Multimedia Communications Project
Queue Management

Paul Stiegele
TU Darmstadt
paul.stiegele@stud.tu-darmstadt.de

Dat Tran
TU Darmstadt
dat.tran@stud.tu-darmstadt.de

***Abstract*—”The aim of the lab was to find among a selection of popular and potentially promising programming languages the one that enables the highest network throughput with data link layer raw sockets when forwarding packets between different interfaces. Based on this evaluation, an emulator was developed in the programming language Go, which allows to investigate the efficiency of active queue management algorithms. These AQM algorithms promise an improved buffer behavior by dropping individual packets before the buffer gets full, which stores the packets if the packets cannot be sent as quickly as they are received. The CoDel implementation used did not ensure a higher throughput, but the latency decreased significantly. ”**

I. Introduction and Motivation

The bandwidth requirements for Internet connections are steadily increasing - for example by the use of online video stream platforms with 4k streaming or the use of video telephony. A high data throughput is needed to meet these requirements. However, if this reaches its capacity limit, a sophisticated approach is required to handle this limitation as good as possible.
The typical reason for bottlenecks are the links between routers which have less bandwidth than required. Assume that there are three hosts as seen in Fig. 3. Host 1 want to send data to host 3 with a very high transmission rate. The link between host 2 and host 3 (egress link) has a very limited bandwidth. Host 2 only takes care of forwarding the packets. The link between host 1 and host 2 has a very high bandwidth.
Unfortunately, in this example, host 2 cannot forward the packets to host 3 immediately, since the egress link with the limited bandwidth has reached its maximum capacity. As a result, the incoming packets will pile up at host 2. Therefore there must be a buffer (a queue) at host 2 where the waiting packets can be temporarily queued. Typically, a router holds a queue for each interface in which newly arriving packets are inserted. Of course this buffer has only limit resources. But if more packets gets received than can be processed, this buffer will be filled up quickly. If the maximum number of bytes or packets (depending on the queue type) is reached, in the simple case the newly arriving packet gets dropped. This process is called *drop tail* and has some disadvantages.
One of them is, that the TCP sending rate will not decrease until the buffer is full and every incoming packet at the tail of the queue gets dropped. And until the sender detects this, it takes a while in which potentially many more packets gets dropped. Therefore, TCP will always fill up the queue to a certain value which is close to the maximum queue size. In other words, the queue has been bloated and it’ll get bigger over time and will never be empty as long as the load does not decrease. This is unwanted in many cases and is called bufferbloat [9].
In addition, there is still another problem with the drop tail method, which is the TCP global synchronization [13]. Qiu et al. has pointed out in their paper about this problem and they suggested to use *RED* (random early detection) to break this globalization. TCP global synchronization occurs when the buffer is full and packets from multiple TCP connections gets dropped at the same time. This leads to a synchronized decrease in the data throughput of each TCP connection, which leads a moment later to a periodic increase and a repeated decrease in the data throughput. In Fig. 1 it can be seen, that this leads to a non-ideal utilization of the possible data throughput.

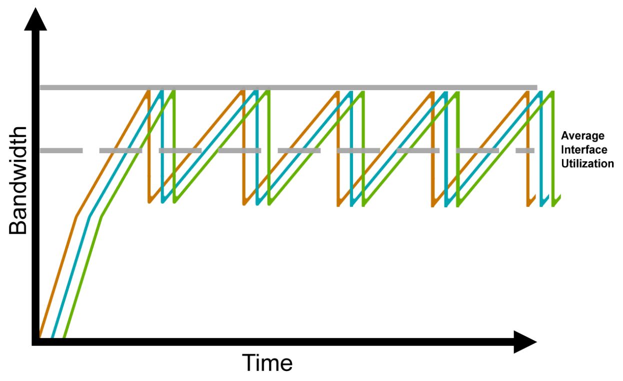


Fig. 1. *Global Synchronization, these three TCP flows suffer from packet lost at the same time and gets synchronized*

Bufferbloat occurs when the transmission of packets is delayed due to very large and already filled buffers. If the utilization of the link is constantly high, the buffer will stay full and every packet needs to walk through the buffer which costs some time. This can be a problem especially for time-critical applications such as VOIP telephony or gaming.
An alternative to the classic drop tail procedure is active queue management(AQM) [1]. In principle, AQM will drop packets even if the queue is not yet full. Therefore it can combat against the bufferbloat and also against TCP global synchronization. Consequently the latency and throughput

will be improved. There are many different AQM strategies, such as *controlled delay* (CoDel) [14], *random early detection* (RED) [4] or *proportional integral controller enhanced* (PIE) [6]. The AQM is realized by the network scheduler of the operating system.

The network scheduler has the responsibility to receive the packets, put them temporarily into a buffer and send them in a specific order depending on which queue algorithm is being used. Some queuing disciplines are already available in modern operating systems. For example, the linux kernel network scheduler has implemented the *fair queue codel* (fq codel) as its default queuing algorithm.

As we have pointed out, there are a lot of queuing algorithms. So there is a desire to compare the performance of these. To create the conditions for such a comparison, in this work, an active queue management emulator was built. The goal is to be able to quickly evaluate the performance of many active queuing disciplines and compare the efficiency as well as point out the disadvantages between them in a mininet emulator testbed [10].

In general, our work consists of two phases:

- Phase 1: Programming language evaluation
- Phase 2: Implementation of the AQM emulator in a suitable language

In phase 1 of the project, we have to determine the most efficient programming language for the purpose of forwarding packets on a data link layer basis. This step will be discussed in detail in the section III, User Space Packet Forwarding. After discovering the most efficient programming language for our scenario, we continued with phase 2 in which we designed and programmed an active queue level emulator which will run in user space. By this approach, the rapid development of various AQM algorithms is possible. For our purpose we have implemented the classic drop tail method and a CoDel implementation and compared them. Section IV, Queue Emulation Framework will cover this phase 2. Afterwards, the conclusion will be discussed in section V.

II. STATE OF THE ART / RELATED WORK

One important goal of the CoDel [11] algorithm is to combat against bufferbloat [7]. If the transmission rate is lower than the arrival rate, then the queue will be filled up quickly with network packets. As a result, the packets will suffer from a long delay due to the long queue. CoDel uses the delay a packet gets while going through the queue as a metric. It makes sure that packets with a delay of less than 5ms will be successfully transmitted. The procedure of the algorithm is shown in Fig. 2 [5]. The two variables *target* and *interval* usually will be set to 5ms and 100ms initially. This means, that within the interval (initially the 100ms) all packets will be dequeued and send even if the delay is more than the targets 5ms. They're never dropped within the interval. Only at the end of an interval, the last packet could be dropped if one of the packets in the interval had a longer queue-delay time than the targets 5ms. To adjust the algorithm, this interval can not

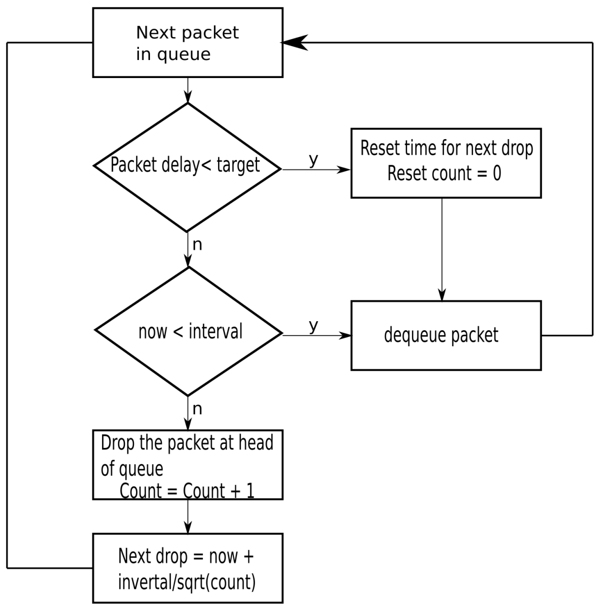


Fig. 2. *CoDel Queueing Discipline*

be constant; it will be modified during the execution. If packets have a delay greater than 5ms for a long period, the interval will gets shorter, which means the drop rate will be increased to reduce the sending rates of the senders and archive that the queue delays fall below 5ms.

Every time we want to dequeue a packet, there are two conditions that need to be checked. If one of them is true, the packet will be dequeued successfully. In contrast, if none of them is true, which means that the end of the interval is reached and the delay of one of the packets inside the interval is beyond 5ms, then we drop the current packet and increase the *count* by 1. This *count* variable will count the number of dropped packets and it's proportional to the drop rate as shown in the formula in Fig. 2.

In other words, when we see more dropped packets, it is a signal of a busy queue. As a result, the drop rate will be increased accordingly. If a packet gets dropped by the CoDel algorithm, the interval will be shortened by the factor $\sqrt{count}$. If the delay of one packet ever falls below 5ms during the interval, the interval will be reset to the default value of 100ms.

There is an implementation of the CoDel queuing discipline within the domain-specific P4 language [2] on programable data plane. Another advantage of their work is the flexibility. The CoDel algorithm was implemented on the P4 reference model bmv2 as open-source and it can therefore be executed on any linux based system. Their result clearly showed that the CoDel algorithm can be used to remove the bufferbloat issue. Instead of having a periodic throughput going up and down like in Fig. 1, CoDel can keep the throughput constant in line with the limited output link.

III. User Space Packet Forwarding

We use Mininet to create a network topology as quick as possible. Mininet [3] is a network emulation orchestration system with which you can create any number of hosts, switches, links and controllers - everything in software. And for the most part, they behave like the real hardware components. The process-based virtualization of Linux in combination with network namespaces is used for this.
In Mininet various topologies can be created. To implement our AQM Emulator, we have created a topology with 3 hosts as shown in Fig. 3. Host 1 and host 3 behave as clients, whereas host 2 takes on the role of buffering and forwarding packages by using our CoDel queue discipline implementation. As mentioned earlier, our work consists of two phases. In phase 1, we implemented the User Space Packet Forwarding with various programming languages, then do the performance comparison between them and choose the most suitable one, which will be used in phase 2 to implement the AQM Emulator Framework including the CoDel queuing discipline.

A. Design of the topology for programming language evaluation

A basic communication must have at least two hosts, which we will call host 1 and host 3 and the network between them. Since we want to keep the network topology as simple as possible, the network between these two hosts is simply a single node which is the host 2 in the middle as we can see in the Fig. 3. According to this simple topology, host 1 and host 3 can communicate with each other via host 2. Therefore host 2 is a forwarding node, which will simply sniff the traffic at one interface and send it to the other interface. We also want to point out that there is no queueing disciplines implementation at host 2 in this phase 1 yet. Instead, the packet will be forwarded as fast as possible, since we want to evaluate which language is the fastest one.

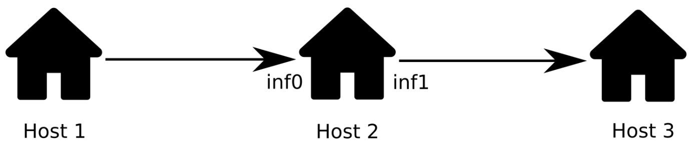


Fig. 3. *Forwarding topology*

B. Linux kernel forwarding

Our first approach after we build up the topology was to have a comparison for our own implementations. So we used the basic linux kernel forwarding.
At first, host 1 and host 3 is sad to use host 2 as the forwarding node. This can be configured with the command *sysctl net.ipv4.ip_forward = 1* at host 2 and *ip route add default via host2_ip dev [h1 interface / h3 interface]* at host 2 and host 3. As sad, with this configuration, the packets from host 1 will be forwarded to host 3 and vice versa by the linux kernel of host 2 without writing any program. Since there is no packet processing involved in this process, the throughput is very high as shown in Fig. 9. Because we cannot easily modify the behaviour of the linux kernel forwarding, we came up with the idea to emulate this forwarding functionality without using the linux kernel. This forwarding functionality is disabled with the command *sudo sysctl net.ipv4.ip_forward = 0*. Now the packets coming from host 1 will not be automatically forwarded to host 3. To be able to forward a packet, we need a simple forwarding program running at host 2, which is called the User space Packet Forwarding. Therefore, the same forwarding task will be written in different languages and executed at host 2.

C. Implementation of the User Space Packet Forwarding

As we have discussed earlier, a forwarding program must be executed at host 2 to serve the communication between host 1 and host 3. All of the implementation of each language have the same structure. This simple forwarding is structured as follow:

```
Thread 1
while(1)
{
raw_socket1.bind(h2-eth0)
raw_socket1.sniff()
if(raw_socket1.packet_arrive)
  send_to(h2-eth1)
}

Thread 2
while(1)
{
raw_socket2.bind(h2-eth1)
raw_socket2.sniff()
if(raw_socket2.packet_arrive)
  send_to(h2-eth0)
}
```

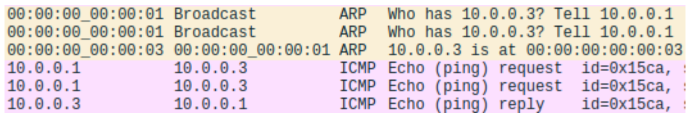
Fig. 4. *pseudocode for User Space Packet Forwarding*

In Fig. 3, we can see that host 2 has two interfaces. Therefore, we bind two sockets to these two interfaces to be able to sniff the whole traffic. Unlike the usual socket on a higher layer, which strips off the header field and sends only data to the application, we used raw sockets on the Data Link

Layer. As the name suggests, raw socket doesn't strip off the headers. Therefore, the application can receive both data and header fields such as MAC address and IP address. Indeed, the application layer can manipulate these header fields such as modifying the destination address and so on. This is the advantage of using raw socket. Basically we have two threads running all the time in an endless loop at host 2 which is shown as pseudocode in Fig. 4.

Each socket will be assigned to a dedicated thread. Each thread can therefore continuously sniff the traffic at the interface that it's been attached to. Every time a packet is detected at any interface, we send them directly to the other interface. So every packet arrives at interface 1 will be forwarded to the interface 2 and vice versa. Now the forwarding program is running at host 2 and host 1 and host 3 are able to talk to each other.

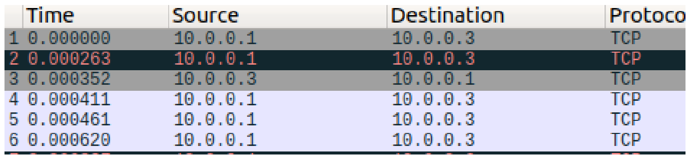
D. *Set up the evaluation enviroment*



```
00:00:00_00:00:01 Broadcast         ARP  Who has 10.0.0.3? Tell 10.0.0.1
00:00:00_00:00:01 Broadcast         ARP  Who has 10.0.0.3? Tell 10.0.0.1
00:00:00_00:00:03 00:00:00_00:00:01 ARP  10.0.0.3 is at 00:00:00:00:00:03
10.0.0.1          10.0.0.3          ICMP Echo (ping) request  id=0x15ca,
10.0.0.1          10.0.0.3          ICMP Echo (ping) request  id=0x15ca,
10.0.0.3          10.0.0.1          ICMP Echo (ping) reply    id=0x15ca,
```

Fig. 5. *Wireshark dump: ARP packets are sent before ICMP packets*

Now we can test our forwarding by simply ping from host 1 to host 3 or vice versa. The needed round-trip-time was captured and is presented in Fig. 8. Similarly, the throughput was also measured by the tool *iperf3*, which is widely used and available in various operating systems. For debugging purposes, we sniffed and captured every packet with *tcpdump* [8]. Then these captured packets can be opened with *Wireshark* [12] for further analysis. For the ping, we observed that the RTT is very high right at the beginning. This could be explained by the fact that ARP packets must be sent before the ICMP packets to resolve the layer 2 address (MAC address) of the destination as we can see in Fig. 5. Since these pings are not the decisive factor for our evaluation, their results were only considered secondary for our evaluation. In particular, there were 100 000 pings that were sent from host 1 to host 3 within a 0,01s interval, in which the first 50 pings out of 100 000 pings will not be included in the evaluation to make the evaluation more precisely because of the slow start. We also captured the packet traffic while the iperf3 tool was being executed. As shown in Fig. 6, TCP packets will be sent back and forth between host 1 and host 3 during the the throughput testing process. Since all tests depends



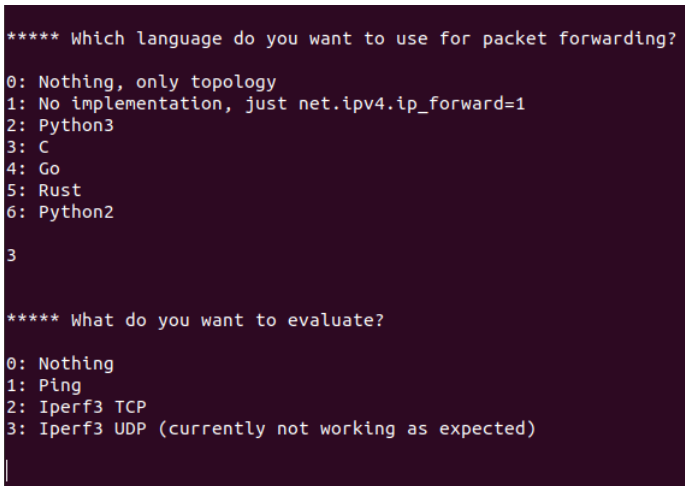
```
  Time       Source     Destination  Protoco
1 0.000000   10.0.0.1   10.0.0.3     TCP
2 0.000263   10.0.0.1   10.0.0.3     TCP
3 0.000352   10.0.0.3   10.0.0.1     TCP
4 0.000411   10.0.0.1   10.0.0.3     TCP
5 0.000461   10.0.0.1   10.0.0.3     TCP
6 0.000620   10.0.0.1   10.0.0.3     TCP
```

Fig. 6. *TCP packets are sent back and forth between host 1 and host 3 when iperf3 is used*

heavily on the available resources, running the evaluation on different computers will result in different numbers and it's not comparable. To get rid of this problem, all tests were carried out on the same system. For the same reason, the absolute results of the test are only of limited significance; the relative ratio is much more important, both when evaluating the fastest programming language for raw sockets and when evaluating the best active queue management algorithm.
For our evaluations a virtual machine was used via Oracle's Virtualbox on which an Ubuntu 18.04. was installed. The system had access to 4GB RAM and 4 CPU cores, each with a base clock of 3.6 GHz (Turbo clock: 4.2 GHz, Ryzen 5 3600, CPU limit: 100 %).

E. *Automate the evaluation process*

As we've seen in the last section, the environment must be set up for performance evaluation. This process takes a lot of time and effort. For example, we have to set up the same network topology for every testing by manually typing command lines even though there are only two different topologies (one with ip forwarding and the other without ip forwarding). That's why a python script was written to automate this process as shown in Fig. 7. Instead of writing every single command line, we only have to run this python script. This script allows the user to choose the network topology and choose the language that needs to be evaluated. There are 5 different languages that were evaluated including Python 3, Python 2, C, Go and Rust.



```
***** Which language do you want to use for packet forwarding?

0: Nothing, only topology
1: No implementation, just net.ipv4.ip_forward=1
2: Python3
3: C
4: Go
5: Rust
6: Python2

3


***** What do you want to evaluate?

0: Nothing
1: Ping
2: Iperf3 TCP
3: Iperf3 UDP (currently not working as expected)
```

Fig. 7. *Python Script to automate the evaluation*

F. *Comparing the performance and choosing the most appropriate language*

Fig. 8 shows how long a packet takes from the sender to the receiver and back again. A shorter round-trip-time promises a better connection, as this means that we have low latency between the sender and the receiver. By far the worst RTT occurred in the C implementation. Here the ping duration was

very high with an average of 13.5ms. The other programming languages differed only slightly: The values for Python 3, Python 2, Go and Rust were between 0.09ms and 0.12ms. Only the ip_forward alternative could beat this with 0.02ms.
In addition to latency, the bandwidth that is actually available is of course the most important factor in choosing the fastest programming language for raw sockets. As can be seen in Fig. 9, ip_forward achieves the highest performance with 5.8 Gbit/s. Apart from this, Go stands out with 3.4 Gbit/s. The languages C, Python 2, Python 3 and especially Rust cannot keep up with this.
Due to the good performance of Go, the increased popularity in the community in the last few years, the now quite high distribution in the network and cloud computing area and because the language is well suited for rapid programming, we ultimately decided to use Go in phase 2 of the project to implement our user space queue level emulator.

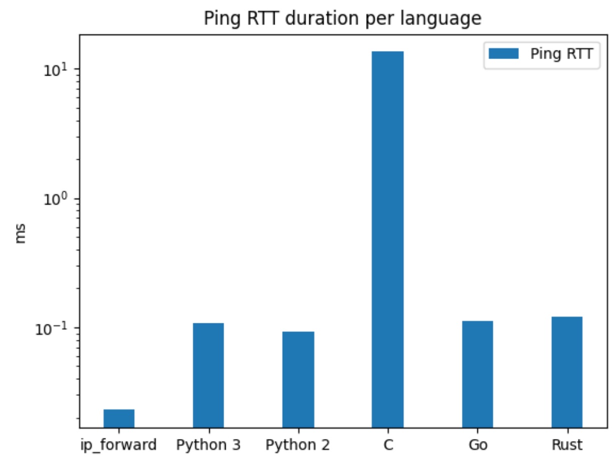


Fig. 8. *Ping round trip time per language*

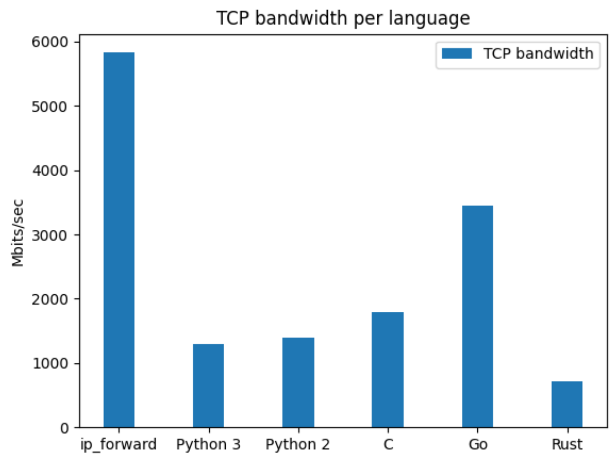


Fig. 9. *TCP throughput per language*

IV. A Queue Emulation Framework

A. *Implementation of the active queue management emulator*

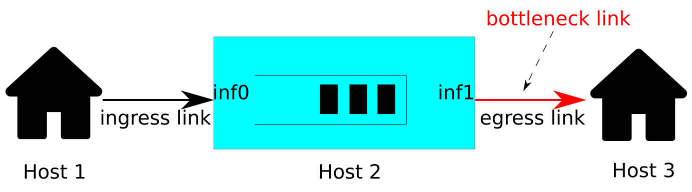


Fig. 10. *Host 2 buffers and forwards packet with interface 0 on the left side and interface 1 on the right side*

In our second step we implemented the AQM Emulator with the programming language Go. Firstly, our network topology has been changed a little bit as shown in Fig. 10. In opposite to the network topology in phase 1 where there was no queue at host 2, we have now a queue implemented at host 2. The reason is because of the bottleneck egress link (marked in red), which has a very low bandwidth. If the sending rate of host 1 is higher than the maximum rate of egress link, the egress link will become quickly occupied. Host 2 can then not forward any packets to the interface 1 anymore. As a result, host 2 has to drop some packets if there is no queue implementation at host 2 . Therefore, there must be a buffer in the middle to prevent such a scenario from happening. Therefore, we came up with the simplified topology which consists of two hosts as clients and another host in the middle for buffering and forwarding purposes as shown in Fig. 10. Host 1 can only communicate with host 3 via host 2 and vice versa, it's very similar as in phase 1. We choose to implement CoDel as a queuing discipline. However, our framework can also be applied for any other AQM as well. To be able to limit the egress link artificially, a rate limiter was implemented. The *token bucket* method was used as the procedure. The token bucket works as shown in Fig. 11. If the sender wants to transmit data,

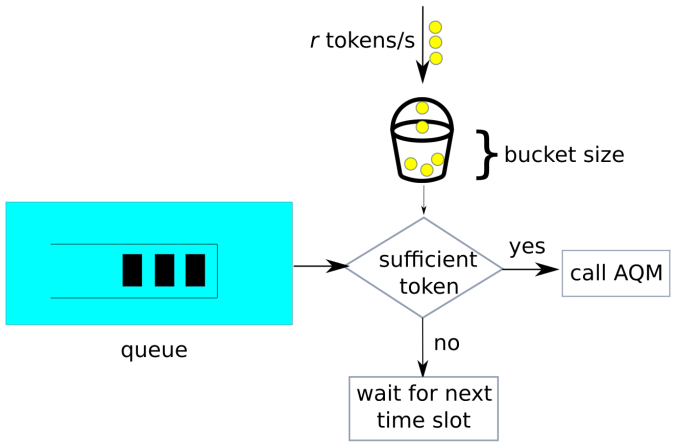
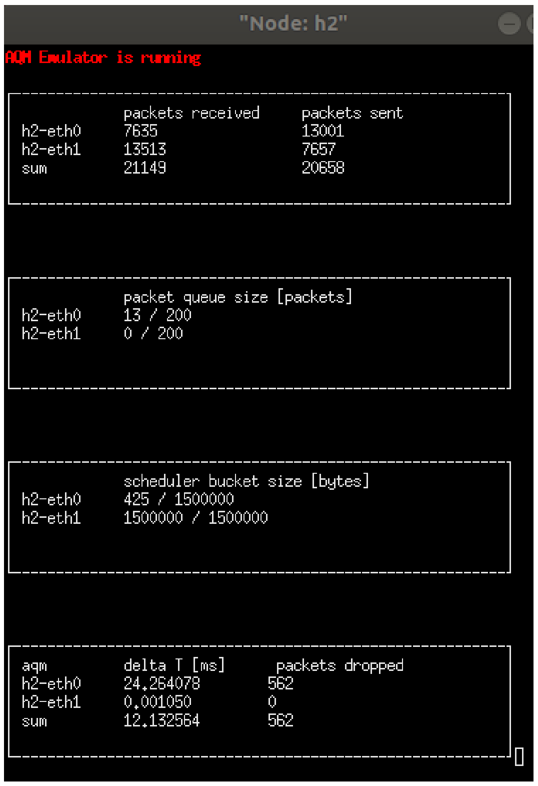


Fig. 11. *Token Bucket traffic shaping principle*

he has to have a sufficient amount of tokens. But there are only a limit number of tokens which are generated for each

time slot. If the sends-function wants to send data but there are no tokens left, then it has to wait until enough tokens are generated. Because of that we are able to reduce the sending rate.
There is one more thing that needs to take into account. As the name suggests, we have here a bucket which has a limit capacity. Whenever tokens are generated, they will be put into this bucket. If there is no data to be sent, the tokens will quickly fill up the bucket. If the bucket has reached its capacity, then the generated tokens will not be put into the bucket anymore. In summary, the size of the bucket and the token's generated rate will shape the sending rate. Obviously, these two parameters are adjustable and can therefore be adapted to fit our needs.



```
"Node: h2"
AQM Emulator is running

          packets received     packets sent
h2-eth0   7635                 13001
h2-eth1   13513                7657
sum       21149                20658

          packet queue size [packets]
h2-eth0   13 / 200
h2-eth1   0 / 200

          scheduler bucket size [bytes]
h2-eth0   425 / 1500000
h2-eth1   1500000 / 1500000

aqm       delta T [ms]      packets dropped
h2-eth0   24.264078         562
h2-eth1   0.001050          0
sum       12.132564         562
```

Fig. 12. *Dashboard of the AQM Emulator while running throughput test*

As can be seen in Fig. 12, it is possible to monitor the most important parameters during the test run via the built-in dashboard. The number of packets received and sent is displayed, as well as the current queue load, how many tokens are currently available in the token bucket, how much time the last packet has spent in the queue and how many packets have already been discarded due to the AQM CoDel implementation.

B. Structure of the code base

The general steps of our Go code base is shown in the Fig. 13. To be able to realize the AQM Emulator, there are five main classes in our implementation:

- Sender class

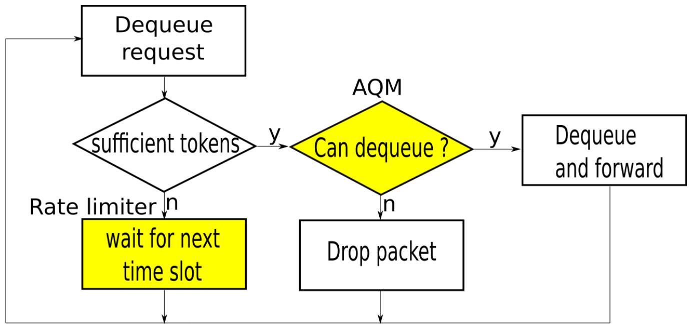


Fig. 13. *General steps of the code base*

- Receiver class
- AQM class
- Scheduler class
- Queue class

Receiver and Sender class are responsible to sniff the traffic with raw sockets, put it in the queues and forward the packets based on the AQM's decision. The Queue class is where the queue buffer is located and it also should capture the timestamps of the packets whenever a packet is enqueued or dequeued. The AQM class is where the CoDel queuing discipline is implemented. The scheduler class is the place where we implemented the token bucket. As we can see in the Fig. 13, this scheduler will be executed in a infinite loop. In this loop, the dequeue request will be continuously generated. Every time a dequeue request arrives, we first check the rate limiter if there are sufficient tokens inside the bucket. If there is enough amount of tokens available inside the bucket, it must call the AQM function for further decision. As we showed in the section "State of the Art / Related Work", there are two possible actions after calling AQM. The first one is to dequeue the packet out of the queue. And the second possibility is to drop this packet if it violates two conditions that haven been shown in Fig. 2.

C. Result

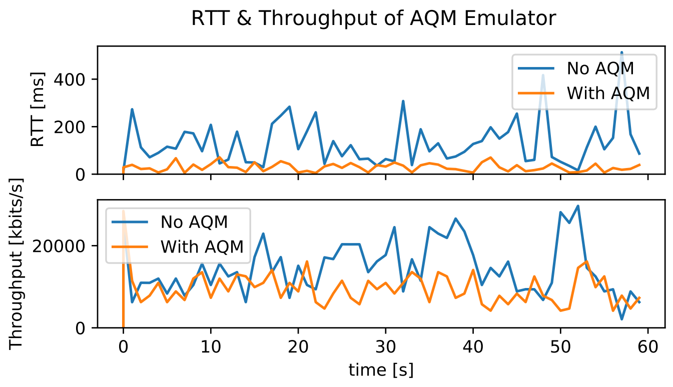


Fig. 14. *RTT & Throughput of AQM Emulator with and without AQM*

In the ideal state, i.e. without the rate limiter intervening or the queue becoming full, the implementation enables a maximum bandwidth between 400-500 Mbps.

These values are significantly lower compared to the value of our Go implementation from phase 1 with 3.4 Gbps (see Fig. 9). This can be explained by the fact that phase 1 was much less complex. If a packet was received, it was sent directly to the other interface without detours. This avoids costly tasks such as context switches, multiple copying of variables, calculations and suchlike.

In order to measure the performance gain by using CoDel, a bandwidth test was done with an active AQM implementation and one without. Care was taken to ensure that despite deactivating the AQM, its calculations continued to be carried out, but were not taken into account. This should avoid a possible deviation due to a lower required processor load in order not to influence the result.

The configuration of the AQM emulator:

- Maximum queue size: 200 packages
- Token Bucket Generation Rate: 0.01 ($\sim$ 10 Mbps)
- Maximum token bucket size: 1.5 * 10 ^{6} bytes (corresponds to $\sim$ 1000 iperf packets)
- CoDel inital interval: 100ms
- CoDel target: 5ms

The bandwidth was limited to approximately 10 Mbps by the Token bucket Scheduler. As can be seen in Fig. 14, the throughput with and without AQM is relatively comparable, but also very fluctuating. After a 60 second test, an average bandwidth of 10.8 Mbps when running with AQM and an average bandwidth of 10.6 Mbps when running without AQM were determined. Since the results are not consistent, a higher throughput for the variant with AQM cannot generally be assumed here.

However, the lower latency when using CoDel as an AQM algorithm is clear: As can be seen in Fig. 14, the RTT without AQM is significantly higher than with AQM. This means that thanks to the use of CoDel, the packets reach the recipient faster on average and there is no bufferbloat in the queue of the emulator. If the queue size is further increased in the variant without AQM, the bufferbloat becomes even larger and the latency increases significantly over time.

V. Discussion and Conclusion

Based on our evaluation, Go is the most efficient language in context with raw sockets at the data link layer level. It's able to provide a very high network throughput with a very low latency. With the Queue Emulation Framework a good basis was developed so that many active queueing algorithm can be tested. In this case, the CoDel algorithm has been tested. Indeed, CoDel can combat against buffer bloat and keep the latency below 5ms periodically. The next steps could be the expansion of the AQM emulator with further AQM algorithms and a comparison among them.

References

[1] R. Adams. Active queue management: A survey. *IEEE Communications Surveys Tutorials*, 15(3):1425–1476, 2013.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George , et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[3] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6, 2014.

[4] Bob Braden et al. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, RFC Editor, April 1998.

[5] Greg White et al. Active queue management algorithms for docsis 3.0. Technical report, April 2013.

[6] Rong Pan et al. Proportional integral controller enhanced (pie):a lightweight control scheme to address the bufferbloat problem. RFC 8033, RFC Editor, February 2017.

[7] Jim Gettys and Kathleen Nichols. Bufferbloat: dark buffers in the internet. *Communications of the ACM*, 55(1):57–65, 2012.

[8] Piyush Goyal and Anurag Goyal. Comparative study of two most popular packet sniffing tools-tcpdump and wireshark. In *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 77–81. IEEE, 2017.

[9] Haiqing Jiang, Zeyu Liu, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Understanding bufferbloat in cellular networks. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*, pages 1–6, 2012.

[10] Karamjeet Kaur, Japinder Singh, and Navtej Singh Ghumman. Mininet as software defined networking testing platform. In *International Conference on Communication, Computing & Systems (ICCCS)*, pages 139–42, 2014.

[11] Kathleen Nichols, Van Jacobson, Andrew McGregor, and Jana Iyengar. Controlled delay active queue management. *Work in Progress*, 2013.

[12] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[13] Lili Qiu, Yin Zhang, and Srinivasan Keshav. Understanding the performance of many tcp flows. *Computer Networks*, 37(3-4):277–306, 2001.

[14] Dave Taht Jim Gettys Eric Dumazet Toke Hoeiland-Joergensen, Paul McKenney. The flow queue codel packet scheduler and active queue management algorithm. RFC 8296, RFC Editor, January 2018.