Coursework 2

Simulation, Job models

Network Performance—DJW—2010/11

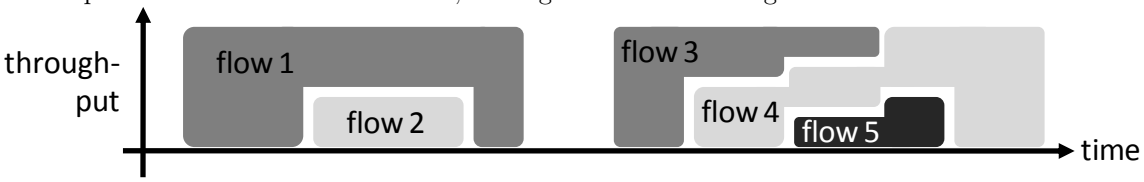
This coursework is worth 3.5% of your final grade. You should hand it in to Computer Science reception by noon on the due date.

Here is an assignment, and a solution submitted by a student. The solution is incomplete, and it contains several errors. **Your task is to explain the student's errors, correct the student's code, and complete the assignment.** Bear in mind the following points:

- The existing code has no simple compilation errors, and no simple problems like exceptions or memory-access errors. Its errors are all conceptual, i.e. they are errors in translating the assignment into code.
- The student has failed to debug his code. There are two main ways to debug simulators: (1) work out by hand what answer it should give when fed with specific inputs, and then test it, and (2) compare its outputs to known results, and investigate the reasons for any discrepancies. The assignment directs you to do both of these.
- You should explain your code and your simulation results; the level of detail given by the student is appropriate, even though his reasoning is faulty. You should also explain carefully the errors you have identified in the student's code.
- While some of the student's reasoning is faulty, other parts are good. It is up to you to evaluate which is which.
- Question 6 asks you to choose which simulations to run, and how to present your results. You should not simply pick some arbitrary parameters and give a listing of your results. Instead you should invent some interesting questions to ask, run simulations to find the answers, and present your findings as a train of reasoning. For example, you might think to yourself "What general conclusions did I draw from my answers to earlier questions? Are those conclusions still valid in this new setting?"
- I do not expect you to run simulations for hours on end. On my very old very slow computer, I ran all the simulations for the entire assignment in 20 minutes.
- Although the student's code is written in Python with data analysis in R, you may use any language you wish for your corrected version, and you may use any system you like for plotting graphs.

Assignment

Here is a simple model for a single bottleneck link shared by several TCP flows. New flows arrive at random times, and each flow has a random amount of data to send, measured in Mb. The flows share the link fairly: when there are n flows, they each get throughput C/n where C is the link speed, measured in Mb/s. Once a flow's data has all been sent, the flow departs. For example, over a period in which five flows arrive, we might see the following:



Question 1. Program a simulator of this system. Your written report should include the source code. It should also include a brief description of how your simulator works—imagine you are giving instructions to a colleague who knows how to program but who knows nothing about simulation or about networks.

Question 2. Feed your simulator with the following data: arrival times $[1,3,4,7,10]$ seconds, flow sizes $[5,3,2,5,8]$ Mb, service rate 2Mb/s. Work out with pen and paper what should happen. Verify that your simulator produces the correct output.

The obvious performance measure that matters to users is the *average flow completion time*, i.e. the average length of time that each flow is active. To investigate this, we will use a random number generator for interarrival times and for flow sizes. The following Python code generates a random number called $\text{Exp}(\lambda)$:

```
import math, random
def rexp(λ): return -1.0/λ * math.log(random.random())
```

If the times between flow arrivals are $\text{Exp}(\lambda)$ seconds, then the average arrival rate is λ arrivals per second. If the flow sizes are $\text{Exp}(1/m)$ Mb, then the average flow size is m Mb.

Question 3. Run simulations with link speed 10Mb/s, flow sizes $\text{Exp}(1)$Mb, and interarrival times $\text{Exp}(\lambda)$ seconds for a range of values of λ in the range 0 to 20. Measure the average flow completion time, and plot it as a function of arrival rate. Repeat each simulation a number of times, and add error bars to your plot to indicate how reliable your findings are.

Theory says that if the interarrival times are $\text{Exp}(\lambda)$, the flow sizes are $\text{Exp}(1/m)$ and the link speed is C, then the average flow completion time should be $m/(C-\lambda m)$.

Question 4. On your graph, superimpose the results predicted by theory. Comment on any differences.

Internet measurements suggest that a better distribution to use for flow sizes is given by the following type of random number, called $\text{Pareto}(\alpha,m)$:

```
def rpareto(α,m): return m*(1-1.0/α)*math.pow(random.random(),-1.0/α)
```

If the flow sizes are $\text{Pareto}(\alpha,m)$ Mb and $\alpha > 1$ then the mean flow size is m Mb.

Question 5. Repeat the experiment, but with $\text{Pareto}(\alpha,1)$ flow sizes, for a selection of values of $\alpha > 1$. Plot a graph with average flow duration on the vertical axis, arrival rate on the horizontal axis, and several lines, one for each value of α. Add a line for the theoretical result from earlier, $m/(C-\lambda m)$.

Question 6. Repeat the experiment, but with burstier arrivals—modify your simulator so that pairs of flows arrive close together, and then there is a longer wait until the next pair. It is up to you to choose suitable parameters and to decide how to plot your results.

Solution, by A. Student

Simulator

The simulator code is given below. The main `sim` function takes four arguments: a generator for interarrival times (`arrivals`) in seconds, a generator for flow sizes (`flowsizes`) in Mb, the link speed in Mb/s (`linkspeed`), and a logging function `logfunc` that takes one argument, a pair (departuretime,starttime), and is called whenever a flow terminates. I made the interface to `sim` generic in this way, to make it easy to modify the simulator to use different interarrival times and flow sizes, and also so that the logging code is kept separate from the simulator logic.

The simulator keeps track of the currently active flows in a list `activeflows`. For each active flow there is an entry in this list, consisting of a pair `(sizeleft,starttime)` where `sizeleft` is the amount of data left to transmit and `starttime` is when the flow started. As the simulator runs `sizeleft` decreases (line 12), but `starttime` is left unchanged and is only there for logging purposes. The `activeflows` list is kept sorted using the `heapq` Python package: this ensures that the first entry in the list is always the flow with the least left to transmit, i.e. the flow that will finish first.

The operation of the simulator is as follows. It maintains a variable `simtime`, the current (simulated) time. It also keeps track of the next scheduled flow arrival, and it calculates the time of the next departure. It works out which of these two is scheduled to happen next, and advances `simtime` correspondingly (lines 11,15). It works out the throughput given to each of the currently active flows in the meantime, and subtracts the appropriate amount from their `sizeleft` record (lines 11–13). If the next scheduled event is a departure then (because we are using `heapq`) the flow to depart is the one at the head of the `activeflows` list, so it is removed (lines 16–18). Otherwise the next scheduled event is an arrival, so we generate a new flow and add it to `activeflows`, and work out the next scheduled arrival (lines 19–22). It does this for a fixed length of simulated time (line 6). I chose the simulation time so that my program runs at a reasonable speed.

```
1  import heapq
2  def sim(arrivals,flowsizes,linkspeed,logfunc):
3      simtime = 0
4      activeflows = []
5      nextarr = arrivals.next()
6      while simtime<100:
7          throughput = linkspeed/len(activeflows) if activeflows else 0
8          if throughput>0:
9              smallestflowsize = activeflows[0][0]
10             nextdep = simtime + smallestflowsize/throughput
11             elapsed,simtime = min(nextdep,nextarr)-simtime, min(nextdep,nextarr)
12             activeflows = [(sizeleft-throughput*elapsed,starttime)
13                            for sizeleft,starttime in activeflows]
14         else:
15             nextdep,simtime = float('inf'),nextarr
16         if nextdep<=nextarr:
17             departingflow,activeflows = activeflows[0][1],activeflows[1:]
18             logfunc( (simtime,departingflow) )
19         else:
20             newflowsize = flowsizes.next()
21             heapq.heappush(activeflows, (newflowsize,simtime))
22             nextarr = simtime+arrivals.next()
```

The simulator harness uses the utility functions below. The `rexp(λ)` function is a generator which generates Exp(λ) random numbers, as specified in the assignment. I have specified the random seed to be used for random numbers in order to make my simulation results reproducible (line 25); without this it is very hard to track down bugs in a simulator based on random numbers. The `LogDuration` class provides a function `logdeparture` (line 31), which can be passed to `sim`; `sim` will then call `logdeparture((departuretime,starttime))` for each flow when it departs, and the logging class will keep a running total.

```
23 import math, random
24 def rexp(rate):
25     random.seed('geranium')
26     while True:
27         yield -1.0/rate * math.log(random.random())
```

```
28
29 class LogDuration:
30     def __init__(self): self.tot, self.n = 0,0
31     def logdeparture(self,record):
32         deptime,starttime = record
33         self.tot = self.tot+(deptime-starttime)
34         self.n = self.n+1
```

Finally, the actual simulator is run as follows. This runs simulations for a range of arrival rates, and at each arrival rate it runs 40 trials. It stores a list consisting of records which are 3-tuples, `(arrivalrate, trial number, average flow completion time)` (line 40). When it has finished, it writes all the results to a plain text file (lines 41–44), which can be loaded into a graphing package.

```
35 results = []
36 for trial in range(40):
37     for arrivalrate in [2,4,6,8,10,12,14,16,18]:
38         logger = LogDuration()
39         sim(rexp(arrivalrate),rexp(1),10, logger.logdeparture)
40         results.append( (arrivalrate,trial,logger.tot/logger.n) )
41 with open('results.csv','wt') as f:
42     f.write('arrivalrate,trial,duration\n')
43     for res in results:
44         f.write( ','.join([str(s) for s in res]) +'\n')
```

Results

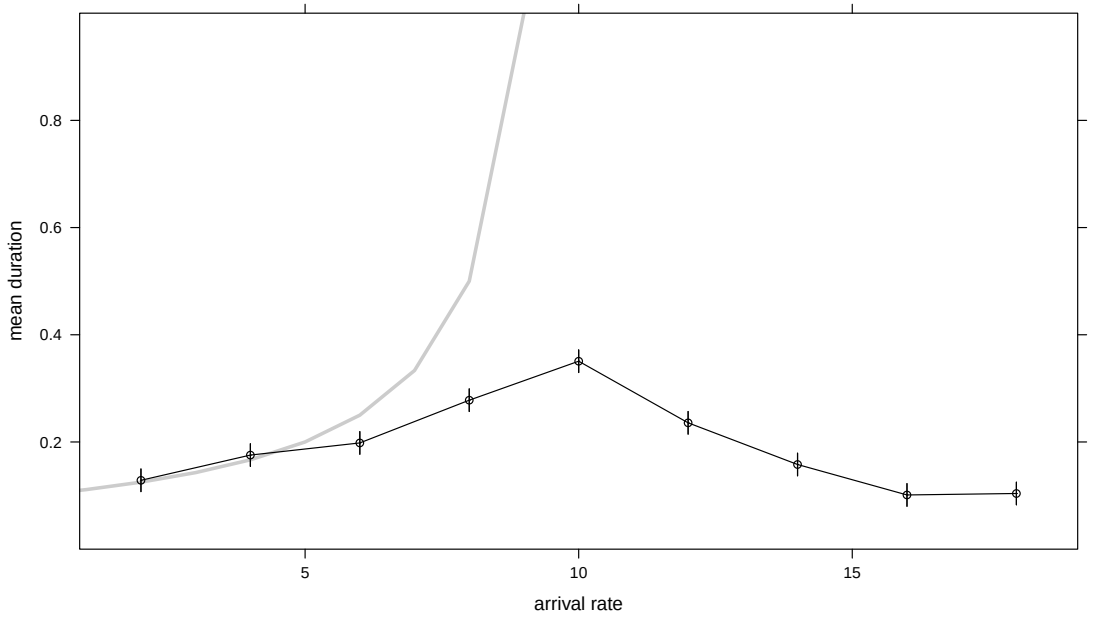
I used R to analyse the simulator output. Here are some sample values from the output.

```
> res <- read.csv("results.csv")
> res[1:5, ]
```

	arrivalrate	trial	duration
1	2	0	0.1284702
2	4	0	0.1755269
3	6	0	0.1981898
4	8	0	0.2779455
5	10	0	0.3506806

I first grouped the data by arrival rate (recall that I ran 40 trials for every value of arrival rate), and in each group I calculated the average across the 40 trials. To indicate how consistent my simulations were I also found upper and lower bounds by discarding the lowest 5% and the highest 5% of the readings; I know that 90% of the time the simulator output should lie between these values. (This is better than just plotting the minimum and maximum across trials, since minimum and maximum will be sensitive to the number of trials I run, whereas these percentile values will not.) The grouping function, `xxtabs`, is contained in a package `djwutils` provided on the Network Performance web pages. I have also plotted in grey the theoretical prediction for the average flow completion time, $m/(C-\lambda m)$ where the mean flow size is $m = 1$ Mb and the link speed is $C = 10$ Mb/s.

```
> library(djwutils)
> res2 <- xxtabs(duration ~ arrivalrate, data = res, FUN = list(low = function(x) quantile(x,
+     0.05), high = function(x) quantile(x, 0.95), mean = mean))
> res2 <- as.data.frame(res2)
> res2$arrivalrate <- as.numeric(as.character(res2$arrivalrate))
> print(xyplot(mean ~ arrivalrate, data = res2, xlab = "arrival rate", ylab = "mean duration",
+     ylim = c(0, 1), type = "b", panel = function(...) {
+         lambda <- seq(0, 20, by = 1)
+         panel.xyplot(lambda, 1/(10 - lambda), lwd = 3, col = "grey80", type = "l")
+         panel.xyplot(res2$arrivalrate, res2$mean, type = "b", col = "black")
+         larrows(res2$arrivalrate, res2$low, res2$arrivalrate, res2$high, end = "both",
+             angle = 90, length = 0.1)
+     }))
```



The error bars are small, indicating that my simulated results are highly reliable. The simulator output agrees with the theoretical prediction for small arrival rates. For larger arrival rates, the agreement is not so good. It must be that the theoretical formula relies on approximations which are only accurate when the arrival rate is small.