CS 143 Network Simulator

Project Report

Sith Domrongkitchaiporn
Sushant Sundaresh
Ahmed Alshaia

December 2015

Table of Contents:

Usage

Operating from the NetworkSimulator/Code/Python directory as ~/, the program may be called from the command line as

python visualize.py --help
python visualize.py --testcase [filename.json] --tcp [reno|vegas|fast]

The test case must be a JSON formatted file in the format of the provided *input_test1.json*.
Units are

- megabytes (flow size)
- megabits per second (link rate)
- milliseconds (propagation delay)
- kilobytes (link buffer)
- seconds (flow start time)

The order of element definitions must be *host, link, router, then flow.*

Plotted data for the following measurements

- per link, link rate (Mbps)
- per link, per direction, buffer occupancy (%)
- per flow source, packet loss as detected at a link (packets)
- per flow source, flow rate as detected at the flow sink (Mbps)
- per flow source, packet round-trip delay (ms)
- per flow source, window size (packets)

are presented in ~/results/plots. This is all the user should need to see.

Aside: some useful raw data for debugging purposes is presented in ~/results/rawdata, parsed by network element. Units are not provided. All measurements observed by the system (in JSON format) are presented in the text file ~/results/all_measurements.txt.

The interested user may additionally modify a portion of *constants.py* to play with congestion control parameters.

Source Listing

- visualize.py
    - constants.py
    - main.py
    - jsonparser.py
    - event_simulator.py
- reporter.py
    - flow.py
        - tcp_fast_working.py
        - tcp_reno_working.py
        - tcp_vegas_working.py
    - node.py
        - host.py
        - router.py
    - link.py
        - train_link.py
        - link_buffer.py
- packet.py
- event.py
    - events.py
- unit_test_benches.py
    - Test benches for most interfaces between objects
    - No longer relevant or functional; too many one-off debugging changes were made

System Architecture

UI Wrapper

Startup

The script *visualize* cleans up its working ~/results directory of the files described above, then calls upon a *main* loop to get things running. It routes the STDOUT stream from the main loop to *all_measurements.txt* and updates progress in the STDERR in the terminal.

The main loop uses a *JSONParser* to parse the input JSON file. It receives a map between network IDs and instantiated network objects from the parser. It passes this map to a discrete *event_simulator* that immediately informs every element who the simulator is, so every element can access the global time. This simulator then requests some last minute setup for flows & routers, now that all elements have been instantiated, and jump starts the simulation by enqueuing events to start flows at specified times, and request the first routing update.

Flows ping the STDERR with their percent completion after a *constant*s defined number of new acknowledgements.

Termination

The simulation ends when every flow source has received an acknowledgment for every single packet. The event simulator will still have flow and router callbacks enqueued at this time.

Post Processing

The *all_measurements* file is then parsed in *visualize* into a map between network elements and any datasets associated with them. Supported measurement JSONs are defined in our *constants* file.

For every link and flow, the measurements defined in *Usage* are plotted. Measurements for which data is not available (e.g. if no packet losses occurred) are not plotted.

Network Elements

Packet

There are 4 different types of packets:

1. Data Packet
2. Data Acknowledgement Packet
3. Routing Request Packet
4. Routing Acknowledgement Packet.

1. Data Packet

Each data packet has its own integer ID, and memory of its flow, transmission time, and whether it has been acknowledged or is in-transit. It fits in 8 kbits. The ID is not unique in the network, as on timeouts, packets are regenerated even though the are not guaranteed lost.

2. Data Acknowledgment Packet

Once a sink receives the packet, it will send an acknowledgement back preserving the packet's original transmission time. Each Ack has a size of 0.512 kbits, and a sink does not check whether the packet has been already acknowledged or not; this is to protect against Ack losses.

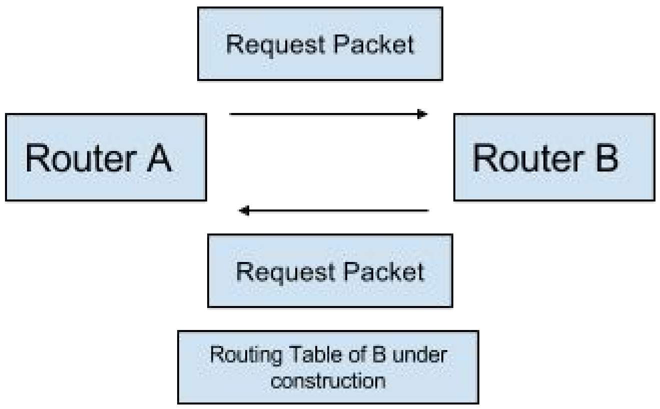
This allows detection of packets sent before and after timeout events, for example, and allows TCP algorithms to choose whether they update congestion estimates or acknowledgement tables depending on whether the packet is indicative of present or past congestion.

3. Routing Request Packet

Routing request packets are relatively small at 0.064 kbits. This packet originates from a router when it is ready to perform dynamic routing. When a different router receives the request, they will respond with a Routing Acknowledgment. Requests stop being sent when a router feels it has converged to a new routing table.

4. Routing Acknowledgement Packet

A routing acknowledgment is 1 kbit. This packet contains a routing table and a link cost for the link used to travel. The diagram below shows how the routing packet system works.



The link cost is dynamically estimated by links, which keep track of how long it takes for an enqueued packet to get transmitted into their channel from their buffer. We sum the queuing delay on either side of the link, estimated as a windowed average of observed delays. The full link cost is then the propagation delay (telepathically known) plus the dynamic queuing estimate. This last is not telepathy, as in reality it's possible for both sides of a link to monitor and communicate their queuing delays.

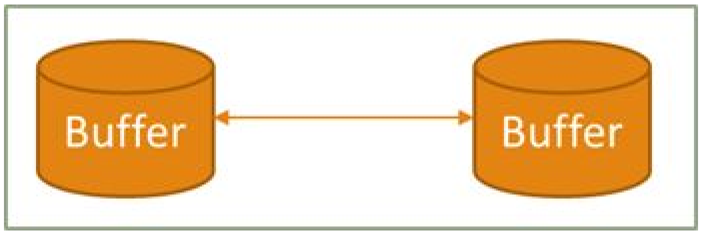
Each packet also has a timeout. As soon as a packet sending event has been enqueued in event queue, the simulator also enqueues another event along with it: timeout packet. Timeout time is set differently depending on the TCP algorithm of the flow. If a packet acknowledgment has been received already, the packet will no longer be able to timeout. When a timeout event occurs, the flow will handle it differently depending on the TCP protocol.

Host

Upon being called by the flow source, the host sends a packet over the link it is connected to. When a host receives a data packet, it calls its flow sink to respond. When a host receives a data acknowledgment, it passes it along to its flow source. The flow source then decides what to send next. So, for flows, hosts are wrappers.

The reason hosts exist as a separate entity was originally for routing. We thought routers would communicate with hosts via routing req/acks. This is necessary for a dynamic network, but since ours in a static network, we modified routing so routers have global information about hosts in the network, and just made host-router links have 0 weight, since there is no choice to be made about how to route.

Links

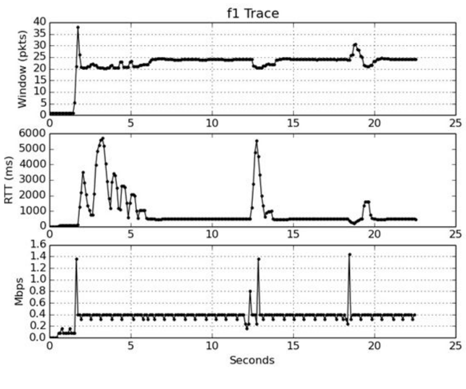


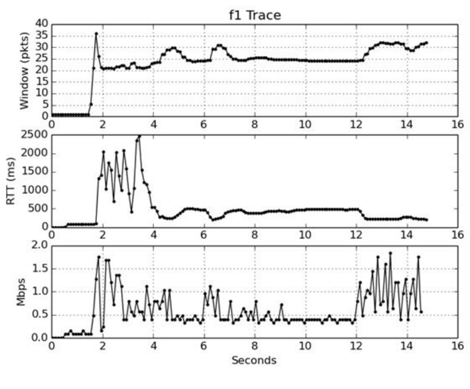
Our links are half duplexes, meaning that information can only flow one way at a time. Links contain packet buffers which in reality are in the nodes (hosts, routers) on either side. There are 2 buffers, one on each side of the link. When packets are enqueued we associate them with the current global time.

Links have a fair amount of telepathy, but that doesn't mean their action is not practical. In a half-duplex you're going to have collisions, and then each side will wait a random amount of time before re-transmitting. This introduces feedback delay which could destabilize our system, but if network nodes (hosts, routers) handle collisions reasonably, we expect the gain in simplicity by pretending we have a perfect stoplight to outweigh the cost of modeling error.

Our original *link* was a pure stoplight. It required packets to be able to reach their destination before anything on the other side of the buffer, before it would transmit then. So it switched directions a lot. This meant we lost throughput. In the worst case, as an example, consider a 10Mbps link with a 10ms delay, that switches direction every data packet. This has a maximum throughput of about 8kbit/10ms or ~1Mbps. So right there we've lost about 90% of our link capacity.

We then switched to a train link, which transmits packets so long as their enqueue timestamp beats that at the head of the other buffer. This allows packet trains, which decreases the effect of propagation delay. We find, for the most part, no difference, but the ability to send packets in bursts allows local peaking up to 2-3 Mbps that increases net throughput. This is shown below.





Focus on the bottom plot of the figure on the left. This is the TCP FAST flow rate for Test Case 1 with a stoplight link. Notice that we stabilize at a flow rate of 0.4 Mbps with little fluctuation. Now consider the plot on the right. Packet trains allow peak flow rates of up to 2 Mbps!

It should be noted that links give routing packets priority. They displace data packets and are enqueued at the head of their buffer. Their timestamp is still current, though, so the arrival of routing packets usually results in packet trains from the other side of the link (as all previously enqueued packets arrived earlier in time). You can think of this as a router that doesn't want to deal with congestion control for its routing updates, but that still respects right of way from external packet sources. This scheme ensures routing packets are not dropped. It speeds up the converging of dynamic routing. Unfortunately it also has a major effect on TCP; RENO and VEGAs, as you'll see later, care a *lot* about their dropped packets. TCP FAST merely blips.

Router

Destination	Link
H1	L4
H2	L1
H3	L1

A Router's job is mostly to route a data packet to the correct link. However, occasionally it will also do dynamic routing. The dynamic routing algorithm we use is Bellman-Ford. We separate the current routing table from the one that is under construction. When a router request is sent to router A, router A will respond with a router acknowledgment packet which includes its routing table under construction. The router acknowledgment packet also includes a cost of the link. The cost of the link is the average waiting time for a data packet in the link for a certain duration. Once router B receives the router acknowledgment, it will update its routing table under construction with the addition of the cost. Router B will them send another request to Router A. This will continue until the tables converge. Convergence is reached when the routing acknowledgment packets does not change the routing table under construction equal to the number of links the router is connected to + its destination. Once convergence is reached, the router replaces its old routing table with the new one. Since this is the case, the routers can still send data packets at the same time that dynamic routing is happening. There is some telepathy involved here since all the routers starts dynamic routing at the same time. We are not worried about router packets getting dropped because it has priority in the link buffer.

Flows

Flow Source

The flow source is called to send data packets over and is called again when it receives a data acknowledgement packet. Depending on the TCP algorithm, it will decide which packet to next send over and whether to increase or decrease its window size.

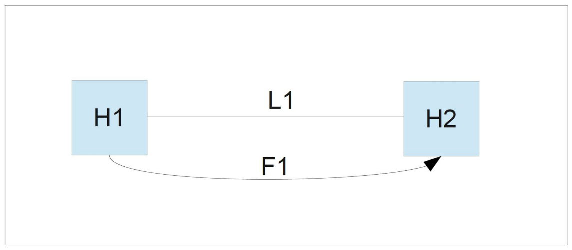
It's easiest to describe how we implemented flows with traces on hand, so a detailed description of RENO, VEGAS, and FAST flow operation follows in *Test Cases and Analysis,* below.

Flow Sink

When the flow sink receives a data packet, it will send back a data acknowledgement packet with the last packet ID it received in a linear sequence from 0. So a sink receiving packets 0,1,...,10, 11, 13, 14 will acknowledge 0,...10, 11, 11, 11. All TCP implementations use the same flow sink.

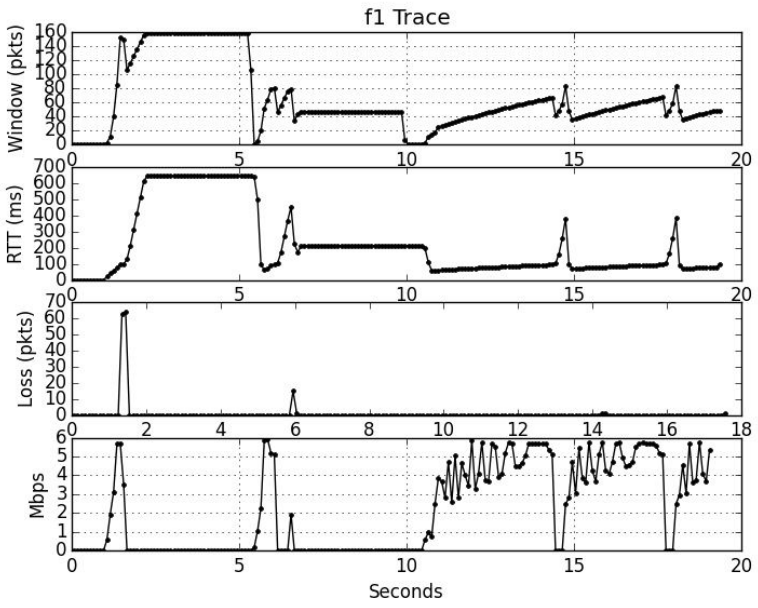
Test Cases and Analysis

Test Case 0



This test case is supposed to send 20 MB from H1 to H2 starting at 0.5 seconds. Below, we've simulated just far enough to demonstrate pertinent effects of the three TCPs we considered.

TCP RENO



First, let's demonstrate that RENO is acting correctly. Link buffers aren't too critical here to understand what's going on so their statistics are omitted.

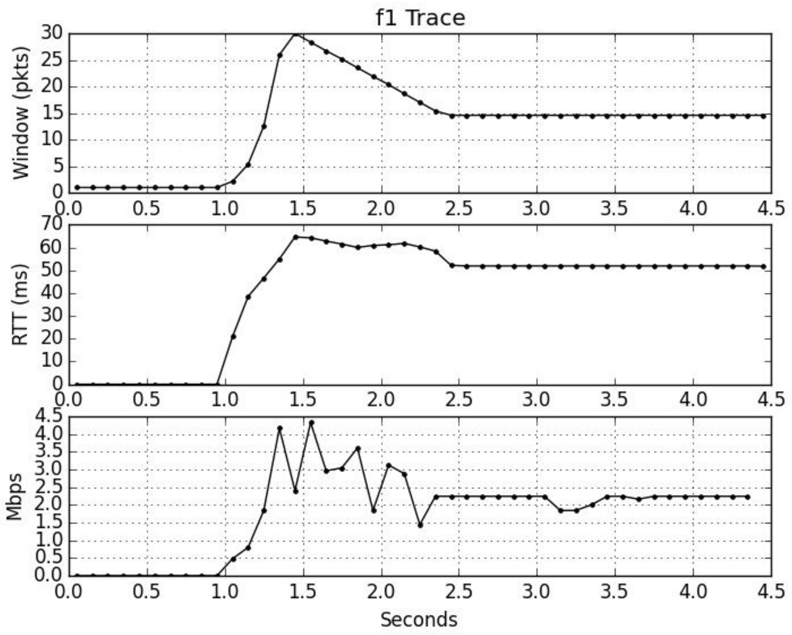
Initially, we're in slow start, so we ramp exponentially. At about 2 seconds, we see our first packet losses, as detected in the links, but we don't realize we've lost packets for a full timeout window afterward, defined in this particular simulation as 4 seconds. So initially, when we start seeing half our packets drop and get some triple acks, we halve our window size and go into FR/FT. Of course, we still have plenty of packets in the network, and many of them are showing triple acks for the same values, since it takes time for our FR/FT packets to reach the sink and move the acknowledgements forward. So for a while we're updating our FR/FT window based on packets we transmitted before entering FR/FT. However, it appears any packet we send is almost immediately dropped, for a while at least, because around 2.5 seconds we see our window size stop changing. Look at the RTT - it's constant. The only way that could happen is if we stopped receiving *any* acks.

Now, RENO was implemented with a fixed timeout window. This is a bug, and was fixed in Vegas and FAST. For RENO, though, after 4 seconds, we recognize the timeouts, which all happen within a few 100ms of each other, since we sent them out in an exponential ramp. So we quickly exit the timeout recovery window After our window size drops to 1, we re-enter slow start, but with a halved congestion avoidance threshold. Notice though we have many timeouts, we only halve our window once.

Very quickly we re-enter FR/FT, and this time we make it through two FR/FT phases before a second timeout. The third time we exit slow start straight into congestion avoidance, with a slow ramp, and thereafter we see only FR/FT cycles, as we've sized our timeouts to allow FR/FT a chance to recover properly.

Overall we're seeing an average of about 4 Mbps with steady state RTTs of about 100 ms, or 5 minimal RTT propagation delays.

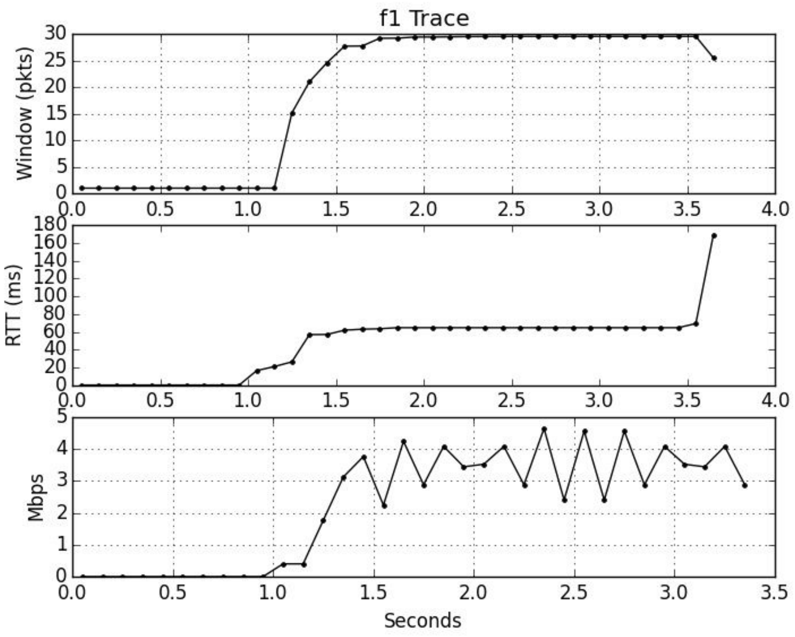
TCP Vegas



RENO gave us an idea for calibrating TCP Vegas. Vegas is slightly different than RENO in that from slow-start, it switches to congestion avoidance when its estimated current RTT is some multiple of the minimum RTT observed. We set this threshold to 5 for the plot above. Notice there are no packet losses this time, and no timeouts. We stay in congestion avoidance, approaching steady stay with a linear behavior characteristic of the Vegas W += 1/W approach.

The specific parameters for the simulation are unimportant, but note Vegas is also both a drop- and queuing- based algorithm in that it responds to timeouts harshly, and has FR/FT very similar to RENO, but updates window sizing in CA based on observed RTTs. We picked our CA parameters to be about 0.4, so our ratio of propagation delay (20 ms) to queuing delay (30ms, with steady-state RTTs in the 50 ms range) yields a target rate of about 0.4 * ⅔ ~ 0.27 the maximum link capacity, or 2.7 Mbps, as observed.

TCP FAST



FAST has no timeout recovery, no fast-transmit/fast-recover. It's the simplest and most robust control system we tested. It updates its window size at a fixed clock (in the case above, every 220ms), and we estimate our RTT with a 15 packet window, linearly averaged. We attempted exponential averaging with a 'time" constant of 10 packets (exp(-window_index/10)) but found this just helped TCP FAST forget the consequences of larger windows even sooner, which led to more oscillation.

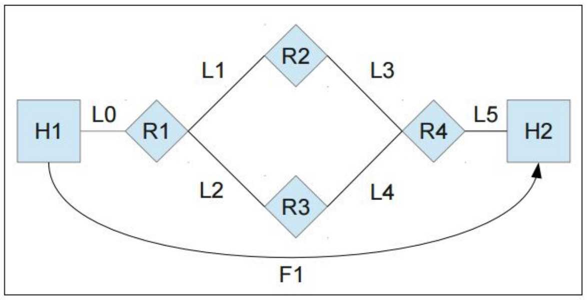
So for the first few Acks, we're just building our estimate of the actual RTT in the network. In this case, the first packet we send sets the minimum observed RTT, since the network is entirely uncongested there. Then, we update our window at 220 ms based on our observed actual RTT (which rises slightly) and our TCP constant *FAST_ALPHA* which was 20. The way to think about this parameter is, if our estimate of RTT (actual) is way larger than the observed minimum RTT (that we've ever seen) then we're probably congesting the network, so our window size will stay close to *FAST_ALPHA,* which is like a minimum window size regardless of network congestion.

Considering where RENO stabilized for window size (around 30-40 packets) we could have made this parameter larger, but we were conservative, so we're more stable than RENO, but slower.

We note no timeouts (because we were conservative), but keep this in mind for the next few test cases: had there been timeouts, or triple acks, we would have assumed these packets dropped, and penalized these packets with an estimated RTT scaled off our maximum observed RTT. In this case it was 3x. For timeouts, we set a very long (1s) base RTT to wait (for when we have no idea what the network congestion is) but after we've seen even one packet, we allow 3x the current observed RTT as a timeout allowance.

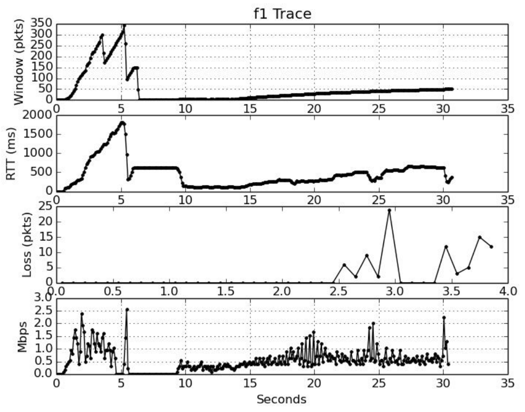
In case you were wondering, the blip in RTT at the end of this simulation happens for many flow sizes, because the last packet we send often sees a train of Acks blocking the link for a while.

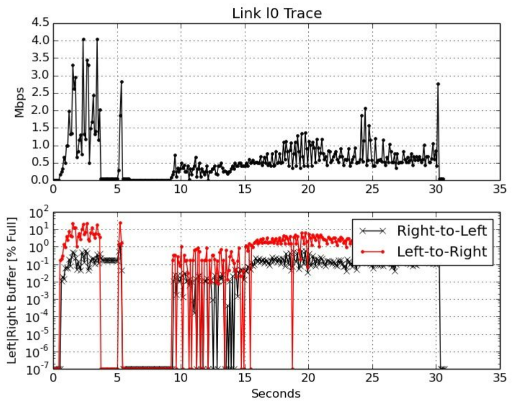
Test Case 1

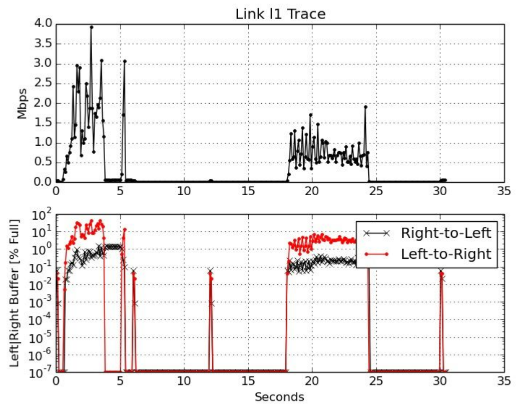


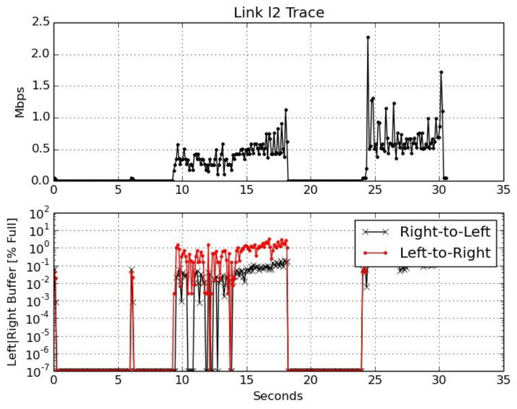
This test case has two hosts, four routers, 6 links, and was simulated below with a 5 Mb flow since the relevant dynamic routing & TCP behavior are all evident by then.

TCP RENO









TCP Reno starts with a slow start and ramps up. After a while, the window size is too large and the flow gets a triple ack. Once that happens, the flow goes into fast retransmit. The window size drops by half and once we've received enough Acks to drop below this threshold, our flow starts to send more packet. However, it then get more triple acks, before timing out. Notice there *are* packet losses early one (check the timescale of the loss plot), so the timeout is not entirely ‘fake,’ in that the some packets were lost, and other were still in the network.

Unfortunately, we were conservative in our design, so on timeout, we wait for *every* packet in our current window to time out, too. We handle them if they come back (in terms of what we send next), but we don’t update our window size based on them. We only update window size based on packets that were sent *after* the timeout event.

So, once there is a time out, the window size drops to 1 and we don’t re-enter slow start till about 200 packets timeout. This takes exactly 1 timeout window (fixed for RENO, remember), or about 4 seconds. Then we do a slow start.

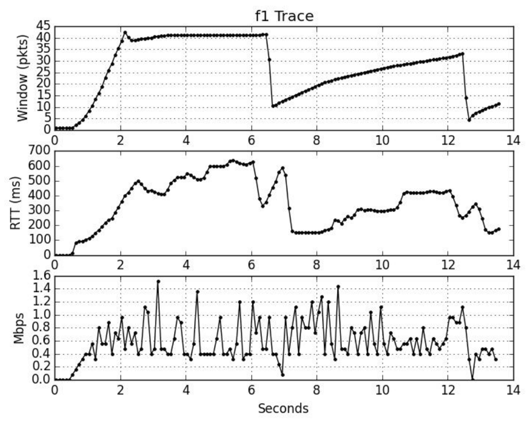
Once we’re in the slow start phase, we’re sending out packets increasing window size exponentially, but remember, *not* every packet timed out. So every once in awhile we get triple acks due to packets from the previous cycle that made it through, and we enter FR/FT halving our window size. So even though looking at links the network is uncongested, and our RTTs are low, we keep our window size low, because we keep entering/exiting FR/FT.

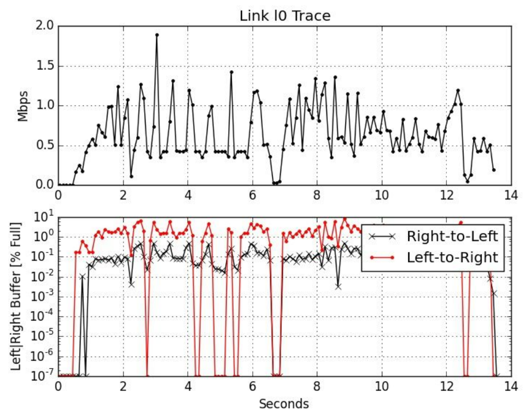
Eventually we get out of this trap, and enter CA, and by the time the simulation ends, we’ve been ramping linearly for a while.

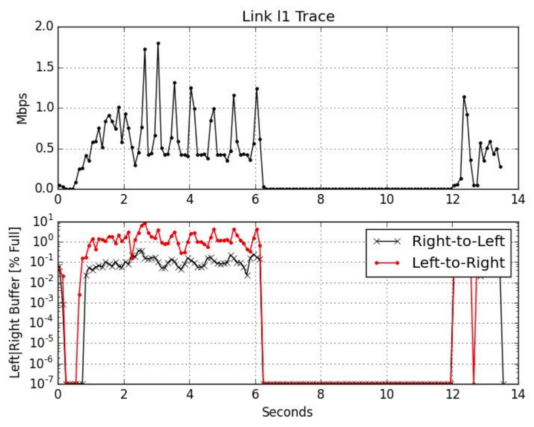
It is important to note that dynamic routing is doing as we expected. Our routing updates every 6 seconds, so the first update overlaps with a timeout recovery phase which means even though the tables update, we see very little traffic.

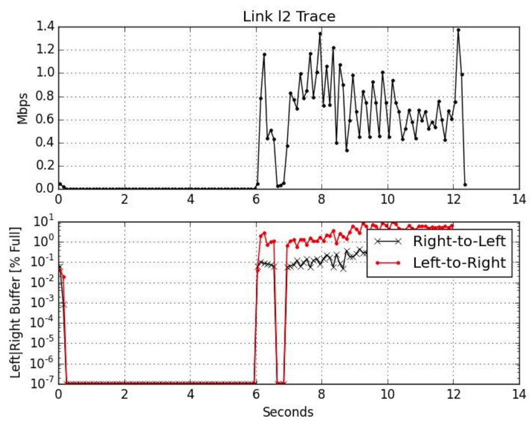
In the first routing, H1->R1->R2->R4->H2. This is because there is no initial traffic. However, in its second routing, there’s a huge traffic in R1->R2->R4. Hence the system routes it through R1->R3->R4 instead. On its third round, the opposite happens and routs through R2 again. These results are shown in the links. After the first dynamic routing, L1 was being used. But after the second dynamic routing, L2 is being used instead. This oscillation in routing corresponds to our analytical results. In the second case, the links from R2 have huge costs, while the links near R3 has 0 cost. The opposite happens in the third case.

TCP Vegas







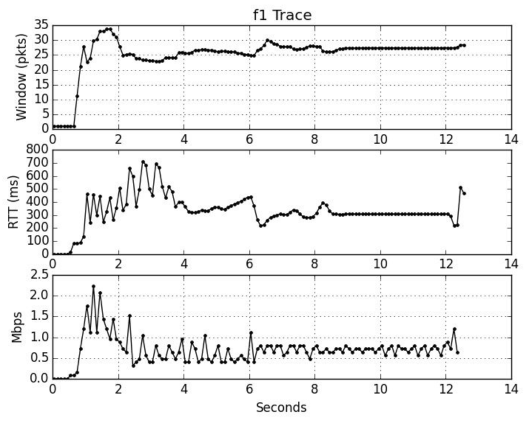


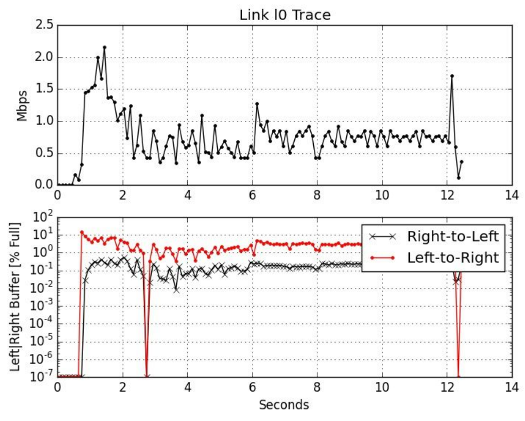
Vegas is really interesting in terms of the interplay between dynamic routing and congestion control. Notice there are no packet losses and no timeouts (because we implemented Vegas with timeout recovery too, and the window size never hits 1 and *stays* there, like in RENO timeouts).

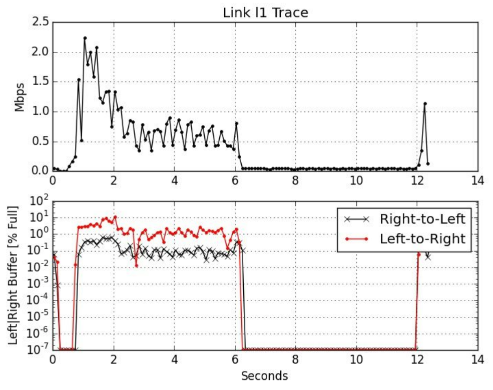
So what's happening here is, every 6 seconds, we get a routing update, which (see the links) changes between the top and bottom paths in the router bridge. So every 6 seconds, we FR/FT, because we've got a ton of packets in one path, but are now sending over a new path, so our latest (further out) packets arrive *before* packets on the other half of the router bridge*!* This works itself out very quickly, but it takes long enough that we do enter FR/FT. Afterward, we go into CA and stay there till the next routing update.

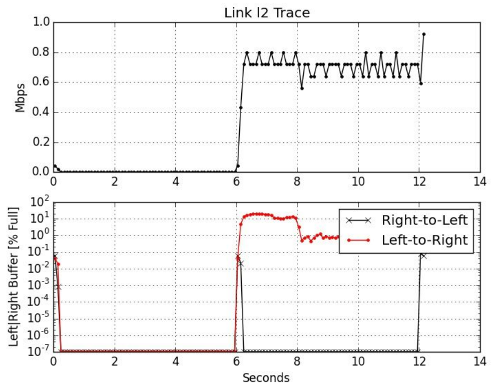
The lesson here is that dynamic routing that oscillates can really screw with some congestion control algorithms!

TCP FAST







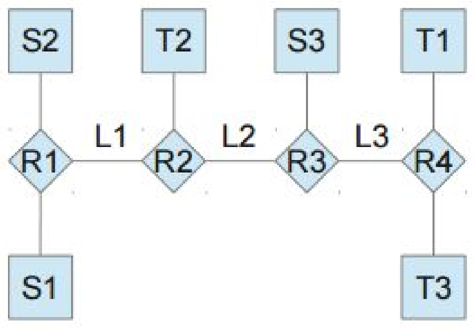


FAST is once again the most stable of the lot, largely because of our conservative parameterization. We see the same blip due to routing in flow rate, with the blip upward because routing allows for packet trains, as discussed in *Links.* This damps out quickly. We see two routing cycles, and that the second (L2-L4) path is a bit faster, likely because we don't have the same startup window overshoot that the first path saw (this depends on an interplay between our window update period and feedback delay in the network).

The real strangeness here is that the L2 trace shows *0* right-to-left traffic. If you look at *~/results/Test Case 1 FAST,* you'll notice that L3 maintains right-to-left traffic through the entire simulation. So it turns out that with our dynamic link costs as they were, R1 decided to route through the bottom path (L2-L4), but R4 decided to route through the top path (L1-L3). This suggests a bug in our routing. In any case, the results are interesting, so let's analyze them.

With this routing structure, data flows L0-L2-L4-L5, and Acks flow L5-L3-L1-L0. Because of how measurements are plotted for Link Buffer Occupancy (a "level" quanity, time-averaged), seeing a packet enter an empty link buffer shows up as two measurements at the same time (an arrival, and an immediate "removal" from the buffer as the packet is being staged for transmission). So, we see 0 occupancy, because the packet is in the buffer for 0 time before being removed for transmission. Our L2 looks like it sees no right-to-left traffic in the second routing update cycle because it really sees no traffic there. Our L1 shows traffic in MBPS a hair above zero in this window, so *it* is seeing the Ack traffic. We haven't shown L3 and L4, but they bear out this reasoning. L3 sees constant Ack traffic throughout, and L4 shows virtually 0 traffic because any Data from L2 immediately gets sent over (in a train) since there are no Acks blocking link flow on the other side of L4.

Test Case 2

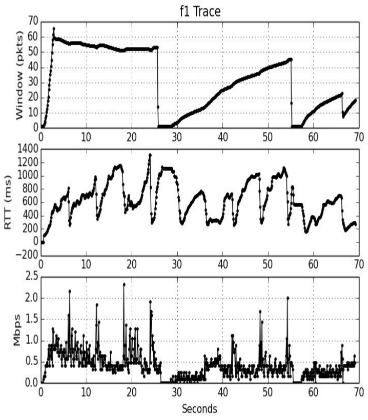


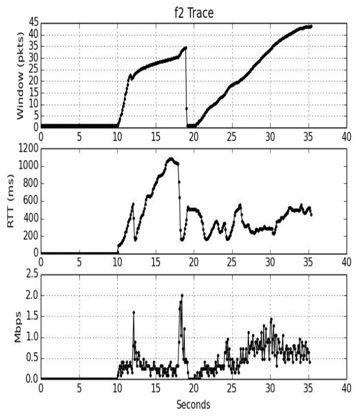
This test case has no routing issues, except for packet-train blips as we've seen before, so it's a perfect place to see what FAST can really do when tuned for throughput. S1 has a flow sending 3.5 Mb to T1 starting at 0.5s. S2 sends 1 Mb to T2 at 10s, and S3 sends 3.5Mb to T3 at 20s.

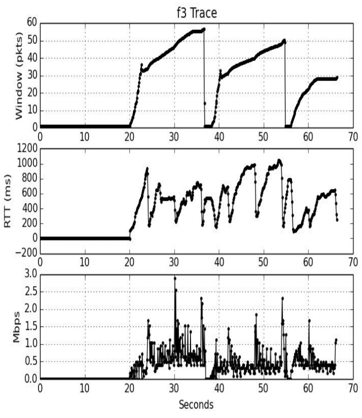
TCP RENO

RENO is not even plotted for this test case because it is so terrible. Normally RENO takes some time to recover from its own slow-start foolishness, but in this case that time overlaps with the other two flows starting up, in their own foolish slow start phases, making our network RTTs so slow as to *never* let any of the flows out of the earliest part of CA. Simulating larger flows would show us the same cyclic CA-FR/FT behavior as we've seen before, but who has the time to wait? Our simulator is *very* slow, in part because of the sheer number of things we measure for debugging & plotting.

TCP Vegas







Careful with the time axis domains on this one. We see timeouts and recovery in flow 1 and 2 when flow 3 starts up, likely because flow 2 doesn't "see" flow 3 in its RTT sampling path, and flow 1 has major delay before it sees flow 3.

In general, we already know VEGAS doesn't update WS fast enough to recover gracefully from large perturbations. Notice that we're definitely timing out in flows 1, 2 but that there are no actual packet losses. We've parameterized conservatively, but that doesn't stop the sudden increase in actual RTT (versus our feedback-delay limited update of estimated RTT) from destabilizing us, causing timeouts based on *old* estimates of 3x RTT. We see this keep happening; we never really enter a cyclic FR/FT - CA cycle as in Test Case 1. We're always timing out and recovering.

We also see that flow 2 and 3 have an unfair advantage over flow 1, reaching larger window sizes out of slow start because they see a congested network on startup as they calculate their RTT minimum. Notice neither flow 2 nor 3 ever sees an RTT < 200ms, whereas flow 1 on startup has seen one with RTT ~ 80 ms.

TCP FAST

Mathematical Analysis

In Test Case 2 we have routers R1-R4, links L1-L3, source hosts S1-S3, and corresponding destination hosts T1-T3. The link capacities of L1, L2 and L3 are $c = \frac{1}{\frac{8192\ bit/packet}{10 Mbit/sec}}$ = 1280 packets/s.

The one way propagation delay of each link L1 - L3 is 10ms

There are three flows that use TCP FAST with $\alpha = 20$:

- F1 from S1 to T1 starts at t=0.5s
- F2 from S2 to T2 starts at t=10s
- F3 from S3 to T3 starts at t=20sec

For TCP FAST, window update is $W(t+1) = \frac{RTT_{min}}{RTT} W(t) + \alpha$

→ $\dot{w}_i(t) \approx W(t+1) - W(t) = \left(\frac{RTT_{min}}{RTT} - 1\right) W(t) + \alpha$

FAST is modeled by $\dot{w}_i(t) = \alpha_i - \frac{q_i(t) w_i(t)}{d_i + q_i(t)}$ $\quad \dot{p}_l = \left[\frac{y_l(t)}{c_l} - 1\right]^+_{p_l(t)}$ $\quad x_i(t) = \frac{w_i(t)}{d_i + q_i(t)}$

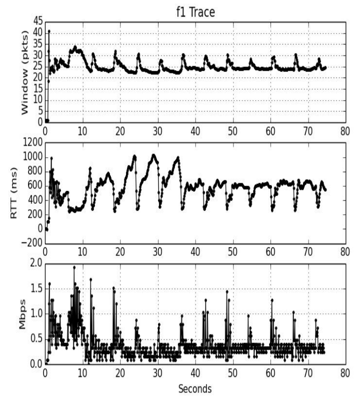
Before t=10s, there is only F1 in the network. It will use up all the capacity of links L1-L3

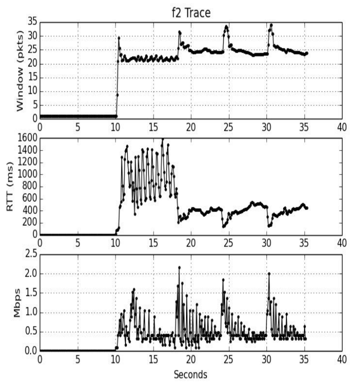
$x_i = 1280\ packets/s$ The queueing delay is $q_i = \frac{\alpha}{x_i} = \frac{20}{1280\ packets/s} = 15.6\ ms$

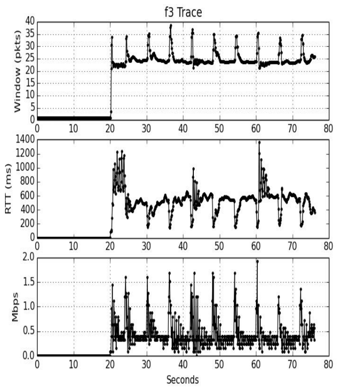
The queue length of F1, which is only on link L1, is 20 packets. The queue lengths on L2 and L3 are 0.

Between 10s-20s, F1 and F2 share link L1, the bottleneck. There are no queues on L2 and L3, and hence p2 = p3 = 0. F2 knows it $RTT_{\min,2}$ as $d_2 + 15.6\ ms$, and hence $x_2 = \frac{\alpha}{q_2 - 0.0156} = \frac{\alpha}{p_1 - 0.0156}$ and $x_1 = \frac{\alpha}{p_1}$.
Solving $x_1 + x_2 = \frac{20}{p_1 - 0.0156} + \frac{20}{p_1} = 1280$ gives $p_1 = 40.9\ ms$
The flow rates $x_1 = \frac{20}{0.04089} = 409\ packets/s$, $x_2 = \frac{20}{0.04089 - 0.0156} = 600\ packets/s$
The queue length on L1 is $1280 \times\ 40.9\ ms \approx 63\ packets$
After 20s, F1 and F2 share link L1 and F1 and F3 share link L3, two bottleneck links. There are no queues on L2, and hence p2 = 0. F3 knows it $RTT_{\min,3}$ as d_3, and hence $x_3 = \frac{\alpha}{p_3}$
Solving $x_1 + x_2 = \frac{20}{p_1 + p_3} + \frac{20}{p_1 - 0.0156} = 1280$ and $x_1 + x_3 = \frac{20}{p_1 + p_3} + \frac{20}{p_3} = 1280$ gives $p_1 = 36.9\ ms$ and $p_3 = 21.3\ ms$
The flow rates $x_1 = \frac{20}{p_1 + p_3} = 344\ packets/s$, $x_2 = x_3 = \frac{20}{p_3} = 939\ packets/s$
The queue length on L1 is $1280 \times\ 36.9\ ms \approx 47\ packets$ and on L3 is $1280 \times\ 21.3\ ms \approx 27\ packets$

Simulation Analysis

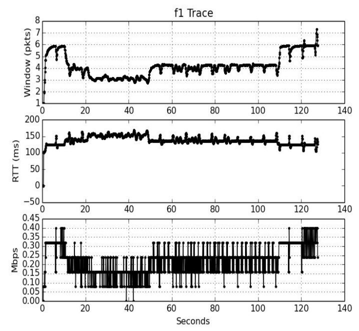


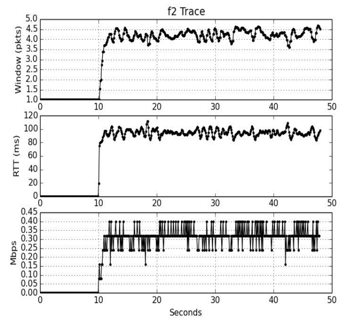


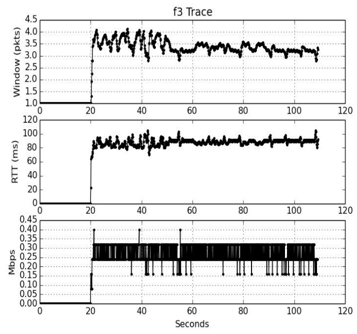


Using the same parameters as in the previous test cases, we see the traces above. Let us start by noticing that ~350 packets/s is ~350kB/s or ~2.8Mbps, and ~900 packets/s is likewise ~900kB/s or ~7.2Mbps. We get nothing close to that, since our links (even with trains) are apparently not up to snuff. Ignore the blips in all traces due to routing (6 second period).

Looking at our window sizes, we see they all stabilize to 25 packets, with roughly equal RTT. Our throughput is likewise similar, though flows 2 and 3 are a tad higher and peak higher, and more often. This suggests our *FAST_ALPHA* is too large, and that we're almost in open-loop with fixed window sizes. In order for all FASTs to operate in closed loop, we found we needed this parameter around 1.0.







This time, we see that flow 2 indeed doesn't care about flow 1 or 3, and that flow 3 does notice flow 2, but only through oscillations in flow 1's output, and hence with significant delay and thus higher amplitude oscillation. Flow 1 ramps in steps as flow 2, 3 start and finish. All have truly terrible throughput, of course, and keep their window sizes negligible, as the RTT quickly reaches the point that we are forced to stay near *FAST_ALPHA.* Flows 2 and 3 see an unfair advantage over flow 1 simply because they start later, as with Vegas.

Project Process

Overview

During the design phase, it was particularly important for us to meet in person at least twice a week for 2-3 hours. This was to make sure that we agreed on a design and could work independently. When team coding and debugging the router & various TCPs, we found group meetings and TA meetings critical.

With TCP, we found that our first revision was messy and needed to be refactored; still, getting it started even when we weren't quite sure what was going on was critical to wrapping up on time.

In retrospect, we ought to have started TCP Fast *first.* It is by far the most interesting, the most visual, and the most like other control systems we've already played with. For example, there are clear mappings between most Fast parameters and things like feedback-delay (timeout scaled off RTT estimated), biasing towards open- and closed- loop responses (FAST_ALPHA), and it is visually obvious how to tune things by looking at overshoot and so on. Finding bugs in Reno and Vegas was much harder, required poring through debugging data dumps, and was actually fairly demotivating as we progressed into the fourth straight week of bugs! Yet, the moment we saw interesting traces, our spirits always jumped again. Long story short, start with Fast.

Plotting is very, very important for debugging but we'd advise next year's teams that most plots useful for debugging are very different (more event based, less averaged) from the plots required.