

Programming Project #4: TCP Tahoe with a Concurrent UDP Flow

Computer and Communication Systems

Dr. Marsic

Group #10:
Owen Healy
Mark Rusinski
Zhi Yang
Kevin Bhavsar

12/6/2011

Problem and Procedure

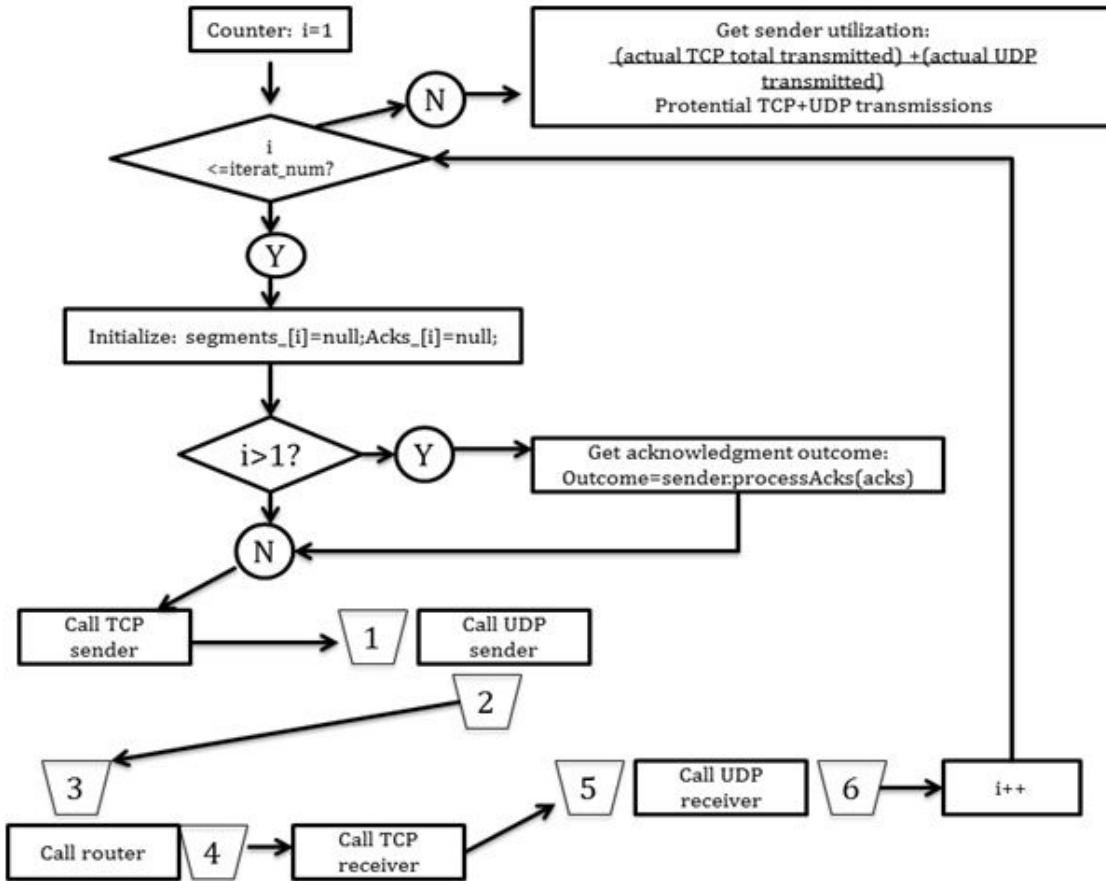
The purpose of these tests was to analyze the effect of a concurrent User Datagram Protocol (UDP) flow on Transmission Control Protocol (TCP) throughput. The major TCP congestion control variables such as flight size, congestion window, effective window, and SSH threshold were monitored to provide an in-depth, graphical analysis of the changes produced by the addition of UDP data packets. The TCP sender utilization and total throughput of the TCP data stream were both observed as well to provide a broader overview of data throughput and packet loss effects due to the competing UDP.

The base case for the trials with UDP implemented had the UDP sender operating on 4 consecutive round trip times (RTTs) ON followed by a 4 RTT OFF period. During every ON period there were 5 UDP packets sent to compete with the TCP for router resources. From this base case both the ON/OFF interval and packet quantity were altered in two respective experiments to determine the effects of a concurrent UDP flow during TCP transmission. Some special cases were also observed in relation to the timing of the UDP packets. The implementation of larger bursts of UDP packets sent in 1 RTT ON with long periods of OFF time provided the data for the aforementioned special cases.

In order to run these tests an augmented version of the provided JAVA project was used. Many and the classes were altered with the DualRouter and UDPSender being the most prominent additions. The new router code incorporates both TCP and UDP packets and contains a random shuffling algorithm to ensure no preference is given to either flow. The newly added UDP sender contains all of the parameters that vary the nature of the UDP flow such as the ON/OFF interval and packet quantities for every interval. Additionally, a UDPSegment class was added to differentiate the incoming packets, and several simulator operations were modified in order to integrate the classes together. The experiments were all run with this newly augmented code.

Simulator

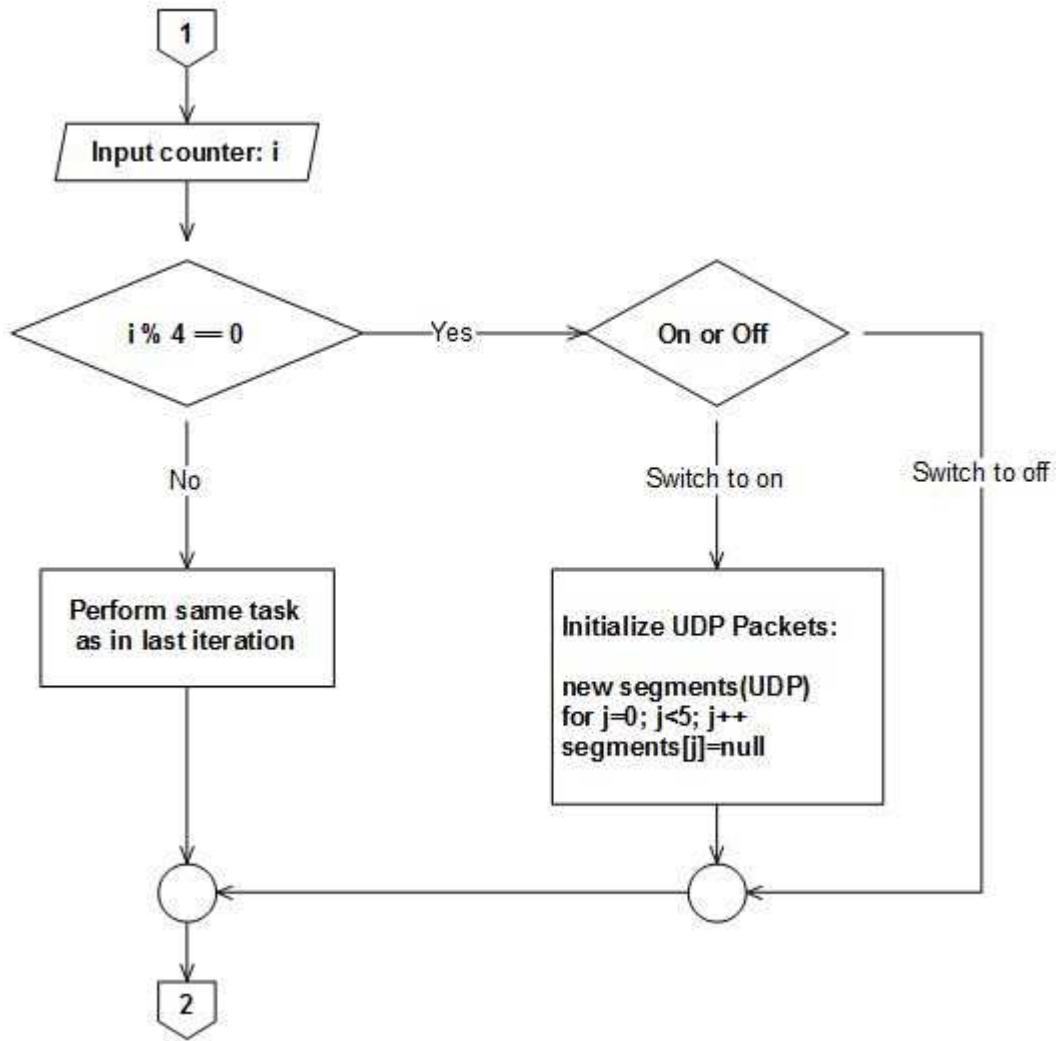
Simulator.run()



This is the modified simulator based off the reference example found in TCPsimulator from the website. For the most part the same algorithm from the reference example was used except for UDP sender, Router relay, and UDP receiver. UDP sender and UDP receiver are two new classes that will be created to implement the UDP flow, and we will be alternating the router class to implement the flow of both UDP and TCP at the same time. The newly developed components are describe in more detail in the following flowcharts.

UDP Sender

UDPsender.run(i)



The UDP sender is responsible for sending five packets over four consecutive iterations, or round trip times (RTT), since they are equivalent in this example. This is followed by an off period of the same length. The sender initially keeps track of the iteration number through the counter variable *i*, that is passed from the Simulator file. The sender recognizes every fourth round trip time by dividing the current iteration number by four, and if the remainder is zero sender is switch into the on state if it was previously off and vice versa. If the sender is activated it initializes the segments class that creates five new UDP packets. These packets are then passed to the router. In the event that the UDP sender is not activated the aforementioned process is skipped.

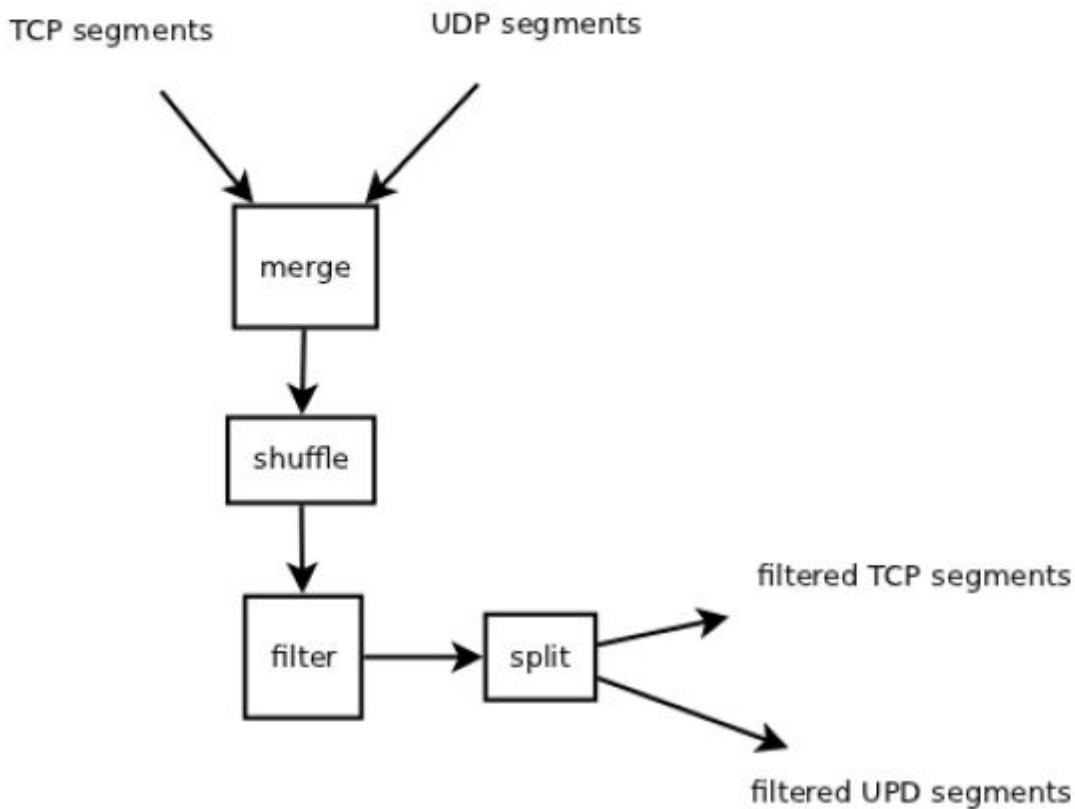
Router

The role of the router is to drop segments. TCP and UDP segments compete for non-dropped slots; hence, the more UDP segments that occur in a simulation, the more TCP segments will be dropped.

The router is stateless. Each simulation step, data flows through the router as follows:

void Router.relay(Segment[] tcpSegments, Segment[] udpSegments)

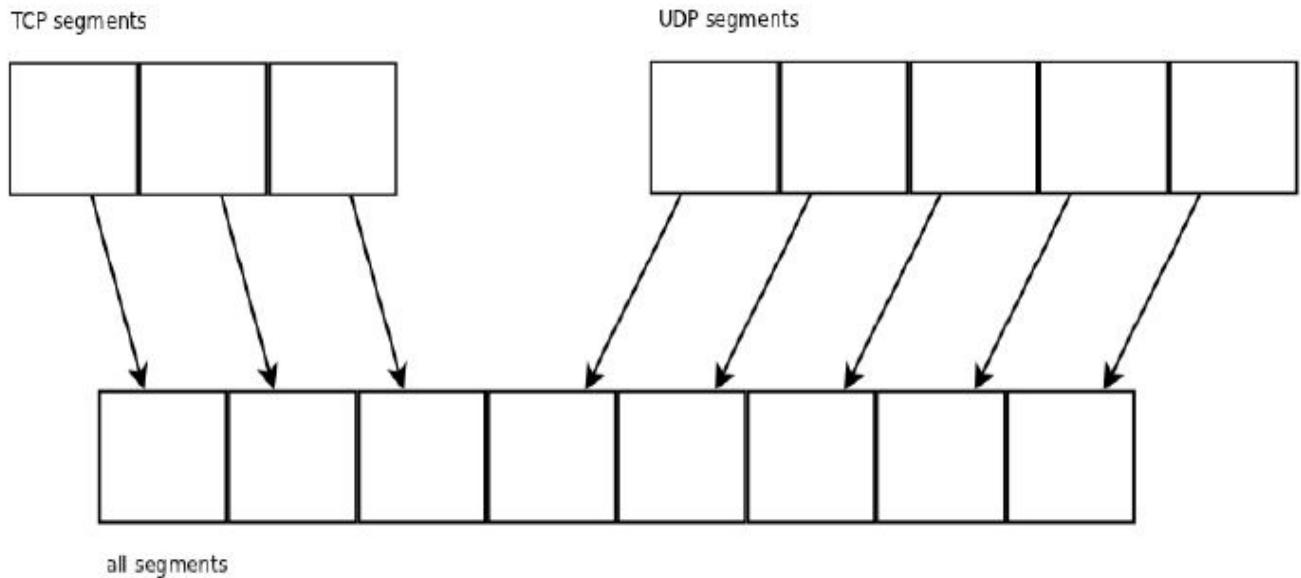
3



4

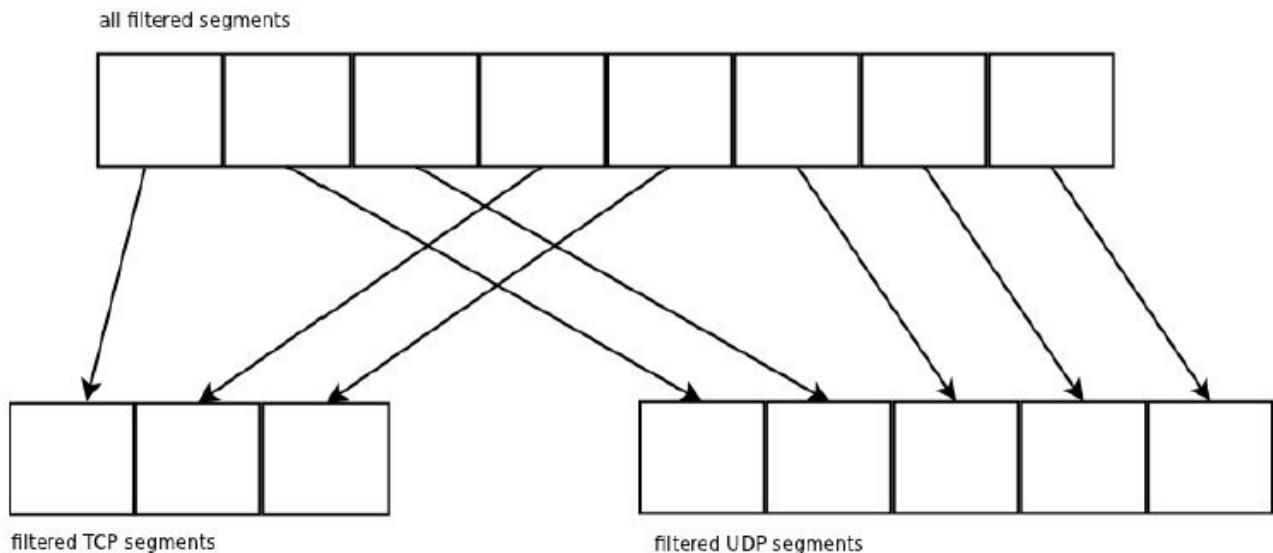
In more detail, the steps are:

Merge



The arrays of TCP and UDP segments are simply concatenated into a single array.

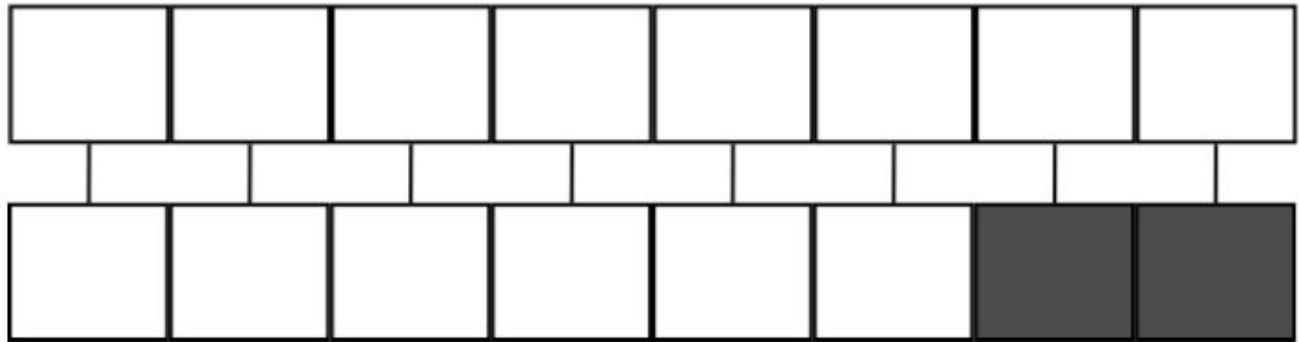
Shuffle



The array of all segments is randomly permuted.

Filter

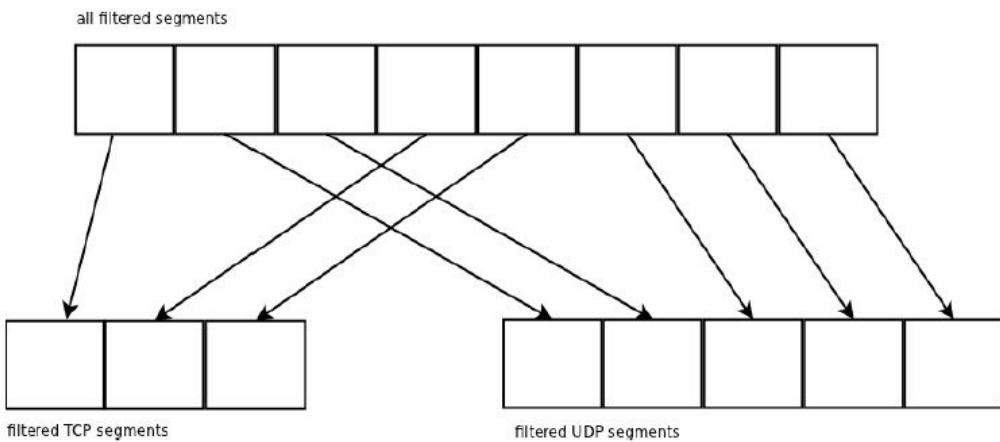
all segments



mask

Filtering involves stepping over the array and null-ing elements according to a predefined mask. The mask is fixed, but because the order of segments in the array is random, all segments have the same (though dependent) chance of being dropped.

Split



The segments from the shuffled, filtered array are then copied back into their original TCP and UDP arrays, based on their protocol types. The arrays are modified in-place; no data is explicitly returned to the caller.

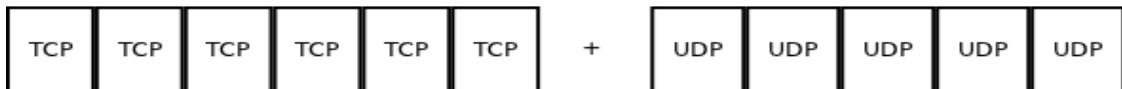
Dropping Segments Fairly

The router was designed with 3 goals in mind:

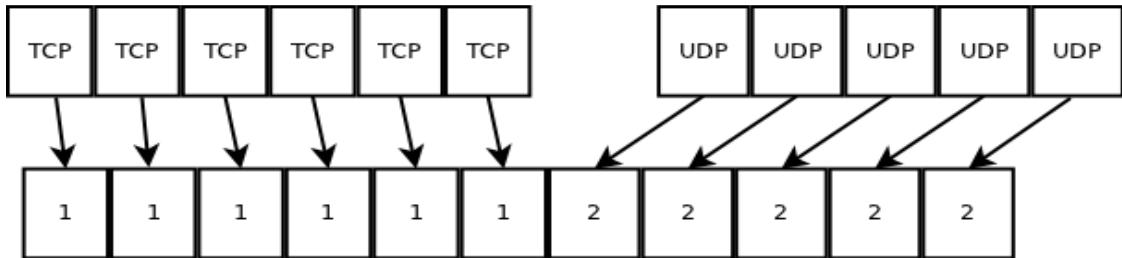
1. When both UDP and TCP are flowing there should be no preference between the priority of the two flows, and their placement should be decided randomly.
2. Segments should not be reordered, even when some are dropped.

3. When no UDP is flowing, the router will behave the same as the TCP-only router, in which packets are dropped entirely deterministically.

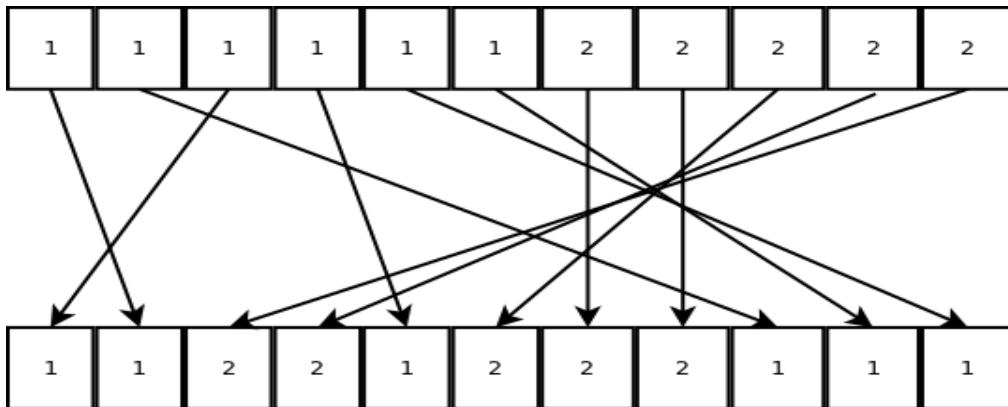
The Router receives an array of TCP segments and an array of UDP segments:



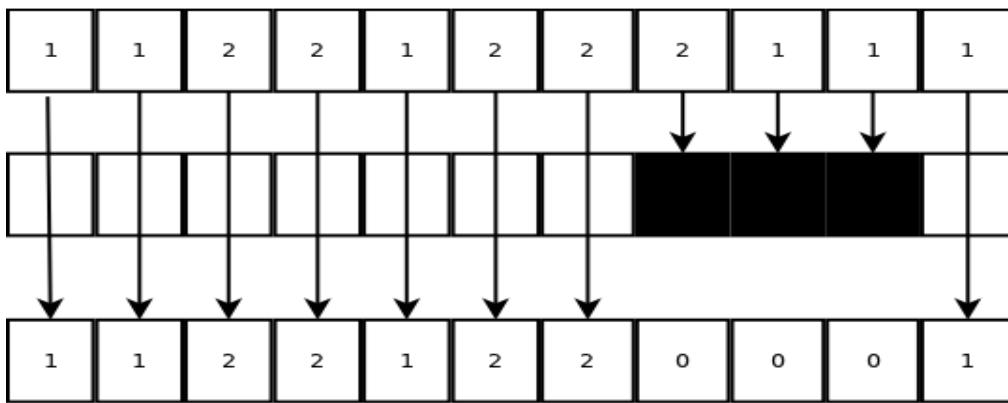
It counts each array and produces an array filled with as many 1s and 2s:



The array is then shuffled:



A preset mask is applied:



Then, the remaining number of 1s and 2s are used to decide how many segments from each flow get through. The original arrays are truncated to that number. Because only the array of 1s and 2s is ever shuffled, and not the original array, the segments are never rearranged. This preserves the current router contents for the next iteration.

Results and Discussion

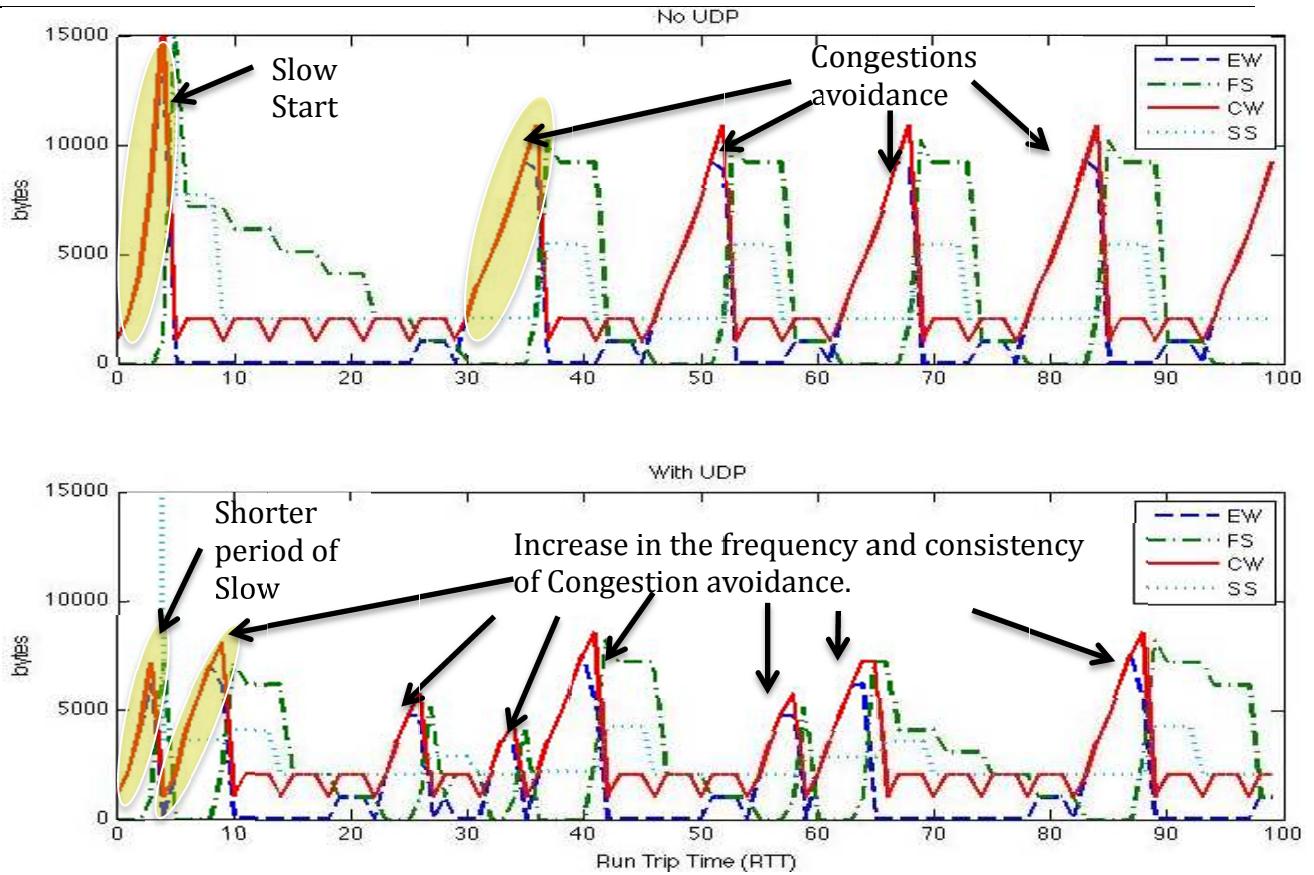


Fig.1: TCP Transmission with & without Concurrent UDP Transmission.

- Without UDP transmissions competing for resources, TCP congestion parameters are graphed in figure 1 over 100-transmission rounds. As observed, TCP kicked off with slow start mode, where the congestion window would increase exponentially. Upon the received of 3 dup acknowledgements, TCP would enter a period of fast retransmission and followed by a period of congestion avoidance. The occurrences of congestion avoidances are periodic as shown in the graph.
- With UDP transmission come into play, the occurrence of congestion avoidances become inconsistency and more frequent over the period of 100 RTT as observed in the bottom graph in the previous slide. Furthermore, the period of slow start in the beginning of the transmission is much shorter due to the incoming UDP packets that caused buffer to reach its limitation faster (since in this simulation, UDP sender operates in 'on' mode to start with). On top of that, the congestion window decreases when UDP sender is on; this is shown in the figure 2.

Now, considered the zoomed in portion of the congestion parameters graph with UDP sender from RTT 32 to 44 showed figure 2. The shaded area is when the UDP sender is on. The graph shows that when UDP sender is on, the TCP has smaller congestion window because its period

of congestion avoidance is much shorter than when UDP sender is off. Furthermore, the consecutive occurrence of the congestion avoidance happened because:

1. There was only one unacknowledged TCP packet at that 35 RTT due to the competing UDP packets before.
2. UDP sender went from on to off mode.

Thus, congestion avoidance takes off after 36 RTT without interruption, and lasted a much longer period.

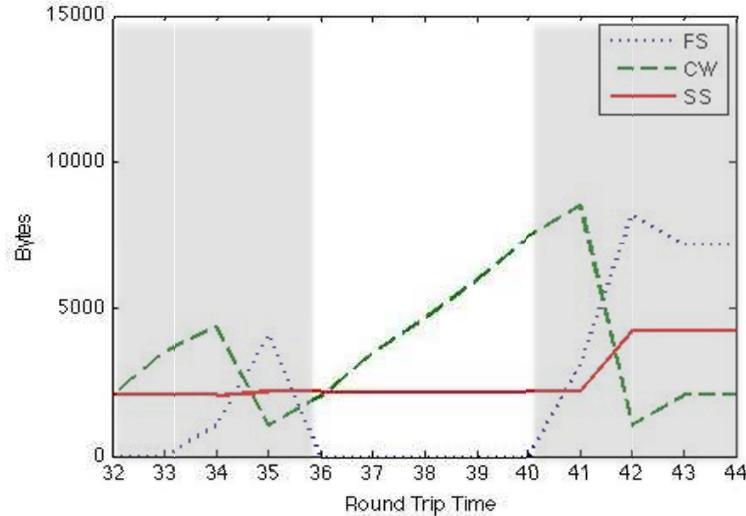


Fig.2: TCP Congestion Parameters from 32 RTT to 44 RTT.

Next, let's look at the bar graph below, which compares the average TCP congestion parameters between transmission with UDP and without UDP. As a result, it shows that TCP congestion parameters on average perform at a higher byte size without UDP than with UDP. This result is theoretically valid because with UDP packets competing for buffer space, TCP is more likely to experience packet losses. And because upon the encounter of packet loss, TCP lowers its congestion parameters to limit the number of packets sent to the router.

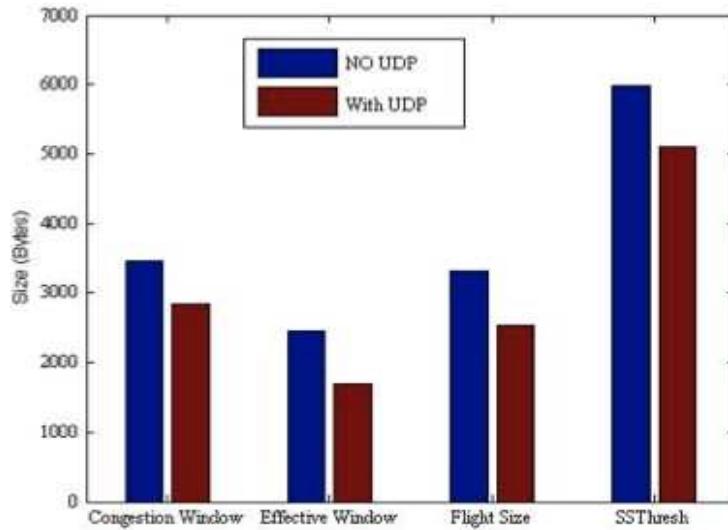


Fig.3: Average TCP Congestion Control Parameters

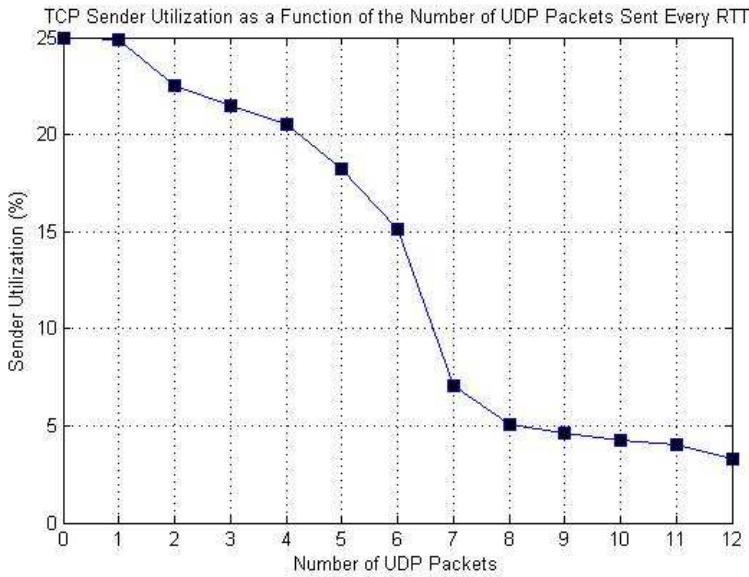


Fig. 4: TCP Sender Utilization as a function of UDP Packets.

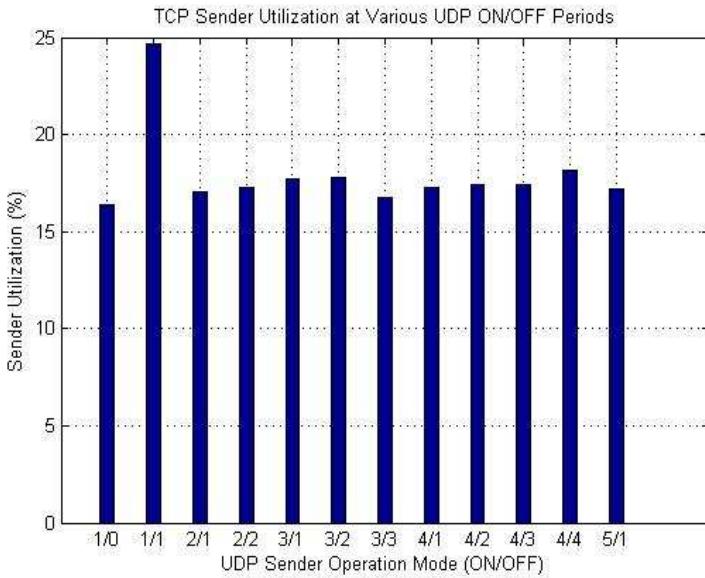


Fig. 5: TCP sender utilization at varying on off intervals

A comparison of Fig. 4 and Fig. 5 shows that the quantity of UDP packets per RTT affects the TCP flow more drastically than the ON/OFF interval of the UDP flow itself. TCP sender utilization is drastically decreased as the UDP flow packet quantity is increased producing variations from 25% to 3%, while nearly all of the ON/OFF intervals vary around 17% utilization. This leads to the conclusion that it is more important how many packets flood the router at one time rather than when the packets are sent; though the timing still matters in special cases.

Figure 4 shows the effect of increasing the quantity of UDP packets in every round trip time (RTT) under a constant ON/OFF interval of 4/4. As more unwanted UDP packets flood the router the sender utilization shows a clear decrease in TCP throughput. Once the quantity of UDP is greater than 7 packets the drop in performance slows down as it approaches the limit of zero.

The bar graph in Fig. 5 shows the effect of different UDP sender ON/OFF intervals on TCP sender utilization. In these tests the number of packets sent every RTT was kept constant at 5 packets. From the bar graph it can be seen that variations in ON/OFF intervals do not reveal any dramatic changes in sender utilization with the exception to the 1/1 interval. However, there appears to be a slight upward trend as the off intervals become longer. There are two exceptions to these observations; the 1/1 interval and the 3/3 interval. This was most likely due to the way in which the UDP and TCP flows overlap in these specific cases. This occasional phenomenon is explained in more detail later.

The plot below shows cumulative TCP segments that make it through the router, not including the ones that are sent but then dropped, both with and without UDP being sent concurrently. This figure gives an idea of how UDP interrupts the TCP flow -- it doesn't always interrupt it, but if TCP tries to send the bulk of its burst during a time when UDP is on, CongWin gets cut shorter than it would otherwise.

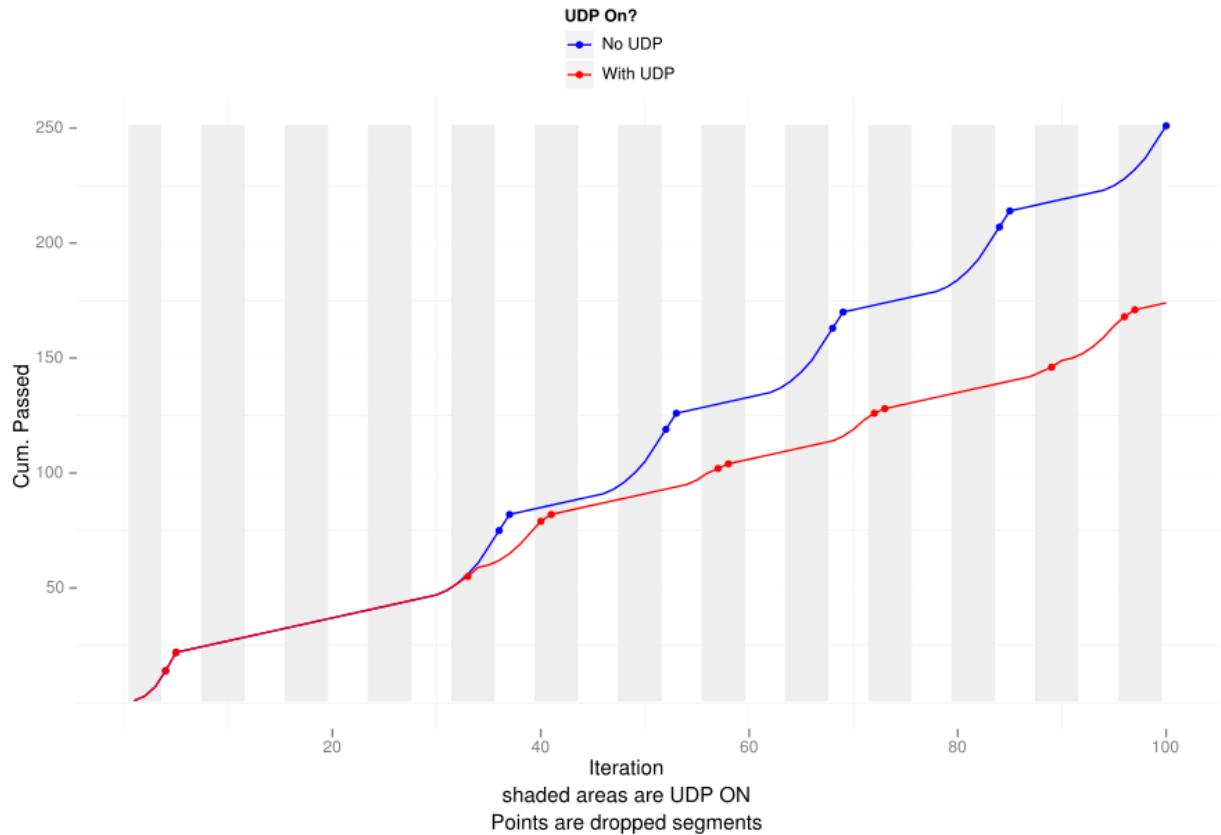


Fig. 6: The cumulative number of TCP segments received with a concurrent UDP flow (red) and no UDP flow (blue).

One interesting consequence of this is that it is not simply the amount of UDP flow that matters, but the periodicity of the flow as well. The plot below shows UDP aggregate throughput (total segments during the simulation) on the x-axis, and TCP aggregate throughput on the y-axis. That is, the x-axis doesn't show the shape of the UDP traffic, it only shows the quantity that was passed through the router.

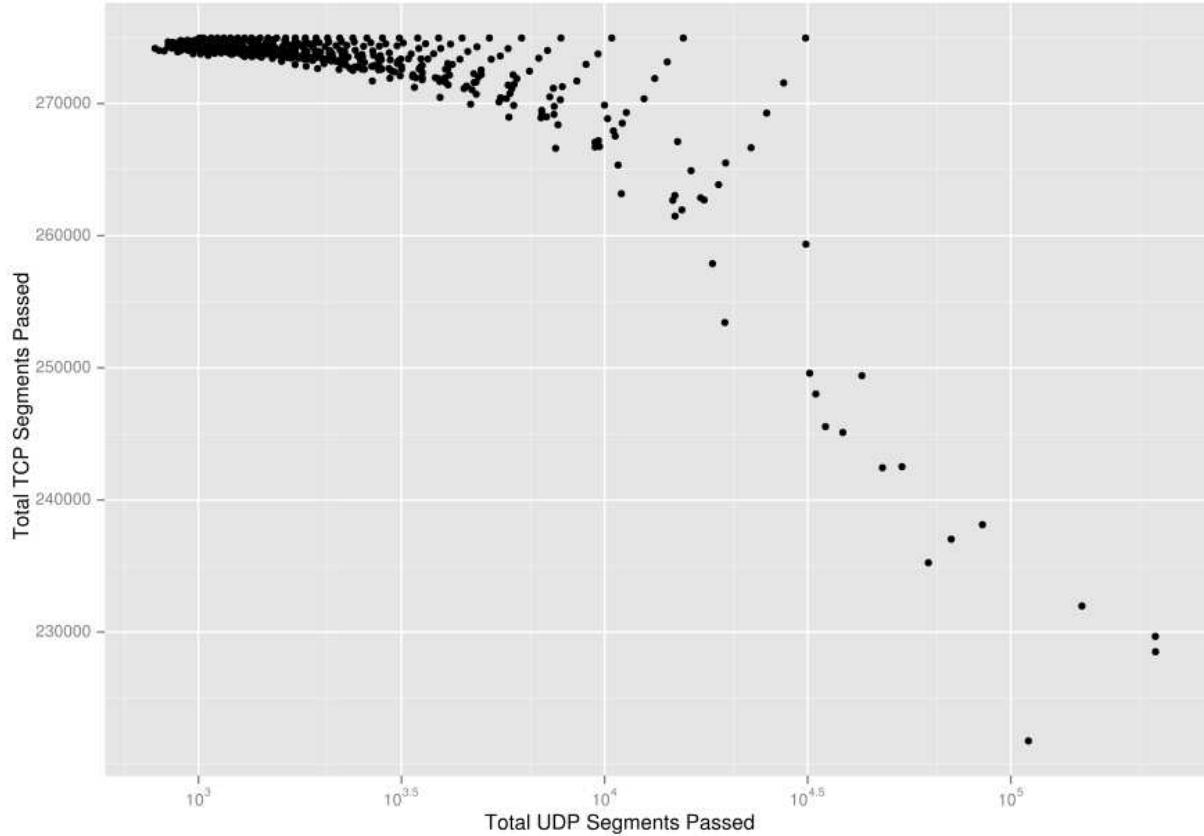


Fig. 7: The number of TCP segments received compared to the amount of UDP packets passed.

Even though Fig. 7 does not show the shape of the UDP traffic, you would expect that towards the right of the x-axis, where more UDP packets are passed, the UDP Sender is sending segments either more often or at higher amplitudes, both which will interfere with TCP and reduce TCP's throughput.

And indeed that does happen, for the rightmost 1/3 of the graph. But in the left 2/3s of the graph, it's a much different story: the bottom envelope of the graph does indeed trend downwards, but the upper envelope stays right at the top, as if the UDP flow wasn't even there.

It turns out that the way this was generated was by making a UDP flow that was very sparse: it is on for only one cycle at a time, and the throughput is changed by changing the gap in between active cycles as seen in Fig. 8.

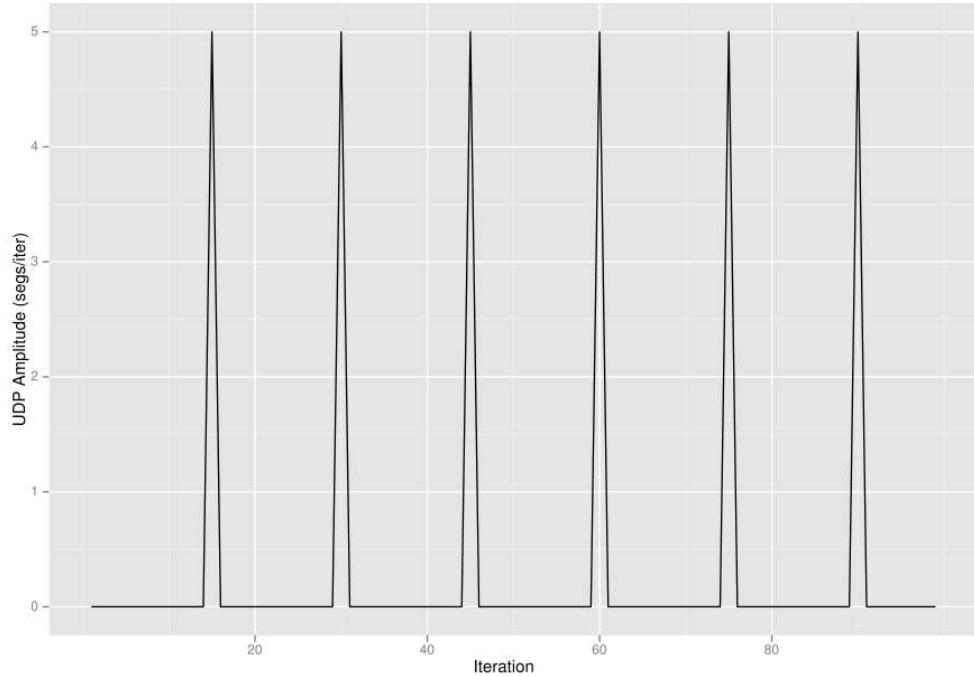


Fig. 8: Visual representation of the nature of the UDP flow used for these test cases.

And, since UDP does not always interfere with TCP, if it's spaced just right, it does not interfere at all. Figure 9 provides an example of this phenomenon.

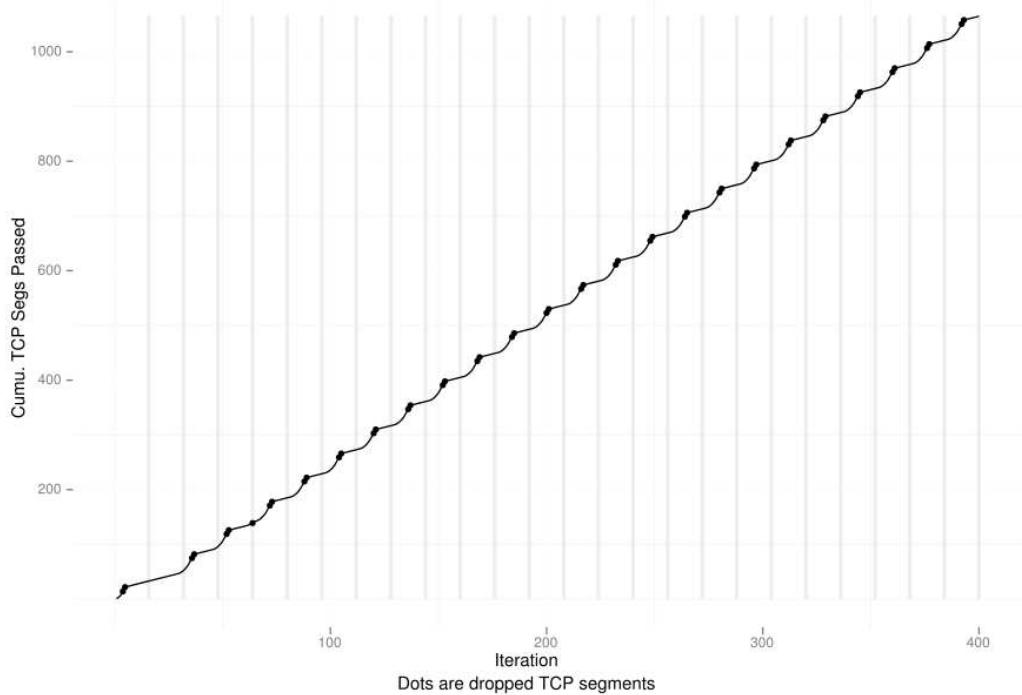


Fig. 9: Cumulative TCP packets received over 400 iterations with a UDP ON/OFF interval of 1/16.

The period of the UDP traffic lines up *perfectly* with the natural oscillations of the TCP traffic, so there is no effect. Of course, just because they have the same period doesn't mean they will line up in a nice way: they might line up in a bad way. But, you can see from the plot that when this *does* happen (near the beginning), TCP phase-shifts so that it is out of the problem area, and the problem never comes back. (*Another clue that this is what is going on, is that in the original throughput plot, if you look at which points had perfect TCP throughput, those are ones where the gap is a multiple of 16.*) Just one point over, at a gap of 15, it is not possible for the two oscillations to line up so the TCP flow will always be interrupted as seen in Fig. 10.

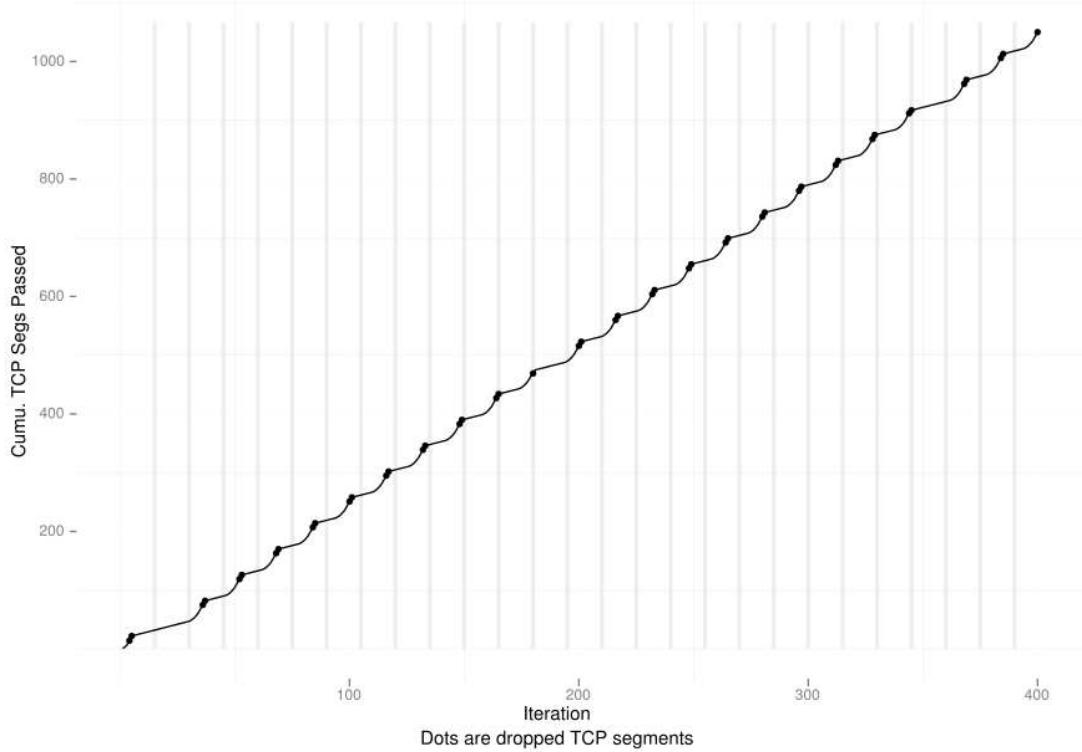


Fig. 10: Cumulative TCP packets received over 400 iterations with a UDP ON/OFF interval of 1/15.