COMPSCI 742 S2 C Assignment 1

Department of Computer Science

The University of Auckland

Due Sunday 21 August 05, 11:59 pm

This assignment will contribute 1/3 of your coursework mark, and 10% to your overall course mark.

Submit your assignment via the DropBox, either in PDF (preferred), or in MS Word format. Assignments in other format will *not* be accepted or marked!

The problem: testing available capacity

Every link along an Internet path has a 'bottleneck' link, i.e. one that sets an upper limit on the data rate between sender and receiver. For example, we often (used to) have campus networks using 100 Mb/s Ethernet, connected via wide-area links using 10 Mb/s Ethernet. In this case we call the link with the smallest carrying capacity the *narrow* link.

If the bottleneck link is carrying other traffic, we describe that traffic as *cross* traffic; it will use some of the link's capacity. We call the remaining link capacity its *available* capacity. We call a link which becomes a bottleneck because it carries cross traffic a *tight* link.

Researchers who have large (Terabyte) files to transfer often want to know the available capacity for the path between two sites, so that they can estimate how long it will take to transfer a file. One way to measure available capacity is to transfer a large test file, and observe the transfer rate for that file. If that file transfer uses all of the available capacity, its transfer rate will give us a good estimate of available capacity. Otherwise, we can run multiple transfers until we do use all the available capacity. *Iperf* is a performance measurement tool that uses this approach.

Now consider how TCP behaves. A single TCP connection will increase its congestion window until it senses a packet loss; that packet loss will usually be caused by lack of space in router buffers. After that TCP will try to keep its sending rate (congestion window) close to the maximum rate. Therefore, a single TCP connection should be able to use all the bandwidth, so there's no reason to run multiple TCP transfers.

Again, if we do run multiple TCP transfers, they should – after a while – adjust their congestion windows so that they share the available bandwidth; that's what we mean when we say that TCP is *network friendly.*

For this assignment, you are to use *ns* to explore the behaviour of multiple TCP connections on a bottleneck link.

Start by downloading the tcl scripts from the 742 'Assignments' web page. You can run *ns* on `hydra.cs.auckland.ac.nz`, as explained in the lecture handout on *ns*.

CONTINUED

Note that *ns* provides a simplified, idealised version of TCP. In particular, Agent/TCP/Reno:

- Doesn't send the opening SYN or SYN ACK packets
- Always sends the same size packets
- Uses *packets* as its units for sequence numbers, and for it's congestion window
- Only has packet losses for the data packets being sent. They don't occur for the returning ACK packets

Nonetheless, *ns's* TCP is realistic enough to clearly demonstrate how TCP behaves in most circumstances.

1. **Familiarise yourself with** `ns` [5 marks]
   For this you'll use the `ack_clock.tcl` script – see the notes on 'Sliding Window' at the end of this assignment handout.

   (a) Run `ack_clock.tcl` using the parameter values for Animation 1 in the 'Sliding Window' notes. What does `nam`'s 're-layout' button do? What happens if you increase the bottleneck link speed to 0.5 Mb/s? [2 marks]

   > Re-layout button draws network using 'ball and spring' algorithm. [C_a specifies attraction in springs (links), C_r specifies repulsion between balls (nodes)] [1 mark]
   > Higher link speed allows higher throughput, smaller queue sizes. [1 mark]

   (b) Run `ack_clock.tcl` again, using the parameter values for Animation 2. Why can't you see (on the `nam` display) any packets on the link between nodes 1 and 2? [1 mark]

   > The link speed (155 Mb/s) is now much higher than the edge links, hence the packet size is too small to be visible. You can see them if you slow the animation, e.g. by setting Step: to about $10\mu s$. [1 mark]

   (c) Run `ack_clock.tcl` again, using the parameter values for Animation 3. What does `nam`'s 'Edit/View' button do? How does 'Edit/View' interact with 'Layout' [2 marks]

   > Pressing Edit/View switches to 'edit' mode, allowing you to manually edit the layout, i.e. move nodes with the mouse. Clicking the button again switches back to 'view' mode. [1 mark]
   > Re-layout recomputes layout whenever you press it. [1 mark]
   > *In View mode you can click on links to produce plots of packets vs time in small strips below the main display. Also, you can click on a node to specify filters, i.e. whether packets with that node as source or destination are displayed.*

CONTINUED

2. **Understand the `a1-net.tcl` script** [5 marks]

   (a) The script sets the following values: run_nam = 1, n_tcps = 3, sim_time = 10, q_size=9. Run ns, look at the nam display, adjust its layout so that the network structure becomes clear, then start it running. What does it do? [3 marks]

   > Sets up a link from R1 to R2, and *n_tcp* TCP links from s(j) to d(j) via R1-R2. [1 mark]
   > Starts first TCP session at 0.1s, and the other TCP streams at 0.5s intervals, after that. FTP data sources send through each TCP. [1 mark]
   > A record is written to file *df_name* every *interval* seconds; it contains the time (seconds), and the ack_ value for each of the TCPs. [1 mark]

   (b) The script starts off with a set of parameter values – you can edit the script to change them. What do the run_nam, n_tcps, sim_time, q_size parameters do? [2 marks]

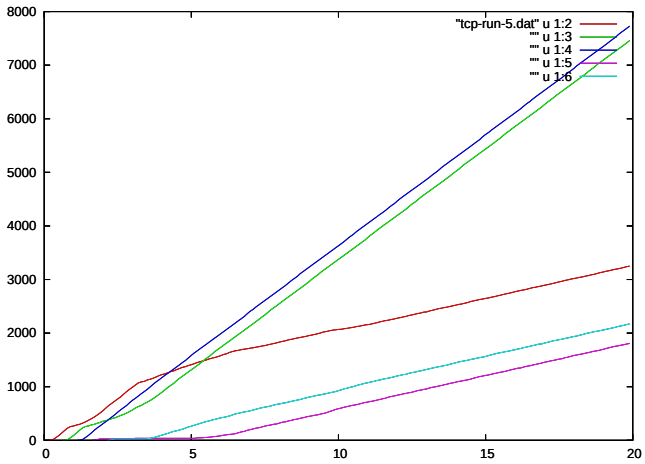
   > *run_nam* = 1 to invoke nam at end of ns run
   > *n_tcps* = number of TCP streams to use
   > *sim_time* = number of seconds simulated time
   > *q_size* = queue size for R1-R2 link (i.e. at R1 node)

3. **Running multiple TCP connections** [5 marks]

(a) Edit the script so as to prevent it generating an `out.nam` file (the file is too big to keep playing with, and we no longer need to see how the simulation is proceeding).

Set n_tcps=1, 2, 5, 10; run for each to get tcp-run-n.dat files. Make plots of the 5 TCPs in tcp-run-5.dat (using gnuplot, excel, etc.) Make a table showing total number of packets transferred for 1, 2, 5, 10 running TCPs after 20s.

Plot of packets transferred vs time, i.e. the data values from the tcp-run=5.dat file; should look something like this –



If you plot the whole 20s, you can see that the plot becomes stable, i.e. the TCPs reach a balance against each other after about 7s. [That generally seems to happen with two or three of them running at about the same rate, and the others running at a slower rate. *Why don't they all settle at the* same *rate?*] [2 marks]

Total number of packets transferred: 8710, 17010, 22568, 20420 for 1, 2, 5 and 10 TCPs. [1 mark]*(+/- about 60 is OK, provided the relative values are reasonable)*

Comments on the traces: [2 marks]*(1 mark for each sensible comment on the plots)*

4. **Plot the total throughput for *n* TCP connections** [5 marks]

   (a) Modify the script to compute total throughput. Verify that the totals are computed and written to the data file correctly. Plot the total number of bytes vs time when 1, 2, 5, 10 TCPs are running.

   Modifying the script: need to add code to the `w_data proc` to sum the packet counts for the individual TCPs, and write it to the output data file. [2 marks]*(1 for the code, 1 for demonstrating that the sums are correct, e.g. by printing a few lines of the file with total and individual counts on each line. Possible 1 bonus mark for ignoring -1 counts for not-yet-started TCPs!)*

   Plot should show that throughput increases for 1, 2, then 5 TCPs, but 10 TCPs have lower throughput than 5. [1 mark]*(Throughput can be in packets or bytes* $(= packets \times 1000)$*)*

   Comment on the traces: [2 marks]*(1 mark for each comment, max 2. 1 bonus mark for plots showing transfer rates for the individual TCPs [but total mark is limited to 20!])*

5. **(OPTIONAL) Improve the simulation** [0 marks]

   (a) Can you find a set of ns parameters that allows a single TCP to build up to almost completely filling the bottleneck link? If you can, how does it behave as the number of TCP connections increases?

   One can achieve this in several ways:

   - Decrease the link speed – e.g. to around 2 Mb/s
   - Increase the R1 queue size – e.g. q_size = 60, also increase the transmit window size (window_ ?) to match.
     This is more like a real-world approach, where we could tune the parameters of our TCP stack.

Sliding Window: ack_clock.tcl

`http://www.isi.edu/cgi-bin/ns-edu/view_db.pl#ack_clock.tcl`
By F.Cela & A.Goller on 02/16/01
`http://www.ce.chalmers.se/~fcela/tcp-tour.html`

TCP implements an algorithm for flow control called Sliding Window; the reader will surely be familiar with this kind of algorithms which are used for flow control at the data link control layer of some protocols as well. The "window" is the maximum amount of data we can send without having to wait for ACKs. In summary, the operation of the algorithm is as follows:

- Transmit all the new segments in the window
- Wait for acknowledgement/s to come (several packets can be acknowledged in the same ACK)
- Slide the window to the indicated position and set the window size to the value advertised in the acknowledgement.

When we wait for an acknowledgement to a packet for some time and it has not arrived yet, the packet is retransmitted. When the acknowledgement arrives, it causes the window to be repositioned and the transmission continues from the packet following the one transmitted last.

TCP achieves following objectives by using the sliding window algorithm:

- The ACK policy makes the protocol self-clocking. Therefore, it dynamically adapts its transmission speed to both the speed of the network and the speed of the peer sending acknowledgements. If the conditions change in the network, so does the sender's transmission rate.
- The "credit" given by the window size makes possible an efficient use of the link.
- The receiver can regulate the information arriving rate by adjusting the sender's transmission window.

CONTINUED

Animation 1: shows the first two issues explained above.
script used: ack_clock.tcl

Parameters for this animation: `ns ack_clock.tcl 0.2Mb 40ms 100 20 2`

Description: Links n0-n1 and n2-n3 have 1Mbit/s bandwidth and 10ms delay. Link n1-n2 has 0.2Mbit/s bandwidth, 40ms delay. Queue size at n0 is 100 segments and TCP uses a 20-segment window size. For the sake of simplicity, the TCP in n0 models the first TCP specification which sends as much data as the send window allows at the beginning of the transmission. Current TCP implementations use a less aggressive start (slow-start). After the connection establishment, we start sending as much data as the send window allows us. Note how the queue grows fast at the beginning of the bottleneck. Around t=3.8 we have sent the first 20 segments that the window allows us, then we run out of window and we have to wait for the first acknowledgement to open the window again. From now on, arrival of each new acknowledgement triggers the transmission of a new segment, the TCP source adapts to the effective capacity of the path, and and we can see how the queue at the bottleneck remains stable.

Animation 2: shows self-clocking

Self-clocking is an interesting property of TCP that allows automatic adjustment of the transmission speed to the bandwidth and delay of the path. Therefore, it makes possible for TCP to operate over links with very different speeds. As an example, we can make the link 1-2 in the previous animation more than 700 times faster and TCP will still work without any problems.
script used: ack_clock.tcl

Parameters for this animation: `ns ack_clock.tcl 155Mb 2ms 100 20 1`

Description: Links n0-n1 and n2-n3 are 1Mbit/s bandwidth, 10ms delay. Link n1-n2 is 155Mbit/s, 2ms delay. Queue size at n0 is 100 segments and TCP uses a 20-segment window size.

Animation 3: shows TCP works well even if we make the bottleneck slower. However in this case we discover the effect of the "data in transit" limitation discussed in (2): as long as the window size is smaller than the actual size of the path we send bursts of data rather than a continuous flow. Therefore, we are not able to use the link efficiently.
script used: ack_clock.tcl

Parameters for this animation: *ns ack_clock.tcl 64kb 100ms 100 20 1*

Description: Links n0-n1 and n2-n3 are 1Mbit/s bandwidth, 10ms delay. Link n1-n2 is 64kbit/s, 100ms delay. Queue size at n0 is 100 segments and TCP uses a 20-segment window size. Around t=0.5 we run out of window and we have to wait for acknowledgements to open it again. We send bursts of packets rather than a continuous flow of packets.