Manually Testing Queueing Algorithms

Abiyaz Chowdhury

Rohit Chatterjee

Course instructor: Prof. Aruna Balasubramanian

I. Introduction

The entire point of the internet is enabling the transfer of packets from one end system to another. Being a huge network with a huge amount of effort going simply into the sustenance of the structure, the internet only provides best-effort delivery guarantees. It is no surprise, therefore, that a lot of thought goes into studying and dealing with packet lost. Packets may be lost in a variety of ways: by being directed to a wrong address, by not being correctly read or accepted and their end destinations, being corrupted or incorrectly split, due to congestion inside the network, and so on. Among these, congestion is often the most pervasive (and pernicious) type of packet loss, and so a great deal of effort goes into minimizing network congestion through various means. This is of course the motivation behind the celebrated congestion control mechanism in TCP, but there are other means to tackle this problem. Notably, TCP congestion control is implemented only at end systems, and therefore does not directly handle local congestion at midpoints or routers - which is really most of the network.

Recall that a router is a device that has multiple connections to other devices on the internet, and its basic function is to forward packets along correct paths to their destination. Thus router function is not immediate, since the router must determine where it must send each packet once it receives it. Further, each connection on the internet has its own capacity or bandwidth which fixes the maximum rate at which packets can be sent along this connection. The upshot of all this is that there is a certain maximum rate at which a given router can despatch incoming packets, and thus if it receives packets at a grater rate then there is congestion at the router. The way this is dealt with is straightforward: the router has a predefined amount of buffer memory to deal with stranded packets, and this is called the queue. The process of managing these packets and using the queue is called queueing. The most natural (and widely used) way to perform queueing is to do it in a first-in-first-out manner: the earliest recieved packet in the queue is sent out first by the router (hence the term queue), and the queue receives packets as long as packets come in faster than they can be sent out. Once the queue fills up, subsequent incoming packets are all dropped till the queue has space to accept more packets again.

On the face of things, one would not expect anything more from the queueing algorithm; there seems to be no apparent reason why the functioning of the queue should be related to the network performance. In other words, if the network is otherwise well-managed, we might expect such congestion to be short in duration, and therefore assume that the queue absorbs such bursty behaviour and in fact smoothens out traffic heading out from the router. However this is not always the case, and we can see that this method of queue operation may actually hamper network performance in some scenarios. For example, consider the case where multiple flows are present in the network and these flows happen to synchronize in such a manner that only a few flows are always present at the head of the queue, and consequently several other flows have

their packets dropped repeatedly. Thus the network only properly services these first few flows, and the other flows in the network receive very little bandwidth. This phenomenon is known as *lockout*. Another undesirable scenario can be brought about by prolonged congestion. If a lot of packets arrive to an already full queue, many packets are dropped simultaneously. This can trigger transport layer level congestion control mechanisms, leading to synchronized responses (back-off) among multiple flows. As a result, the overall load on the network may drop below the capacity of the link and then rise back to exceed the link's capacity resulting in a full queue and so on, giving rise to large oscillations in the network utilization. This oscillating behavior is opposite of the buffer's expected function, which is to also act as a smoothing filter. It also implies inefficient bandwidth usage. This phenomenon is known as *full queue*.

Thus we see that it is not straightforward to manage queueing behaviour with congestion control in mind. Ideally, a good queueing algorithm should be able to identify signs of imminent congestion and high traffic volumes, and take actions to avoid such outcomes. In more concrete terms, the queueing algorithm should manage to keep the average queue size low for the most part, by taking specific actions such as dropping or marking packets (thereby initializing transport level congestion control mechanisms). A low queue size implies that the router can handle and smoothen out bursty traffic, and also that the "queue lag" for any given packet is low: this is the delay the packet faces from just being in the queue till it is sent out. There are various queueing algorithms that try to accomplish this in different ways. In this project, we try to manually run some of these algorithms on a virtual network (using Mininet) and try to analyse their performance and compare the results.

II. Problem Statement and Methodology

We do our study on Mininet (MiniNExT, to be exact). Mininet is a very powerful platform to set up virtual networks, and offers a lot of flexibility in fine-tuning network parameters. Apart from letting the user set up custom network features and topologies, it offers router functionalities that also cover queueing mechanisms and and associated parameters. Mininet allows one to adjust queueing behaviour at as router in the following two ways:

- One can use Openflow functionalities at a router (using the inbuilt OpenvSwitch functionality that specifically deals with virtual routers and switches). This allows the user to set upper and lower size limits for the queue, and also implement additional queue-like structures for flows or groups of flows known as meters.
- One can use the Linux *traffic control* functionality, **tc**. This allows much more fine grained control over the queueing process, and allows the user a choice between various queueing algorithms, and allows them to adjust various related parameters such as queue size limits, behavior related parameters, and action related settings (whether to drop/mark packets etc).

As such, these options are mostly enough for practical purposes, i.e. running a virtual network that implements some feature, or studies some other aspect of network performance. We felt however that these methods do not allow us fully to studying the queue algorithms in detail. Things like throughput, packet loss and approximate queue sizes can be estimated using these means, but the actual behavior of the queueing algorithms seemed to elude us. We therefore decided to implement several queueing algorithms ourselves and see how they performed, to get a better understanding of these algorithms.

To that end, we manually implement and inspect three queueing algorithms: the most basic FIFO queuing algorithm, a slightly different, *delay based* queueing algorithm that dispatches packets

in the orderthey were *generated*, and finally the celebrated Random Early Detection algorithm. The questions we try to address are as follows: How do these algorithms behave, in terms of queue size? How do they handle individual traffic? And how do they compare to each other?

III. Model discussion

IV. Model evaluation

V. Conclusions

References

[1] James H. Stock, Mark W. Watson, *Forecasting inflation*, Journal of Monetary Economics 44 (1999) 293-335