Coursework 2

Fixed point, Drift models

Network Performance—DJW—2011/12

This coursework is worth 8% of your final grade. You should submit your answer by noon on 16 January, either by handing it in to Computer Science reception or by emailing it to `d.wischik@cs.ucl.ac.uk`*. If you email it, then please keep emailing it once per day until you receive an acknowledgement that I've received it.*

The TCP throughput formula says that the transmission rate of a flow is

$$x = \frac{\sqrt{2}}{\mathrm{RTT}\sqrt{p}} \quad \text{pkt/s}$$

where the round trip time RTT is measured in seconds, and p is the packet drop probability. The dependence on RTT is thought by some to be undesirable.

TCP increases its window size w by $i(w) = 1/w$ when it receives an ACK, and it decreases it by $d(w) = w/2$ when it detects a drop. These two functions give rise to the TCP throughput formula. Find different functions to use for increase and decrease, so that the throughput formula does not depend on RTT. *Hint. First try* $i(w) = \alpha/w$ *and* $d(w) = \beta w$ *for arbitrary constants* α *and* β*. Try to pick* α *and* β *that give the behaviour you want.*

Consider a link with service rate $C = 500$pkt/s and buffer size $B = 60$pkt. Let there be 32 TCP flows using the link, four flows at each of 8 different values of propagation delay, where the propagation delays are evenly spaced in the range 0.01 to 0.25 seconds. Assume that the round trip time is propagation delay plus queueing delay of B/C. Simulate a drift model of TCP, where each flow's window size evolves according to

$$\frac{dw(t)}{dt} = \frac{1}{\mathrm{RTT}} - p(t)\frac{w(t)^2}{2\mathrm{RTT}}$$

and where the drop probability $p(t)$ is an appropriate function of the current window sizes. What is the fixed point throughput for each flow? How long does it take, starting from zero initial windows, until all flows are within 5% of their fixed point throughput? (This is a measure of how fast TCP is at reaching its target throughput.) Repeat, but with your congestion controller rather than TCP.

The Python code in `cw2-runsim.py` at `http://www.cs.ucl.ac.uk/staff/d.wischik/Teach/NP/problems.html` runs a packet-level simulation of this scenario, with TCP congestion control. The R analysis below looks at how throughput depends on propagation delay, and it also checks the validity of the throughput formula. Run a packet-level simulation of your congestion controller, and produce equivalent graphs as the R analysis instructs.

The same Python code also runs a packet-level simulation of a scenario in which new flows arrive as a Poisson process of rate 0.5 flows/s, and where each flow has size 950 pkt. The R analysis below looks at how flow completion time depends on propagation delay. (If your congestion controller is slow to reach its target throughput, then you will see larger flow completion times.) Run a packet-level simulation of your congestion controller, and produce equivalent graphs as the R analysis instructs.

Note. You can use any programming language you like to run the simulations and to plot the graphs. However, I recommend you run the provided code, both Python and R, and simply make a small tweak to the congestion controller class. The congestion controller class for TCP, extracted from `cw2-runsim.py`*, is below. You may find it helpful to experiment further with the simulator, e.g. look at the buffer occupancy or the total arrival rate at the queue. If so, I recommend you read through the documentation of the simulator in* `cw2-sim.py`*, and then add logging similar to what I have used for the congestion controller.*

Extract from the Python simulator code

```
class TCP:
  def __init__(self,logfunc=None): self.logfunc = logfunc  # a logging function, called every ACK/drop
  def initwnd(self,currenttime):    # called by the simulator, to get the initial window size
    self.w = 1.0          # target window size i.e. number of packets in flight
    self.inflight = 1  # current number of packets in flight
    return 1              # the number of packets to send immediately when the flow starts
  def receive(self,isAck,pktdata,rtt,currenttime):  # called by the simulator, when an ack arrives or drop detected
    # isAck=True for an ACK, False for a drop
    # pktdata can be ignored (see detailed docs in cw2-sim.py for details)
    # rtt is the round trip time experienced by this packet just arrived
    # currenttime is the current simulation time, in seconds
    self.inflight -= 1
    self.w = (self.w+1.0/self.w) if isAck else max(self.w/2.0,1)
    numtosend = max(int(math.floor(self.w))-self.inflight,0)
    self.inflight += numtosend
    if self.logfunc: self.logfunc(currenttime,'ack' if isAck else 'drop',self.w,self.inflight,numtosend)
    return numtosend   # the number of new packets to send out
```

Analysis of TCP packet-level simulation data

First load in the graphics library. My library `djwutils` provides a convenient function `bwplot2` for plotting error bars; it is available at `www.cs.ucl.ac.uk/staff/D.Wischik/Teach/R`.

```
> library(lattice)
> library(djwutils)
```

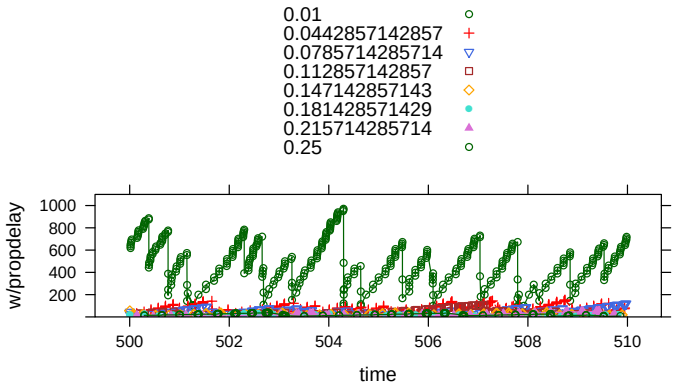
Scenario with long-lived flows

When you run `cw2-runsim.py` it produces a CSV containing simulation traces for a selection of flows. It has one row per ACK or drop. It has columns for which flow got the ACK/drop, what propagation delay the flow has, what time the ack/drop reached the source, and what the new window size is.

```
> trace <- read.csv("cw2-longlived-trace.csv")
```

The next plot shows window divided by propagation delay, which is roughly equal to throughput, as a function of time. It has one line for each of the flows represented in the log file. We see the classic TCP sawtooth.

```
> trellis.par.set(col.whitebg())
> print(xyplot(w/propdelay ~ time, groups = propdelay, data = trace,
+     subset = time > 500 & time < 510, ylim = c(0, 1100), type = "b",
+     auto.key = TRUE))
```

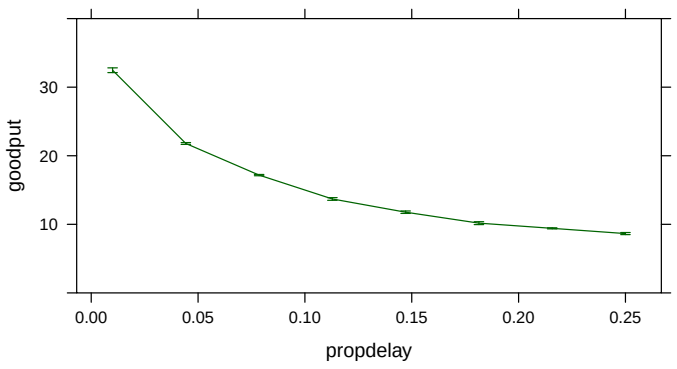


When you run `cw2-runsim.py` it also produces a CSV with flow summaries. This file has one row for each flow, and columns for propagation delay, round trip time, loss rate experienced by this flow, sendrate (pkt/s sent), and goodput (pkt/s that aren't dropped).

```
> flows <- read.csv("cw2-tcp-long-flowstats.csv")
```

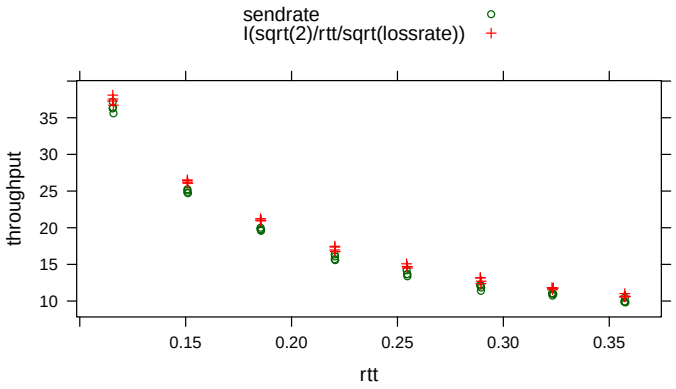
The next plot shows goodput as a function of propagation delay. The error bars show standard errors (the error bars described at the end of §2.4 in lecture notes.) **You should produce the equivalent graph for your congestion controller.**

```
> print(bwplot2(goodput ~ propdelay, data = flows, type = "l",
+     ylim = c(0, 40), err = "se"))
```



The next plot shows the sendrate for each flow, compared to what the TCP formula predicts. In applying the TCP formula I have used the actual RTT experienced by the flow, which consists of propagation delay plus queueing delay. I have also used the loss rate experienced by the flow, which might conceivably be different from flow to flow. **You should produce the equivalent graph for your congestion controller.**

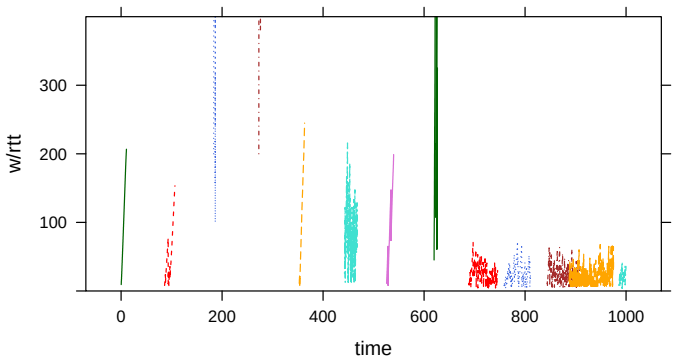
```
> print(xyplot(sendrate + I(sqrt(2)/rtt/sqrt(lossrate)) ~ rtt,
+     data = flows, type = "p", ylab = "throughput", auto.key = TRUE))
```



Scenario with Poisson flow arrivals

It's useful as a sanity check to plot window sizes as a function of time, for selected flows. If the simulated system is stable, we expect early flows and late flows to be reasonably similar. If the simulated system is unstable, we expect late flows to get systematically lower throughput.

```
> trace <- read.csv("cw2-tcp-openloop-trace.csv")
> print(xyplot(w/rtt ~ time, groups = flowid, data = trace, ylim = c(0,
+     400), type = "l"))
```

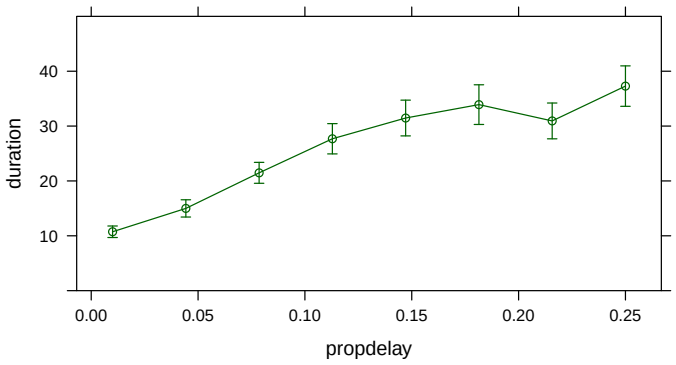


This CSV of flow statistics has the same columns as in the long-lived flows scenario, plus three additional columns: when the flow started, whether it has finished, and how long it took to finish.

```
> flows <- read.csv("cw2-tcp-openloop-flowstats.csv")
> flows$finished <- flows$finished == "True"
```

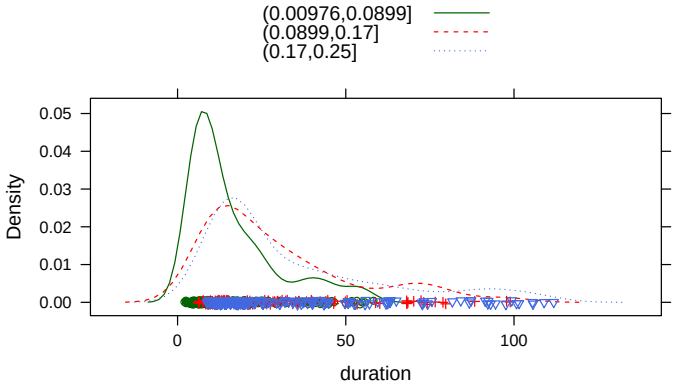
Does propagation delay influence the average flow completion time? **You should produce the equivalent plot for your congestion controller.**

```
> print(bwplot2(duration ~ propdelay, data = flows, subset = finished,
+     err = "se", type = "b", auto.key = TRUE, ylim = c(0, 50)))
```



It's interesting to know not just the average flow completion time, but also its distribution. For the next plot I've split the propagation delays into three bands (low, medium and high), and I've plotted the distribution separately for flows from each band.

```
> print(densityplot(~duration, groups = cut(propdelay, 3), data = flows,
+     subset = finished, auto.key = TRUE))
```



It's also interesting to check if the throughput formula is accurate. In the next plot I show actual measured sendrate compared to what the TCP throughput formula predicts; if they are equal then the points should lie on the grey line. I'm using log-transformed axes to get a more even spread of points across the graph. I've also used different symbols for the three bands of propagation delay I used earlier, so we can see how propagation delay affects the accuracy of the throughput formula.

```
> print(xyplot(log(sendrate, 10) ~ log(sqrt(2)/rtt/sqrt(lossrate),
+     10), groups = cut(propdelay, 3), data = flows, aspect = 1,
+     xlab = "theoretical sendrate", ylab = "actual sendrate",
+     panel = function(...) {
+         panel.abline(a = 0, b = 1, col = "grey80")
+         panel.xyplot(...)
+     }, type = "p", auto.key = TRUE))
```