checkInitializer`. However, it is invoked when the *Checker* thread is about to be terminated.

4.1.3 Dumper Thread Configuration

The *Dumper* thread writes the captured packets based on a Packet-Filter expression to a *pcap*-file.

`maxDumperRules` This variable limits the number of applied packet-filter rules. If set to 0 FlowScope is not able to store any dumping rules and thus, no packets are saved in a *pcap* file.

`buildPacketFilter(flowKey)` The `buildPacketFilter` function creates a packet filter based on the `flowKey`. To pass the filter to FlowScope it is required to return a string in the *packet filter language* [19].

4.2 Employed Algorithm

We use a SYN based approach as proposed in [16]. The authors found that, despite the simplicity of the algorithm, other tested algorithms might not offer big advantages over the SYN based approach. Whereas the SYN based approach is easy to implement and is comparatively less resource intensive, which makes it suitable for online RTT estimation. The larger challenge in this proof of concept was the integration into FlowScope. As outlined in Section 4.1.1, it is not possible to store an unbounded amount of data per flow in the hashmap managed by FlowScope. This implies that we are limited to a maximum of 128 B for the flow state. 34 B are used by counters for the minimum, average and maximum RTT, together with counters for the number of packets and the size observed. Our data structure to store the sequence number and the timestamp is 8 B in size (Figure 3), so we can store (128 B − 34 B) : 8 B = 11.75 sequence numbers with a timestamp. So we can store eleven sequence numbers together in a list. Despite the implementation of the `extractFlowKey` function, the main point of the implementation was the creation of the `handlePacket` function.

5. EVALUATION OF THE IMPLEMENTATION

Figure 4 shows the average round trip time during different flows between a pair of IP addresses. These numbers are gathered based on a replay of a SSL-scan stored in a *pcap* file with MoonGen [3]. We observed that all reported round trip times are 0. We conjecture that a replay of a stored SSL-scan might not be the most suited material to test the algorithm. One of the most probable causes for the result in Figure 4 could be, that most of the ports are closed and did not send a packet or terminated the connection shortly. Based on Figure 5, we further assume that the maximum of 11 sequence numbers are a too little subset to calculate the round trip time for this packet sample.

```c
struct timestamped_tuple {
    uint32_t sequence_number;
    uint32_t timestamp;
};
struct flow_state {
    uint32_t byte_counter;
    uint32_t packet_counter;
    uint64_t last_seen;
    double avg_rtt;
    uint32_t max_rtt;
    uint32_t min_rtt;
    uint8_t tracked;
    uint8_t rtt_index;
    struct timestamped_tuple rtts[11];
};
```

Figure 3: The Used State C-Structure

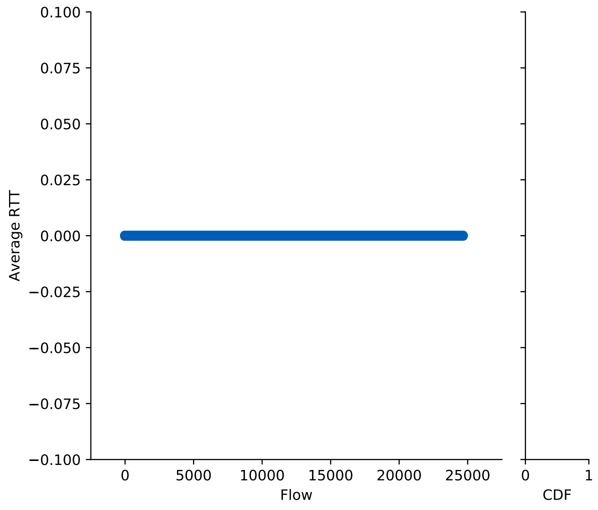

Figure 4: Round Trip Time per Flow Between Two Randomly selected IPs

6. VALIDATION

To further validate the aforementioned solution to evaluate the round trip time with FlowScope. And to confirm our assumptions about the number of sequence numbers we need to store, we created another approach that uses a dynamically expanding list of sequences.

6.1 Implementation of the New Approach

Therefore, we created another user module that is based on the previous approach but employs a dynamically expandable list to store the round trip times and sequence numbers. The captured samples of sequence numbers and round trip times are stored in a doubly linked list. The nodes of the list are *heap*-allocated and managed by the user module with `malloc` and `free`. Hence, we are not limited by the space constraints of the FlowScope hash map. As we are now able

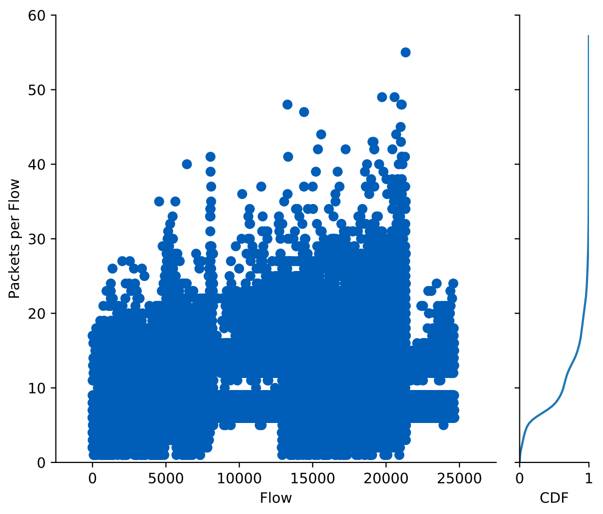

Figure 5: Packets per Flow Between Two Randomly selected IPs

to store every amount of sequence number that we capture, we are able to compute the average RTT after we finished collecting the sample. This opens the possibility to do further analysis on the round trip time. The improved flow state C-structure is depicted in Figure 6.

6.2 Evaluation of the New Approach

We tested the new approach again with an excerpt of a SSL-scan as before, albeit an different part of the SSL-scan. To reproduce this test with different data refer to Section 8. We found that the new approach yields results that we already expected from the implementation that used only a limited number of stored sequences. The data depicted in Figure 7 and Figure 8 was captured with the user module outlined above and flows with no RTT samples have been filtered out. We ascertain that the RTT is highly variant over the number of flows captured in this sample (Figure 7). To explore the distribution of the round trip times, the ten highest average round trip time are depicted in Figure 9. These respond initially promptly with an SYN-packet and then need apparently more time to process the next messages.

7. FUTURE WORK

Our original solution did not yield expected results. Conjectured cases for the shortcomings have been the properties of the SSL-scan or the previous calculation of the round trip times. However, these shortcomings have been caused by the limited amount of stored sequence numbers. As we found by employing a different approach in Section 6. To evolve this proof-of-concept into a better employable tool, it needs to be further tested with different packet traces that exhibit a different traffic pattern than a large amount of short lived connections or connection resets which might highlight further shortcomings of the two approaches.

Furthermore it needs to be evaluated, if FlowScope is the right tool for this task. Section 4.2 outlines the limits of FlowScope for the initial approach. The main limitation of FlowScope was the limited amount of storage available for

```
struct List_node {
  uint32_t sequence_number;
  uint32_t timestamp;
  struct List_node* next;
  struct List_node* prev;
};

struct RTT_List {
    uint32_t rtt;
    struct RTT_List* next;
    struct RTT_List* prev;
};

struct flow_state {
    uint32_t byte_counter;
    uint32_t packet_counter;
    uint64_t last_seen;
    uint32_t max_rtt;
    uint32_t min_rtt;
    uint8_t tracked;
    struct List_node* seq_list;
    struct List_node* ack_list;
    struct RTT_List* rtts;
};
```

Figure 6: The Improved C-Structure for the Flow State

each flow. Albeit, this could be mitigated in the second approach by using a heap allocated doubly-linked list which is not managed by FlowScope. However, manually allocating and disposing memory is prone to errors and an automatic solution to memory management is preferred here.

8. REPRODUCIBLE RESEARCH

We invite our readers to test our software and reproduce our findings with different input data. For this reason we provide the FlowScope modules at [18]. To run FlowScope with the user modules introduced herein, to follow the steps delineated below:

1. Server A and B need to be directly connected via network cards that are supported by the *libmoon* framework.
2. Install MoonGen [3] from [1] on server A.
3. Continue with installing FlowScope from [14] on server B.
4. Now, the repository with the FlowScope user module referenced by [18] needs to be cloned alongside the FlowScope installation.
5. Run `./libmoon/build/libmoon lua/flowscope.lua ../flowscope-tcp-rtt-analysis/src/TCPRTTTime Analysis_avg_w_seq.lua 0` on server B. This starts the collection of analysis data. To use the second approach from Section 6 substitute `TCPRTTTimeAnalysis_avg_w_seq.lua` with `TCPRTTTimeAnalysis_w_malloc.lua`
6. Now, to replay a captured *pcap* file, `./build/MoonGen examples/pcap/replay-pcap.lua -r 1 0 <path-of-pcap>`

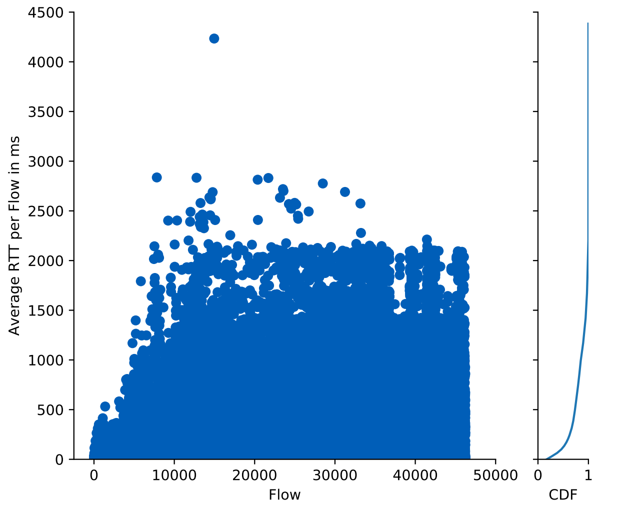

Figure 7: Average Round Trip Time Per Flow With the New Approach

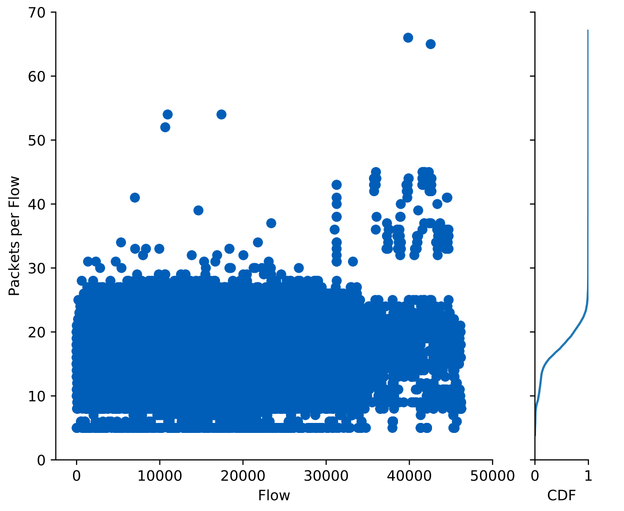

Figure 8: Packets Per Flow With the New Approach

needs to be run on server B within the MoonGen directory.

If these instructions deviate from the `Readme.md` of [18], please follow the instructions of the `Readme.md` as these instructions will provide up to date instructions.

9. CONCLUSION

With this paper, we contribute to the ongoing evolution of a toolset to measure and analyse high-bandwidth networks passively and during operation.

We started with a topic-focused introduction to the TCP protocol and the round trip time as a metric to measure the latency of networks. Then we continued to outline the requirements of FlowScope to a user module and the implementation of a module that collects data to measure the round trip time. During the implementation, FlowScope showed some limitations. Mainly, the inability to store expanding flow status data, in our case, a list whose length is not known a priori. This is mitigated by the second approach outlined in Section 6. We further evaluated the possibilities to improve FlowScope to be suited for this matter and showed possibilities for further research.

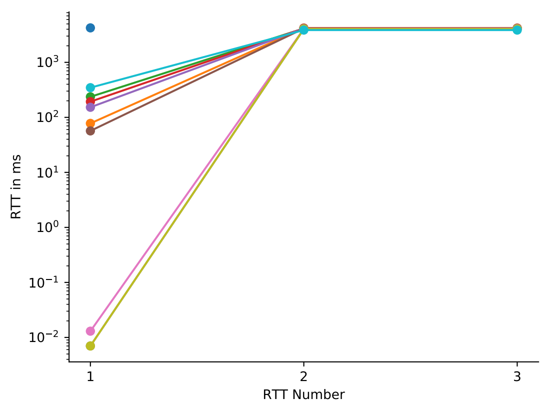

Figure 9: The Distinct RTT of the Ten Highest Average RTTs

10. REFERENCES

[1] P. Emmerich. emmericp/moongen: A fully scriptable high-speed packet generator built on dpdk and luajit. `https://github.com/emmericp/MoonGen`. Last visited 2018-06-14.

[2] P. Emmerich. libmoon: libmoon is a library for fast and flexible packet processing with dpdk and luajit. `https://github.com/libmoon/libmoon`. Last visited 2018-06-13.

[3] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. *Proceedings of the 2015 ACM Conference on Internet Measurement Conference - IMC '15*, pages 275–287, Oct. 2014.

[4] P. Emmerich, M. Pudelko, S. Gallenmüller, and G. Carle. FlowScope: Efficient packet capture and storage in 100 Gbit/s networks. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9, Stockholm, Sweden, June 2017. IEEE.

[5] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-Level Traffic Measurements from the Sprint IP Backbone. *IEEE Network*, 17(6):6–16, 2003.

[6] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, 1991.

[7] H. S. Martin, A. J. McGregor, and J. G. Cleary. Analysis of internet delay times. In *Proceedings of passive and active measurement workshop in Auckland, New Zealand*, pages 1–8, 2000.

[8] S. McCreary and K. Claffy. Trends in Wide Area IP Traffic Patterns: A View from Ames Internet

Exchange. *13th ITC Specialist Seminar on Measurement and Modeling of IP*, (May 1999):1–11, 2000.

[9] M. Mellia, A. Carpani, and R. L. Cigno. TStat: TCP STatistic and Analysis Tool. *Quality of Service in Multiservice IP Networks*, 2601:145–157, 2003.

[10] M. Pall. ffi.* api functions. `http://luajit.org/ext_ffi_api.html`. Last visited 2018-06-14.

[11] M. Pall. The luajit project. `http://luajit.org/`. Last visited 2018-06-13.

[12] J. Postel. Transmission control protocol. RFC 793, RFC Editor, Sept. 1981.

[13] M. Pudelko and P. Emmerich. FlowScope/exampleUserModule.lua at 7beb980. `https://github.com/pudelkoM/FlowScope/blob/7beb980e2cb64284666ba2d62dda5727c7bfd499/examples/exampleUserModule.lua`. Last visited 2018-06-14.

[14] M. Pudelko and P. Emmerich. Flowscope. `https://github.com/pudelkoM/FlowScope`, 2018. See commit `7beb980` for the version used for this paper.

[15] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ecn) to IP. RFC 3168, RFC Editor, Sept. 2001.

[16] S. Shakkottai, R. Srikant, N. Brownlee, A. Broido, and K. Claffy. The RTT Distribution of TCP Flows in the Internet and its Impact on TCP-based Flow Control. *Cooperative Association for Internet Data Analysis (CAIDA)*, pages 1 – 15, 2004.

[17] The Linux Foundation. DPDK: Data plane development kit. `https://dpdk.org/`. Last visited 2018-06-13.

[18] C. Wahl. Tcp rtt analysis with flowscope. `https://github.com/sn0cr/flowscope-tcp-rtt-analysis`, 2018. See commit d73a208 for the version used for this paper.

[19] A. Wingo, J. Muñoz, and L. Gorrie. Pflang specification. `https://github.com/Igalia/pflua/blob/3c0b47078a24a0306e557af6063bc918e5897adb/doc/pflang.md`. Last visited 2018-06-14.

[20] A. Wingo, J. Muñoz, and L. Gorrie. pflua: Packet filtering in lua. https://github.com/Igalia/pflua, 2018. Last visited 2018-06-13.

[21] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. *ACM SIGCOMM Computer Communication Review*, 32(4):309, Oct. 2002.