Sampling Linux packet transmission queues with Fast Queue Sampler

Maher Sanalla, Zuher Jahshan, Roy Mitrany

Abstract - Linux offers a tool for managing and manipulating packets transmission, the traffic control subsystem (TC). Besides the ability to configure the kernel packet scheduler (Queueing Discipline - Qdisc), TC also provides the option of sampling the transmission queues.

In this project, we demonstrate Fast Queue Sampler (FQS), a new method developed to sample the transmission queue size (i.e., count the number of awaiting packets to be transmitted).
This method is orders of magnitude faster than its TC equivalent.

To present FQS, we will first review the kernel's Data structures that implement the packet scheduler. We will finish by comparing FQS results against TC results.

Index Terms – Quality of Service (QoS), Queueing discipline (qdisc), Traffic Control (TC), Fast Queue Sampler (FQS)

I. Introduction

Every sample of the qdiscs using TC invokes "fork" system call (i.e., a new process is born) that gives an enormous time overhead for simply sampling the queue. Using TC is fine if there is no need for sampling the qdiscs in high frequency. To (ASK ROY WHY?), a high-frequency sampling method is required; hence TC is not useful.

The information regarding the transmission queues is held by the kernel Data Structures; hence to access them, a code to Linux core must be implemented. For the sake of avoiding a recompilation of the kernel - which takes a lot of time, FQS implementation was written using kernel modules - pieces of code that can be loaded and unloaded into the kernel upon demand.

Furthermore, parameters were passed from kernel space towards user space and vice versa using sysfs - a feature of the Linux 2.6 kernel that allows kernel code to export information to user processes via an in-memory filesystem.

To test FQS against TC, a net topology was built using Mininet - a *network emulator* which creates a network of virtual hosts, switches, controllers, and links.

II. QOS IMPORTANT DATA STRUCTERS

The basic functionality of the Queueing discipline (qdisc) in Linux is to decide how the input network packets will be accepted in the correct order, and on what bandwidth rate. Furthermore, to determine how the output network packets are arranged in queues and transmitted with compliance to an allocated bandwidth rate.

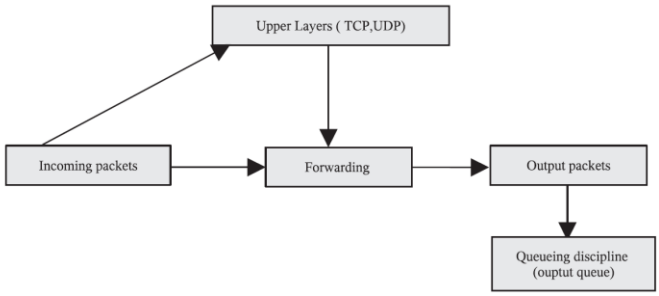

Fig. 1(a). Network packets handling diagram in Linux. [2]

The default qdisc attached to the Linux network interface is *pfifo_fast*

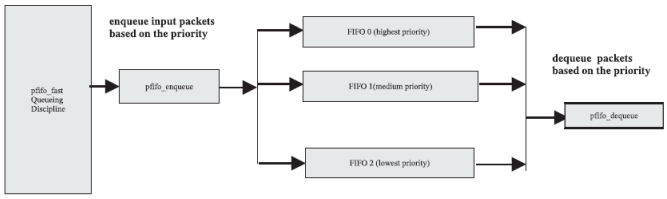

Fig. 1(b). the design of pfifo_fast queuing discipline. [2]

pfifo_fast contains three different FIFO queues based on priority, as seen in Fig. 1(b). The highest priority packets go into FIFO0, and these highest packets are dequeued first before handling any packets in FIFO1 and FIFO2. Similarly, packets in FIFO1 are considered first before any packets handling in FIFO 2.

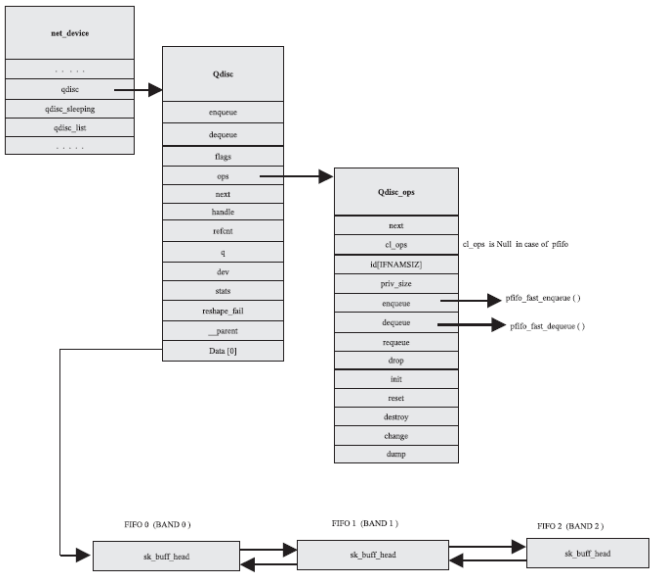

Fig. 1(c). Net device, Qdisc data structures in Linux. [2]

Each network interface is defined by *struct net_device*. This structure has call-back routines specific to hardware. When the module is installed for the network card, the *net_device* object is initialized with device-specific call-back routines and certain parameters in the *init* routine. The qdisc data structure of the *net_device* represents the queueing discipline for that network interface. It contains *Qdisc:data* i.e., pointer to the packet list head inside the queue, In the case of default *pfifo_fast*, this points to an array of *sk_buff_head* structures as seen in the figure above.

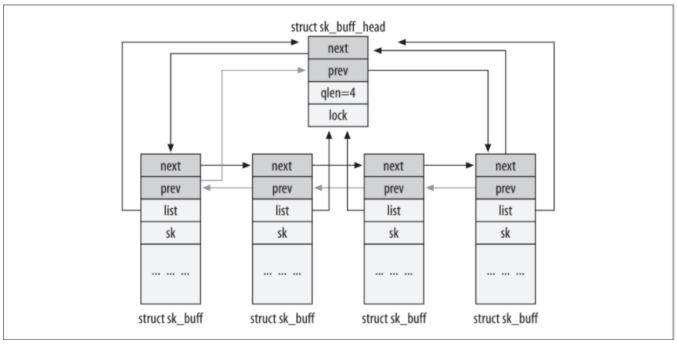

Fig. 1(d). the data structure of sk_buff, sk_buff_head in Linux. [4]

sk_buff_head is simply a list head as described in the previous figure.

Finally, *sk_buff* represents the packet structure on Linux. Thus, the number of packets under a certain *sk_buff_head* is exactly *sk_buff_head::qlen*.

III. IMPLEMENTATION

Algorithm

According to the data structures held by the kernel, FQS algorithm goes through all the list of *sk_buff_head* elements (in the case of *pfifo_fast* there is exactly three of them), sum up the *qlen's* of those sk_buff_head's, and return it.

Formally

- $input: net\_dev$
- $output: packets\ in\ Qdisc$

1) $curr = net\_dev \rightarrow qdisc \rightarrow data$

2) $total\_packets = 0$

3) $while\ curr \neq null$
   a) $total\_packets \mathrel{+}= curr \rightarrow qlen$

4) $return\ total\_packets$

Methods and Refinements

The algorithm was implemented using kernel modules to avoid recompilation of the kernel, which demands roughly 2 hours. Also, the algorithm's correctness is conditioned with the assumption that the queueing discipline used is *pfifo_fast.*

As stated in the algorithm, the net_device must be passed as an input. In Linux, every process lives under a network namespace – a tool that differentiates and separates instances of network interfaces and routing tables that operate independently of each other; therefore, to examine the net_dev held by some process, one must know under what network namespace the process lives. Hence the *process id (pid)* must be passed as an input as well (using *get_net_ns_by_pid* function, the network namespace is obtained). To pass those parameters and return the desired output we used sysfs.

To assure correctness, the net_device lock must be acquired, this lock is responsible that net_dev essential fields will not be removed during the process.

IV. SIMULATION

To test our FQS performance, we created a virtual network using Mininet that contained multiple clients, a single router and a single server, as seen in Fig. (4)a.

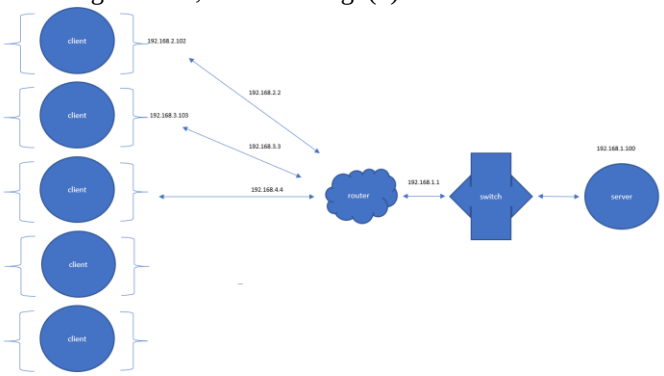

Fig. 4(a). the virtual network used in our testing.

The network contains a variable number of clients that ranged from 1-100. Each client is in a different subnet and is linked directly to the router. The router is connected the server via a switch as seen in the right-side of Fig. (4)a. Every interface in the described topology uses The Linux default Qdisc implementation "pfifo_fast", which meets the preconditions of using FQS. Every link between two interfaces has a limited bandwidth of 1000 bits per second (bps). We set a 500ms delay in the router's qdisc to trigger queue-overload.

The traffic that was transmitted was headed from each client to the server and passed through the middle-man – the router. Intuitively, more clients lead to more traffic through the network, meaning more packets will overload the router. Which results in filling the router's Qdisc.

We used both tools, our FQS and the known TC on the router's environment to check the router transmission queue size over time. In both cases, we used Python script that continuously sampled the queue for a duration of 3 minutes.

When sampling with TC, a new Linux process is created for every sample since it is an independent Linux terminal command, while sampling with FQS can be done via a single process that is continuously running. This behavior led to a difference in overhead and performance of the sampling, as can be seen in the following section.

V. RESULTS

We tested an environment that contains 2,5,20,50,100 clients versus a single server. For each configuration, we measured the total number of successful samples using both implementations – TC, FQS.

Moreover, for every possible absolute variance between two consecutive samples - we counted its number of occurrences. As can be seen in Fig. 5(a).

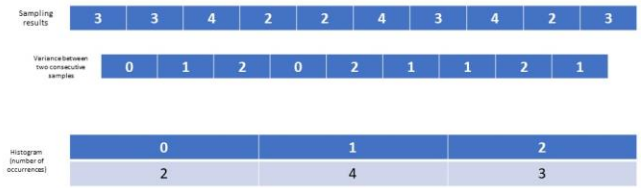

Fig. 5(a). An example of the variance measurement.

Figures 5(b)-5(f) show the results of our simulation.

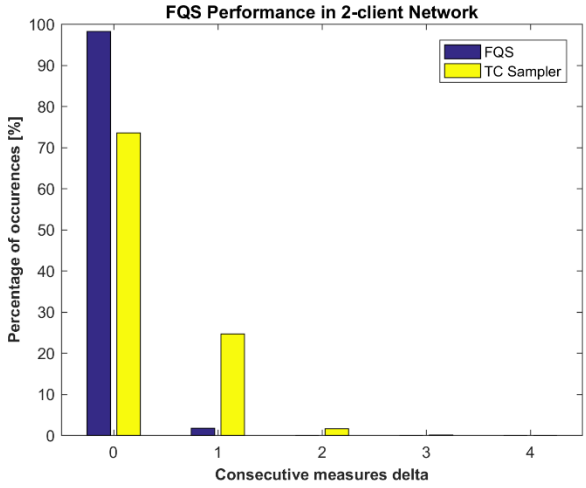

Fig. 5(b). "FQS vs TC measures variance" in a 2-client-1-server network.

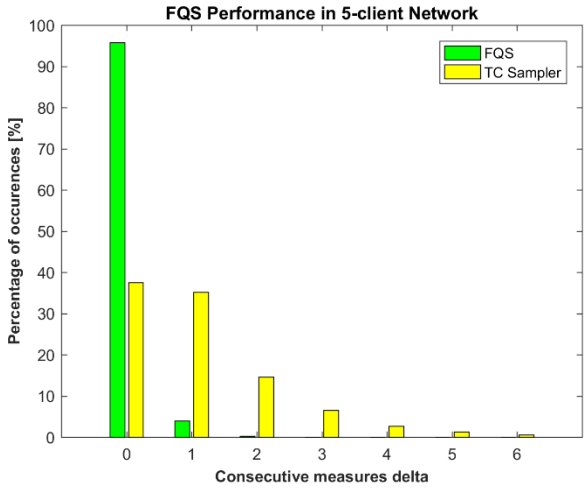

Fig. 5(c). "FQS vs TC measures variance" in a 5-client-1-server network

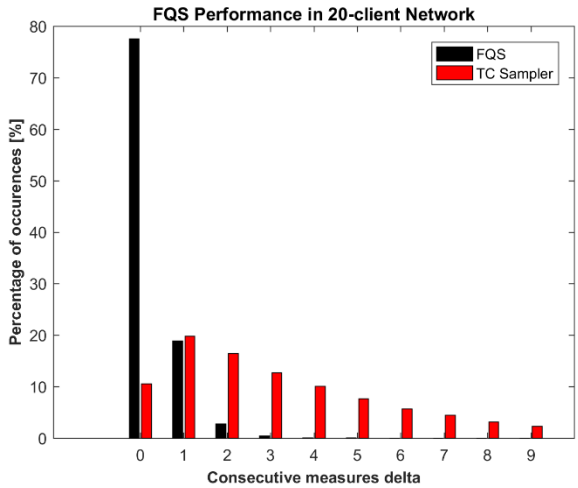

Fig. 5(d). "FQS vs TC measures variance" in a 20-client-1-server network

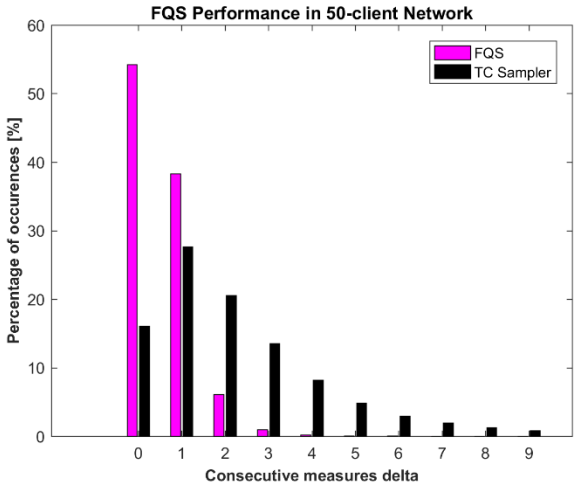

Fig. 5(e). "FQS vs TC measures variance" in a 50-client-1-server network

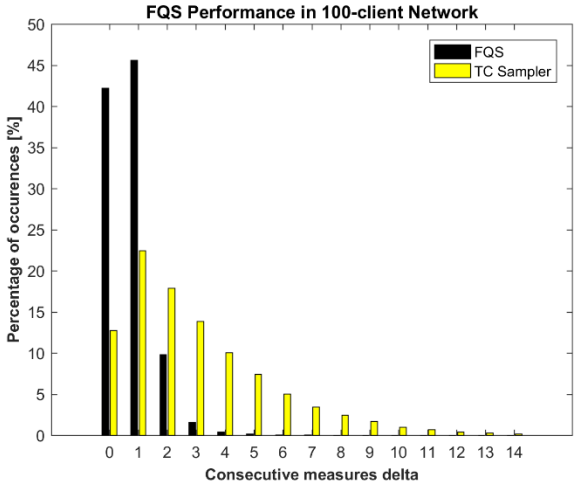

Fig. 5(f). "FQS vs TC measures variance" in a 100-client-1-server network

VI. Evaluation

Based on our results from the figures 5(a)-(f), it can be seen that the FQS sampler than TC sampler as we will elaborate. First of all, the percentage of low-variance measures (0-1) is much higher in FQS samples than in TC samples. This indicates that the FQS was able to catch queue events more clearly i.e., FQS samples represented a more refined partition of the test-interval than TC. Figures 6.(a)-(c) highlight this point furthermore.

Number of clients	0-variance [%]	1-variance [%]	0-1 variance [%]
2	98.2	1.7	99.9
5	95.7	3.9	99.6
20	77.6	18.8	96.4
50	54.2	38.3	92.5
100	42.2	45.6	87.8

Fig. 6(a). the percentage of Low-variance measures using FQS under different network environments.

Number of clients	0-variance [%]	1-variance [%]	0-1 variance [%]
2	73.5	24.7	98.2
5	37.6	35.2	72.8
20	10.5	19.7	30.2
50	16.1	27.6	43.7
100	12.7	22.4	35.1

Fig. 6(b). the percentage of Low-variance measures using TC under different network environments.

In addition, it can be noted that FQS is much faster than the TC, since it performed much more samples than the TC sampler did in the 3-minute interval - As shown in Fig. 6(c).

environment	2	5	20	50	100
Number of samples for 3 minutes (FQS)	33,349471	24,668,412	6,091,856	5,070,866	4,176,729
Number of samples for 3 minutes (TC)	80,805	74,678	58,002	55,786	48,312
speedup	x412	x330	x105	x90	x86

Fig. 6(c). "TC VS FQS" – comparing the total number of samples performed throughout the 3-minute test.

Fig.6(c) emphasizes the fact that FQS sampler is more efficient than its TC equivalent. Which can be explained due to the fact that every TC sample taken leads to the creation of a new process where on the contrary, FQS uses only a single process to perform its samples.

The high frequency of FQS samples - 200K samples per second (i.e., a sample every 5[μs]) in comparison to the TC's 0.5K samples per second (i.e., a sample every 2[ms])– enables it to capture more queue-events (arrival/departure of a packet). Formally, the probability that a sample occurs between two consecutive queue-events is larger in FQS rather than in TC. Hence, the percentage of 0/1-variance samples is much more prominent in FQS as can be seen in Fig.6(a)-(b).

Moreover, it can be noted that the performance of FQS decreases gradually when the number of clients increases – 33 million samples drops to only 4 million (Fig. 6(c)). This can be explained due to the fact that every Mininet client is considered an independent process. Thus, increasing the number of clients means increasing number of processes running on the CPU. In that case, the OS will preempt the FQS more often. Meaning that the epoch allocated for FQS is shortened. This leads to a decrease in the total number of samples.

The probability of capturing a queue event decreases accordingly – as can be seen in Fig.6(a) "0-1 variance" Column. Equally affecting, is the larger number of clients which generate heavier traffic through the network.

Lastly, according to Fig.6(c), the FQS/TC speedup decreased dramatically as the number of clients rose. This can be explained by the following thesis:
FQS implementation writes the samples to a buffer, and only after the buffer is full it prints it content. While its safe to assume that TC does not use memory as often, it simply reads the queue length and prints it. Granted the fact that the larger the number of processes the larger the number of cache-misses[3], FQS will be more affected due to the increase of processes when compared to TC since FQS is more memory-reliant than its equivalent. Consequently, increasing number of

clients which means increasing the number of processes will affect the FQS more.

VII. CONCLUSION

We presented in this paper a pfifo_fast transmission queue sampling tool which can sample at a very high frequency - FQS. (mention Goal of FQS – TODO Roy)

Linux traffic control subsystem (TC) has a similar tool as the one presented in this paper. We tested the performance of both tools under different environments and analyzed the results. Overall, we can deduce that FQS is significantly more efficient and accurate than the TC sampler.

Future work may be dedicated to upgrading FQS to support a bigger range of Queuing-Disciplines such as Token-Bucket filter Queue(TBF), Class-Based Queue(CBQ), etc.

VIII. REFERENCES

[1] Patrick Mochel, “The sysfs Filesystem, Proceedings of the Linux Symposium”, vol. 1, pages. 321-334, 2005

[2] S.Seth, M.Ajaykumar “TCP/IP Architecture, Design and Implementation”

[3] M.Jahre, L.Natvig “Performance Effects of a Cache Miss Handling Architecture in a Multi-core Processor”

[4] C.Benvenuti “Understanding Linux Network Internals.”