Inference of Traffic Differentation Policing Rate

Martynas Rimkevičius

School of Computer and Communication Sciences

Semester Project

June 2025

Responsible
Prof. Katerina Argyraki
EPFL / NAL

Supervisor
Pavlos Nikolopoulos
EPFL / NAL

Supervisor
Mahdi Hosseini
EPFL / NAL

1 Introduction

Traffic differentiation (TD) is when ISPs apply different traffic management techniques to incoming traffic, based on its source and/or content. ISPs may prioritize certain traffic by sending them through faster-dedicated routes, or they may throttle them using shapers and policers. ISPs may deploy TD practices with good intentions, for example, to prevent video traffic from congesting their network [1]; or they may do so to serve a business need [2].

Nonetheless, the user, who either supects that the traffic might be policed, or knows their traffic is being policed using Wehe application [3], might want to use the full capacity of the throttled rate. Using the maximum capacity of a known link rate is a problem that can be solved by configuring TCP receive window, adjusting TCP congestion control mechanism or asking the sender to send at a given rate. However, we are going focus on another part of this issue – inferring the bottleneck rate from the recorded network traffic flow.

The goal of this project is to investigate the accuracy and reliability of various policing rate estimation methods that are already created and used, and to create our own methods. We focus on both sender-side estimation and receiver-side estimation, as there can be an advantage of knowing the bottleneck rate for both. On both sides we show how different token bucket configurations affect the estimation accuracy by keeping the rate the same, and varying the burst length and queue size of the policer. In addition to that, we test two different network topologies: simple sender and receiver, and cross traffic topology with background traffic.

2 Background

2.1 Traffic throttling

Traffic throttling is when network operators use a rate limiter to limit the rate of traffic traversing their network to a pre-defined rate. It is implemented using a token bucket mechanism. The token bucket is defined by three parameters: the *rate*, which specifies how quickly the token bucket is refilled, the *burst length*, which sets the capacity of the bucket, and the *queue size*, which defines the limit before the queue checks for available tokens. When the token bucket is empty and a packet arrives, the rate limiter may drop it (produce loss) or put it in the queue (produce delay). Depending on the queue size, packets either experience more delay (if the queue is deep), or loss (if the queue is shallow). In this report, we investigate how the inference of the limitting rate works with both deep and shallow queues, as well as different burst lengths.

2.2 Existing work

There has been some work done that inspired this method exploration such as Flach *et al.* paper [4] where authors create traffic policing using an inferred rate, the paper by Li *et al.* [3], where they identify differentiation and then infer the throttling rate using statistical methods.

In the paper by Flach *et al.* [4], the algorithm is based on the preprocessed server-side packet capture, where all lost packets are identified. The method computes the data delivered between the first and the last lost packet. We explore the accuracy of this method in this report. However, the authors do not describe how the packets are marked as lost, which we solve with a simple algorithm that uses both sender and receiver packet captures.

Wehe application creators, Li *et al.*, [3] develop a different method. This method is based on running the same traffic over the same route multiple times with the assumption that if traffic is throttled once, the traffic will be throttled the same way again. Therefore, the average throughputs distribution of these tests would show the throttling rate instead of a random distribution. They use kernel density estimation (KDE) of throughput to find large probability densities that represent the throttling rate. An advantage of this method is that this estimation could show multiple local maxima, which means that KDE is able to estimate multiple throttling rates. The drawback of this method is that it requires many tests and throughput samples to produce the KDE. In contrast to this approach, we focus on single test policing rate inference, which means we cannot use probability density function for our estimation.

3 Setup

3.1 ns-3 network setup

We use ns-3 simulator [5] to create topologies and run experiments with sender, receiver and a token bucket filter. The simulator allows us to explore any network conditions and setups, as well as to get the ground truth of the sending rates or dropped packets.

We create two types of topologies, the simple topology and the X topology, where there is cross-traffic in addition to measurement traffic. The simple topology consists of the sender node (*S1*), the receiver node (*R1*), and the intermediate node where the token bucket filter (*TBF*) is installed as shown in Figure 1. The link rates between these nodes are the same: 200Mb/s. The X topology expands this simple topology by adding another sender (*S2*) and receiver (*R2*). However, to achieve actual cross-traffic, two more nodes are added of which one has a FIFO queue installed (*Queue X*). The other node (*Helper*) is used to make the traffic share one link and induce delay in measurement traffic when two flows share the same queue. X topology is shown in Figure 2.

As shown in Figure 2, network links are configured to investigate different traffic ratios between measurement traffic and background traffic without having a bottleneck before the *TBF* node. The link between *Queue X* and *Helper* nodes is configured to be limitting enough to have packets from both flows end up in the FIFO queue, but not policing where we would see loss at *Queue X*.

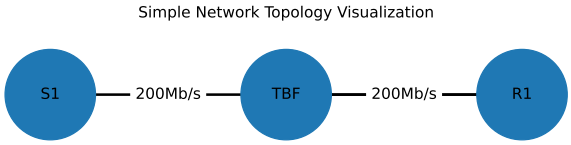

Figure 1: Simple topology

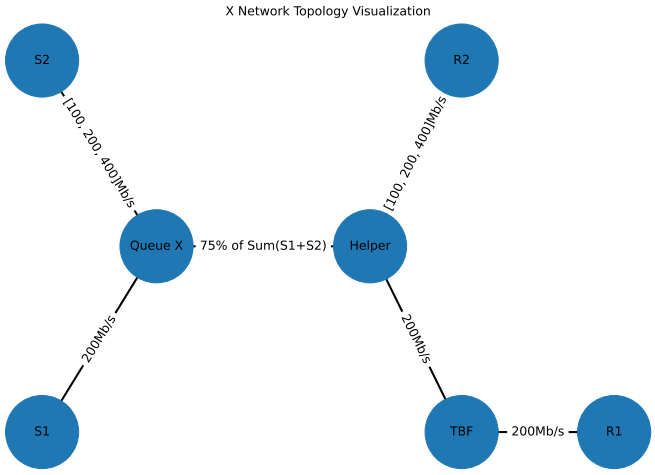

Figure 2: X topology

3.2 Traffic applications

We use two types of applications to send the data: a bulk-send application that sends constant packets and a modified bulk-send application that sends variable-size packets. These applications provide data for a TCP socket, which by default is configured to use CUBIC congestion control.

The bulk-send application sends as much traffic as possible trying to fill the bandwidth until the application is stopped. The application run time is set to 10 seconds. This type of application is a fitting choice as it does not allow the token bucket filter to generate and store tokens - they have to be used up to send packets. Important to note that at the simulation start, the token bucket is full. This type of applcation is commonly used in ns-3 simulations. We set the constant packet size to 1448B which equates to 1 Ethernet's maximum transmission unit size (1500B).

The variable packet size bulk-send application is an extension of a default ns-3 bulk-send application by adding a uniform random number generator for packet sizes. This generator gives a number between specified maximum and minimum values in bytes. In our case these bounds were 1 and 1448, given that we wanted to keep the packets up to maximum transmission unit.

3.3 Parameters

In our experiment, we use two variable parameters for simple topology, queue size and burst length of the token bucket filter (TBF), and three variable parameters for X topology, where we add a traffic ratio parameter. The goal is to do a parameter sweep of policer configurations while keeping the policer's out rate constant and round-trip time (RTT) the same as shown in Table 1. Note that, due to the additional links required for the X topology, the RTT increases from 0.02 to 0.03s.

The queue size is specified in the multipliers of the bandwidth delay product (BDP), where $BDP = out\ rate \times RTT$. Since RTT is different for the two topologies, the computed queue sizes are also different. In this report the figures only contain the estimation up to $5 \times BDP$ queue size as that is an expected size of the real-world routers due to space complexity.

The burst length is specified in the multipliers of the maximum transmission unit (MTU), which is 1500B. Since the packets of constant size application and variable size application are both limitted by MTU, we use that as a multiplier to specify how many tokens can be accumulated in the token bucket.

The traffic application is another variable used to evaluate the inference algorithms, as for some algorithms, different packets could make the calculation less accurate. We only explore both sides of this variable for simple topology, as X topology is a more complex scenario. Therefore, we only use variable-sized application to simulate measurement traffic in X topology. For background traffic we use constant-sized application as it does not have significance if it is variable, or not.

The traffic ratio defines a multiplier used to calculate background traffic rate from to measurement traffic rate. We change the send rate by changing background server's outgoing link rate and receiver's link rate, as shown in Figure 2.

Parameter	Values
Rate	2 Mb/s
RTT	0.02s (simple topology), 0.03s (X topology)
Queue Size	[0.1, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 3, 5, 10, 20, 25, 30, 35, 40, 50, 100]$\times BDP$
Burst Length	[0.5, 1, 5, 25, 100]$\times MTU$
Traffic application	[Constant-packet app, Variable-packet app]
Traffic ratio (only in X topology)	[1:2, 1:1, 2:1]

Table 1: Network setup parameters

4 Server-side rate estimation algorithms

Given that the goal of the algorithm which finds policing rate is to later adjust server's sending rate such that it maximizes this bottleneck, we wanted to investigate the methods to infer the rate from the server's side. First algorithm we explored was based on a paper by Flach *et al.*[4] with a small change, as we had to use both server's and client's packet capture to obtain our estimation instead of just server-side. Another method was to use TCP congestion window together with the observed round-trip time (RTT) and retransmission timeout (RTO) to compute the estimated rate.

4.1 Loss-based rate estimation

We explored an algorithm to estimate the policing rate created by Flach *et al.*[4]. This algorithm is based on sender's packet capture and assumes that this data has been preprocessed by marking packets as lost or not lost. We fulfill this assumption in the Section 4.2. Once the packets are marked, the algorithm takes the first lost packet, the last lost packet, and computes goodput between them – sum of all packet sizes that were marked as not lost over the time between the first and the last loss event.

Figure 3 shows this loss-based estimation algorithm values for different variables. Generally, this algorithm underestimates the actual rate, as it is supposed to be 2Mb/s and all values are close or under the 2Mb/s line. The constant-sized application in a simple topology is the simplest case, in which we can already see that for higher burst lengths, the estimation is off by 15%. It is worse for variable-sized application (which is used as the X topology measurement traffic as well), where up to queue size of $5BDP$, the rate is underestimated. Note, that for burst length smaller than one packet, the estimation is always equal to 0Mb/s as there are no observed packets delivered between the first and the last loss. For higher queue sizes, where loss might not occur, or occur only once, we would see the same effect, where estimated rate is 0Mb/s.

4.2 Loss detection algorithm

We developed an algorithm to mark sender's packets as lost or not lost using both server-side and client-side packet capture. This detection is shown as pseudocode in Algorithm 1.

The algorithm uses two pcap files, in which we can find a packet sequence number, source and destination ports, and packet timestamp. For each packet, from the sequence number,

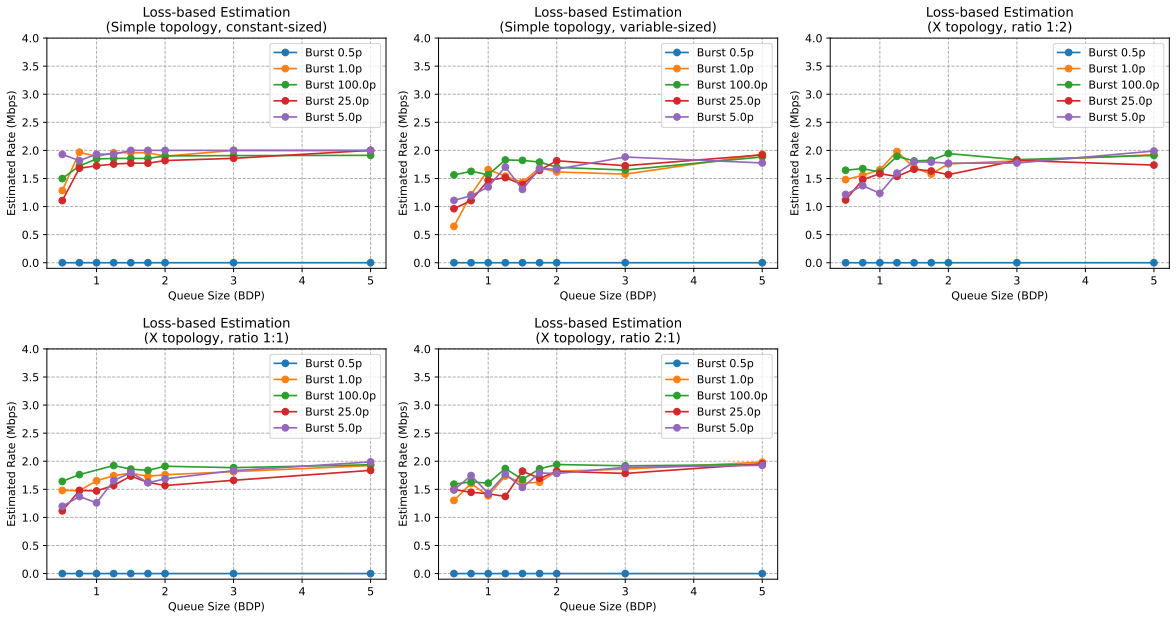

Figure 3: Estimated rate in different scenarios using loss-based estimation on server's side

source and destination ports, we compute a hash. This hash is then used to compare sender's transmitted and receiver's received packets.

By default, we know that if a sender's packet hash does not show up in any received packet hashes, it means it is lost. Next, we go through all received packets, and check whether the received packet's hash occurs more than once in the set of sent packets hashes. Intuition here is that if there are more than one sent packets with the same sequence number, all transmissions except the last one will have been dropped. Using this intuition we mark all packets with the same hash and timestamp (that is before the received packet timestamp) in the server set as lost, and set the latest packet by time as not lost.

This algorithm was developed in a simulation environment where we can trace which packets were dropped by the policer. This way we verified that we mark all packets correctly in the simple topology experiment. However, it is a limited scope algorithm as it does not account for multiple path sending where a packet could be delivered out of order and not discarded. It also requires packet captures at both the sender and the receiver, which is not always possible in real world. One way to extend this algorithm would be to use data hash instead of sequence number, as then, if retransmits are concatenated, the hash would still show up. Another way, would be to do sequence and acknowledgement number analysis only on sender's side. This method could identify which packets where acknowledged (mark as not lost) and for which packets sender did not receive acknowledgement immediately (mark as lost).

Algorithm 1 Mark server-sent packets as "lost" or "not lost"

Input:
  C ▷ set of client-received packets (c_pkts)
  S ▷ set of server-sent packets (s_pkts)
Output: Each packet in S has a boolean field `lost`

Initialization:
for each $p \in C$ **do**
  p.lost $\leftarrow \neg\big(p.\text{hash} \in \{\, s.\text{hash} : s \in S\}\big)$ ▷ If client never saw this hash, mark as lost
end for

Adjust for retransmissions:
for each $p \in C$ **do**
  $sent \leftarrow \{\, s \in S \mid s.\text{hash} = p.\text{hash}\}$ ▷ All server packets with the same hash
  if $|sent| > 1$ **then**
    $sent\_before \leftarrow \{\, s \in sent \mid s.\text{time} < p.\text{time}\}$ ▷ Those sent before the client received it
    for each $s_b \in sent\_before$ **do**
      s_b.lost $\leftarrow$ true ▷ Any earlier transmission attempt failed
    end for
    Let $s^* \leftarrow \arg\max_{s \in sent\_before}(\, s.\text{time}\,)$ ▷ Find the last send before arrival
    s^*.lost $\leftarrow$ false ▷ That last-before packet actually arrived
  end if
end for

return S

4.3 Rate estimation using congestion window

The next rate estimation algorithm we explored was using the TCP congestion window at the sender's side. By default, the congestion window is not included in the packet capture. Since the simulator has a way to track change in the congestion window of the TCP socket, we used this data together with traced RTT and RTO to compute estimated available bandwidth, which is the maximum rate at the policer. We compute the estimated rate at each congestion window change: $rate(t) = CWND(t)/(RTT(t_{RTT})$ or $RTO(t_{RTO}))$, where we choose between using RTT or RTO at timestamp t by choosing the closest t_{RTT} or t_{RTO} timestamp.

The rate estimation cannot be used from the full flow, as the burst length creates false impression for congestion window that the link is unrestricted. The long burst makes the congestion window grow rapidly, and thus overestimate the bandwidth. We only care about the time where congestion window change is stable and fluctuates between the same values. Therefore, we apply a filter where we discard the estimated rates of the first 4 seconds of the experiment. This can be seen in the Figure 4 where we keep the rate estimation values (on the right) that are after the congestion window stabilizes (on the left). Last, to estimate the rate over the whole (filtered) flow, we take the mean of the estimated rate. Intuitively, when the congestion window fluctuation stabilizes, the maximum value is the one before loss occurs as the limit is overestimated, and the minimum value is highest backoff where the limit is underestimated.

The congestion window estimation accuracy is shown in Figure 5. In general, the congestion window method overestimates the rate for high burst length, but for smaller burst lengths it is very close to the actual rate, given that the queue size is more or equal to $1BDP$. However, this method is sensitive to calibration, or empirical choice, from when to take the

measurement, meaning when it is determined that congestion window will fluctuate steadily. In a simulated scenario, where there is no changing traffic or packet reordering, it can be done algorithmically but due to lack of time an educated estimate was made from an eye-test. Note, that burst length of 0.5 packet is missing, because there was no change in congestion window after the empirically chosen 4 seconds mark, and therefore the estimation does not produce any values.

Another drawback of this method is that, along the packet capture, to get the RTT and RTO values, in the real world we need to record congestion window values, which needs to be done either by linux kernel module tcp_probe or the congestion window needs to be estimated from the packet capture itself.

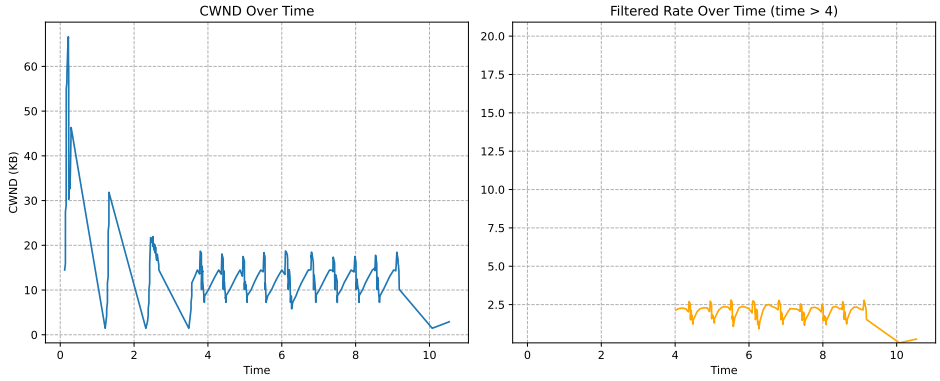

Figure 4: Congestion window over time and filtered computation of rate.

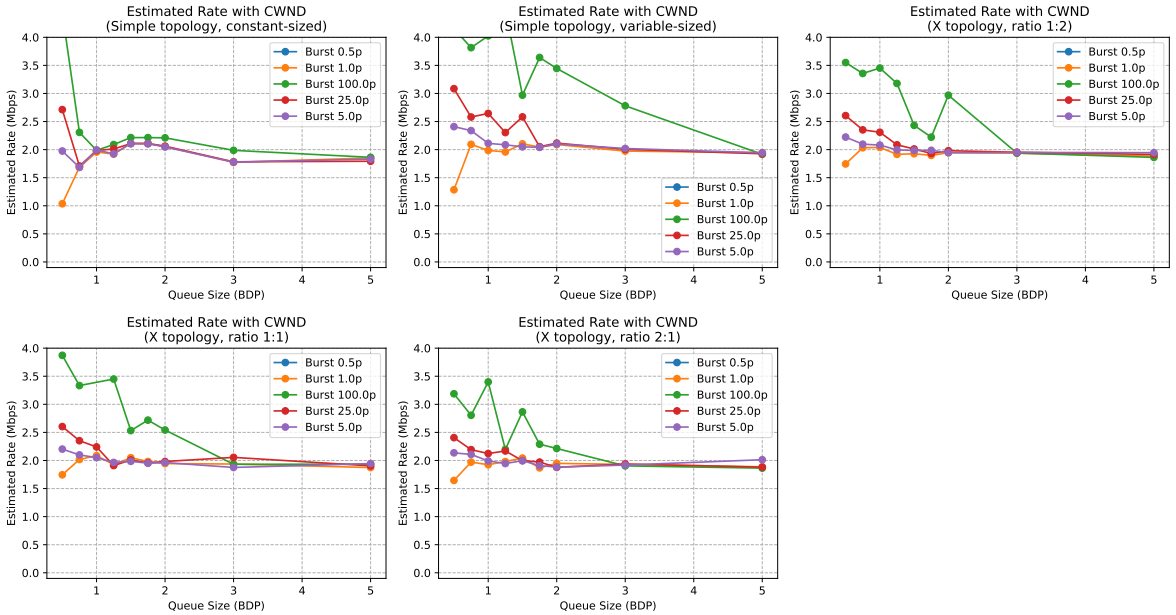

Figure 5: Estimated rate in different scenarios using congestion window on the server's side.

5 Client-side rate estimation algorithms

Another view of estimating policing rate can be on the receiver's side. Receiver having this estimation, given that the sender is the same and traffic follows the same path, could advertise the receive window size such that the sender would not go into congestion avoidance mode and would not experience loss. Therefore, we explored several methods to estimate the policing rate on the client's side. First, we investigated if simple goodput computation would be sufficiently correct to infer the policing rate. Next, we sampled the goodput using a selected sample time length. Last, we explored how the slope of the cumulative arrival over time curve could indicate the policing rate.

5.1 Rate estimation using goodput

Since we have all of the packets received by the receiver, we can try to infer the policing rate by just computing goodput. In this method we compute the goodput between the first delivered packet and the last delivered packet shown in the receiver's packet capture. However, this method is more likely to estimate the TCP throttled rate rather than the policing rate, as TCP does not utilize full link capacity due to timeouts and backoffs.

Client goodput calculation accuracy is shown in Figure 6. It can be seen that for queue sizes under $1BDP$, the accuracy is poor, especially in simple topology with variable-sized packets. In small queues, the drops occur more frequently, so when the packets do go through the policer, especially for higher burst lengths, they come in a burst. However, queue sizes smaller than $1BDP$ are not common, as usually the queue is configured to produce traffic for the bandwidth. Since here the timestamp of the first and the last packet impacts the estimation, it matters whether the first or the last packet were lost or delayed. In general, this method is most accurate for high burst length, as can be seen by the 100 packet lines in the Figure 6 being closest to 2Mb/s in most cases. In contrast, small burst length like 1 packet or 5 packets are not measured as accurately as the others.

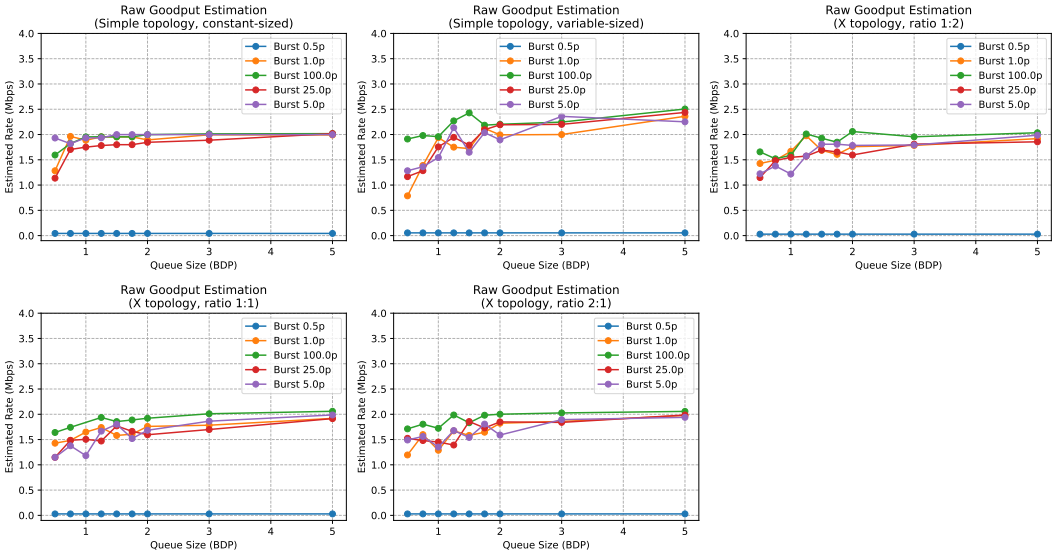

Figure 6: Estimated rate using client-side goodput computation.

5.2 Rate estimation using goodput sampling

Another algorithm, inspired by the Wehe [3] approach of finding the policing rate, is throughput sampling at the receiver's side. We use a slightly different method than computing probability density function from throughput recordings. Our approach uses only single packet trace.

We sample throughput by computing it at every timestep, for which we picked 1 RTT. We use initial RTT flag from packet capture as our RTT. We cannot update RTT as the receiver does not send any data and receives the only acknowledgement during TCP three-way handshake. Then, we compute sums of packet sizes in these RTT-long gaps, and find the median of these sums.

Figure 7 shows the accuracy of received throughput sampling. This is the most underestimating algorithm for small queue sizes (up to $1.5BDP$), as the sample time chunk is chosen to be 1 RTT. This chosen sample time is too short for small queue sizes where the drops occur more often than in the bigger ones, and thus it oversamples too many 0s and pushes the median down to lower value. Observe that in simple topology with constant-sized packets, it seems that instead of policing rate, the TCP throttling rate is calculated by this method.

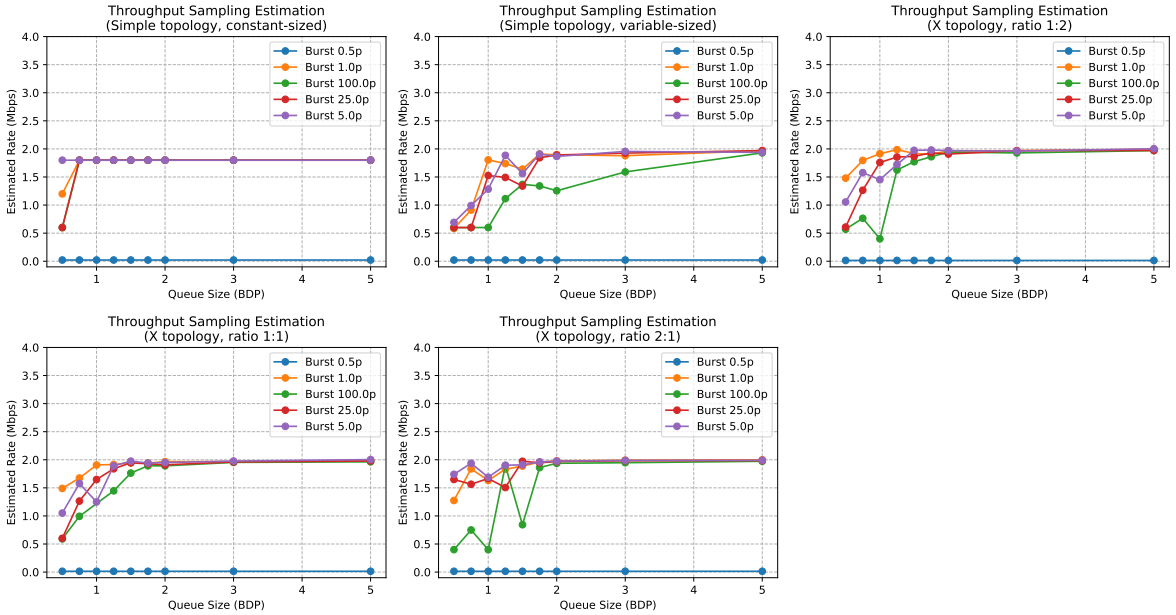

Figure 7: Estimated rate using client-side goodput sampling computation.

5.3 Rate estimation using cumulative byte arrival

The last client-side estimation that was investigated was using cumulative byte arrival at the receiver. At each received packet's timestamp, we compute the sum of all previous and current packets' lengths. This way we can get the first derivative which would show how many bytes arrive per time unit. We compute the first derivative by calculating the time gaps between the packets in receiver's pcap, and dividing the difference of data arrived at the packet times by

the time gaps. Cumulative data arrival and its first derivative are shown side by side in Figure 8. The flat part of the cumulative arrival curve means a timeout occurred due to loss, and the increase after timeout usually follows a very steep gradient. This repeating steep gradient can be seen in the Figure 9, where the traffic has very high burst length and small queue size. Thus, the traffic experiences many losses, and during those loss timeouts the bucket refills its tokens, which are then used up by letting packets just flow through.

To get actual estimated rate, we take the median of the derivative values, as median is not as affected by high outliers but only by the count of them. The outliers seen in the first derivative are packets arriving one after the other with only a transmission delay gap between them, meaning they were not impacted by the throttling and were together in a burst. However, if the median still overestimates the policing rate because of many bursts like in Figure 9, we can be more conservative and take the 20th percentile of the rates distribution. In our experiments, the 20th percentile seems to slightly underestimate the rate but only by around 2-3%. Due to lack of time we omit discussion of which percentile is best for future work.

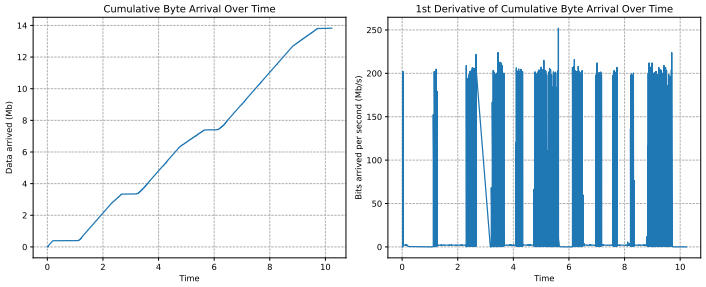

Figure 8: Cumulative arrival over time and first derivative of X topology with ratio 2:1, burst length of 5 packets and queue size of $1BDP$.

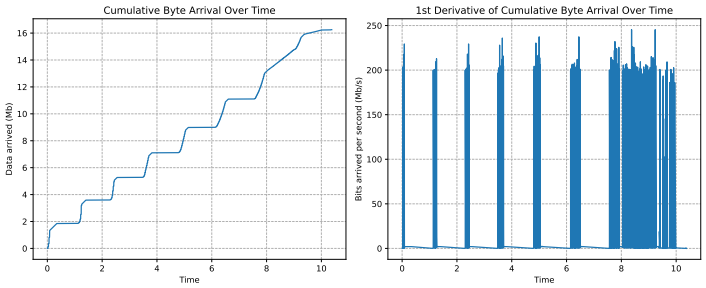

Figure 9: Bursty traffic cumulative arrival over time and first derivative of X topology with ratio 2:1, burst length of 100 packets and queue size of $1BDP/$

As shown in the Figure 10, the policing rate for queues sizes up to $2BDP$ is overestimated. However, this overestimation mostly is because of the big burst lengths, because of which we observe many tangent values equal to the link rate. Therefore, due to big count of high estimates, the median becomes higher than the policing rate. However, as already explained earlier, it is possible to find better estimates using lower percentiles. Nonetheless, this algorithm is more accurate than most other methods because once the impact of high

burst is minified by the higher queue size, we see that the estimation is very accurate – within 1% of the set policing rate.

In general, this method would work well for cellular ISPs as the throttled traffic is expected to be more confined to the limits, which would result in smaller burst lengths. High burst lengths, as seen in Figure 9, mean that traffic can flow unrestricted for hundreds of milliseconds which defeats the purpose of the policer.

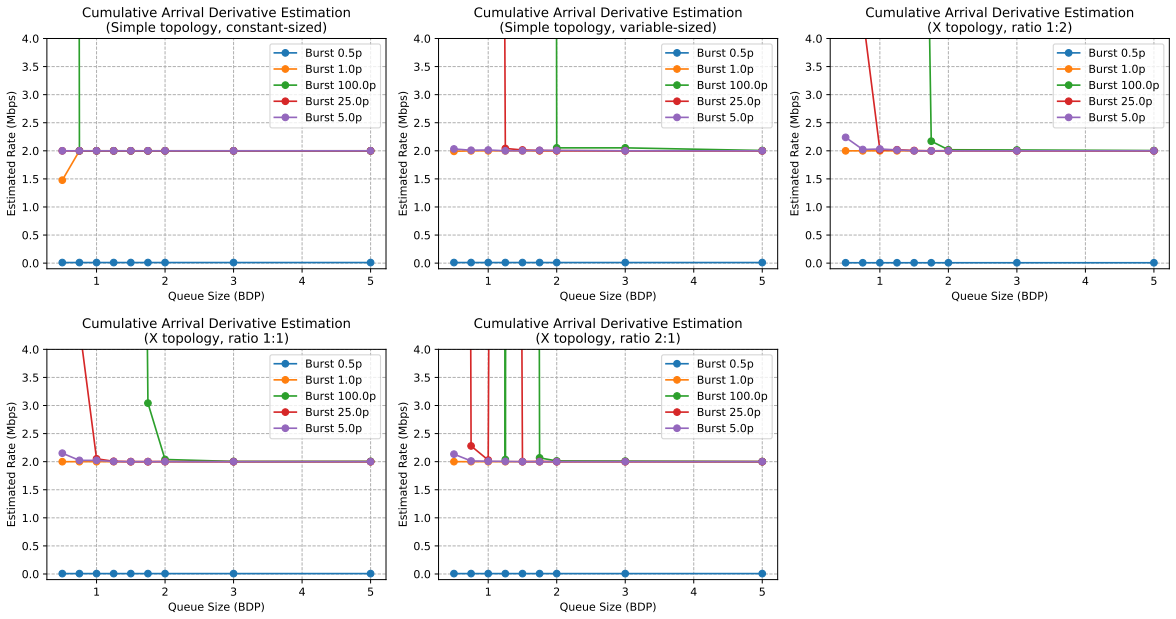

Figure 10: Estimated rate using cumulative byte arrival.

6 Future Work

One part of traffic differentiation techniques that was not explored due to lack of time is delayed throttling. This is a technique which applies traffic policing to the network flow only after a certain amount of time. In the paper of Wehe application [3], authors discuss their method to detect a point of change in the throughput over time curve together with KDE method to find the policing rate. However, it would be useful to put these methods, and specifically cumulative arrival derivative method, to the test and see how accurate they would be with delayed throttling.

Another part that was cut for time is exploring how different congestion control algorithms effect estimation accuracy. In this project, only TCP CUBIC was used as it is the default congestion control in the ns-3 simulator.

Given that cumulative arrival derivative is an accurate estimation for most network conditions, it would be interesting to have an improved estimator. This estimator could use more sophisticated statistical analysis which would be able to choose which percentile to use in high burst scenario, or traffic with a lot of timeouts (as shown in Figure 9).

Last, in the future it would be interesting to put together an integrated system that uses a throttling rate inference method together with the adjusted TCP sending rate. Then the performance of this system should be compared to current TCP congestion control algorithms in use.

7 Conclusion

After analysing five different policing rate estimation methods, we have found that client-side estimation method cumulative arrival differentiation is the most accurate. However, if the goal of the policing rate inference is not to overestimate, then the other methods, such as loss-based server-side estimation, are better.

A general trend observed over all the inference methods is that for small queue sizes under $2BDP$, they do not perform as well as for bigger queue sizes. This means, a higher count of loss events makes the estimations less accurate. This can be explained by the token bucket refill during the timeout after sender experiences loss, thus first packets after the timeout end up in a burst. We account for this burst in loss-based estimation by taking goodput over long period of time between the first and the last loss, in congestion window method by taking an average of the estimated rates. The cumulative arrival method can be affected by high count of bursts, but as it is the client-side protocol and it can be seen that the traffic is bursty, the estimation can be changed to use lower percentiles of the gradient distribution.

References

[1] "At&t video management." (2024), [Online]. Available: https://www.att.com/support/article/wireless/KM1169198/.

[2] "At&t business fiber jumps in the fast lane with gigabit speeds." (May 2016), [Online]. Available: https://www.prnewswire.com/news-releases/att-business-fiber-jumps-in-the-fast-lane-with-gigabit-speeds-300266126.html.

[3] F. Li, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove, "A large-scale analysis of deployed traffic differentiation practices," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, Beijing, China: Association for Computing Machinery, 2019, pp. 130–144, ISBN: 9781450359566. DOI: 10.1145/3341302.3342092. [Online]. Available: https://doi.org/10.1145/3341302.3342092.

[4] T. Flach, P. Papageorge, A. Terzis, *et al.*, "An Internet-Wide Analysis of Traffic Policing," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 468–482, ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934873. [Online]. Available: https://dl.acm.org/doi/10.1145/2934872.2934873 (visited on Mar. 3, 2025).

[5] nsnam, *Ns-3*, en. [Online]. Available: https://www.nsnam.org/documentation/ (visited on Jun. 2, 2025).