Coursework 2 : model solution

Note Title 13/03/2009

This coursework presents two different methods for allocating bandwidth:
* a cap/priority scheme, whereby some users ("takers") are allowed to send at full speed, up to some cap
* the status quo, in which all flows share bandwidth equally.

We are asked to model these two, in order to gain insight into how bandwidth should be allocated.

5 marks

Explain briefly the equations for θ_T and θ_S. Draw a state space diagram for this system, and note down all the transition rates.

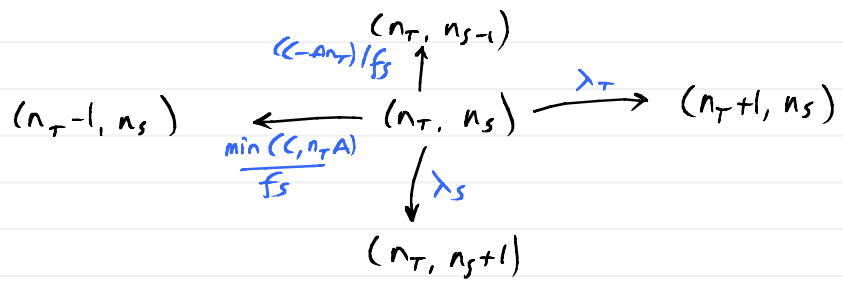
θ_T and θ_S are the throughputs obtained by takers and sharers respectively. When there are N_T takers and N_S sharers,

* the takers share the link between themselves, $\theta_T = \frac{C}{N_T}$, unless this means exceeding their rate limit A, in which case $\theta_T = A$. Hence $\theta_T = \min\left(\frac{C}{N_T}, A\right)$
* The shakers share whatever bandwidth is left over, ie $C - N_T\theta_T$. Hence

$$\theta_S = (C - N_T\theta_T)/N_S = (C - N_T \cdot \min\left(\tfrac{C}{N_T}, A\right))/N_S$$
$$= \left(C - \min(C, AN_t)\right)/N_S$$
$$= \max(0, C - AN_T)/N_S$$
$$= (C - AN_T)^+/N_S.$$

From some arbitrary state (n_T, n_S), there are four possible transitions:

* a new taker arrives, which happens at rate λ_T
* a new shaver arrives, which happens at rate λ_S
* a taker departs. There are n_T takers, with mean filesize f_T, and each gets throughput θ_T; hence the departure rate is $n_T\theta_T/f_T = \min(C, n_T A)/f_T$
* a shaver departs. Like for the takers, this has rate $n_S\theta_S/f_S = (C - An_T)^+/f_S$



Explanations of the formulae were all satisfactory.
Most people got the state space right. Some problems were

* Some students got confused about what happens when a new taker arrives and suddenly there are so many takers that $\theta_S = 0$. What happens: do current sharers get kicked out? Are new sharers blocked? or do they just sit and wait? My state space diagram assumes they just sit & wait. If you want to assume otherwise, you should say clearly what you are assuming, and you should make sure you have arrows for all possible transitions, including the transitions which result in kicking people out.

25 marks for code + output

Program an event-driven simulator of this system. Your program should be able to use any distribution for flow size, i.e. it should not be a simple Markov chain simulator.

See attached listing.

I wanted to use the same code for the cap/priority scheme and for the status quo, so I programmed a generic simulator which only knows that there are two classes of flow, call them A and B; I pass a "throughput function" to the simulator to tell it what scheme to use. The functions look like this:

```
def ratecap(c,a): return lambda nt,ns: (min(float(c)/nt,a) if nt>0 else 0, max(c-nt*a,0)/float(ns) if ns>0 else 0)
def procshare(c): return lambda nt,ns: (float(c)/(nt+ns) if nt>0 else 0, float(c)/(nt+ns) if ns>0 else 0)
```

I used cunning book-keeping — I didn't store "amount left to be transmitted" for each file, since that would require me to update every entry whenever time advances; instead I stored "incremental amounts". The complete set of simulations in this report took 56 seconds of running time, on a 3-year-old slow laptop.

My code reports the mean # takers, and the mean # sharers.
By Little's Law, we know

mean completion time for takers = mean # takers / λ_T.

mean completion time for sharers = mean # sharers / λ_S.

This saves me the bother of making my simulator log flow completion times

For each set of parameters, I did 15 runs of 10 000 events.

15 marks

Test the correctness of your program by comparing its output to theoretical results that you have been taught. Report these tests.

I ran four validations:

```
> # Summary of how many observations there are in each experiment
> xtabs(~expt, data=df)
expt
procshare   ratecap validate1 validate2 validate3 validate4
       15       195        15        15        15        15
> unique(df[,c('expt','lambdat','lambdas','ft','fs','c','a')])
         expt lambdat lambdas    ft fs   c   a
1   validate1       0      30 -1.00  1 100  -1
16  validate2      30      30  0.01  1 100   1
31  validate3      30      30  1.80  1 100 100
46  validate4      30      30  1.80  1 100  95
```

Validation 1: if there are no takers, then the sharers run pure processor-sharing. Test this by setting $\lambda_T = 0$.

```
> # Validation 1: we expect #sharers = rho/(1-rho), wait=fs/(C-lambdas*fs)
> meanci(df$ws[df$expt=='validate1'])
           n       lower        mean       upper
15.00000000  0.01401262  0.01428237  0.01455212
> 1/(100-30*1)
[1] 0.01428571
```

← lower & upper ends of the confidence interval

← theory result is well within the confidence interval

Validation 2: if takers are very small, and have low peak rate, the sharers should be very close to validation 1.

```
> # Validation 2: we expect a similar answer
> meanci(df$ws[df$expt=='validate2'])
           n       lower        mean       upper
15.00000000  0.01435605  0.01473376  0.01511146
```

← very close to Validation 1; just a little bit higher as we'd expect.

Validation 3: if takers have a rate cap $A \geq C$, then they do pure processor-sharing.

```
> # Validation 3: we expect #takers = rho/(1-rho), wait=ft/(c-lambda*ft)
> meanci(df$wt[df$expt=='validate3'])
           n       lower        mean       upper
15.00000000  0.03759765  0.03963870  0.04167976
> 1.8/(100-30*1.8)
[1] 0.03913043
```

← theory result is well within the confidence interval

Validation 4: if takers have a fairly high rate cap $A = 95$, $c = 100$, then the mean taker completion time should be very close to validation 3.

```
> # Validation 4: we expect a similar answer
> meanci(df$wt[df$expt=='validate4'])
           n       lower        mean       upper
15.00000000  0.03874682  0.04086723  0.04298764
```

← very close to Validation 3; just a little bit higher as we'd expect.

Roughly $\frac{1}{2}$ the students produced correct output;
only $\frac{1}{3}$ of students did any validation.

When your program simulators, it is very easy to introduce subtle bugs
These bugs don't make your program crash; they just make it give
incorrect output. Therefore it is vital to validate, by choosing
scenarios where you know what the output should be.

You should aim to run tests which test every part of your code.
My tests test (a) the shaver book-keeping
(b) the taker book-keeping
(c) what happens when there are both shavers & takers.

You should think carefully whether your simulator should agree <u>exactly</u> or
<u>approximately</u>. For me, validations 1 & 3 should agree exactly,
2 & 4 approximately. I used confidence intervals to test whether the
disagreement was significant. When there is significant disagreement,
you have to look more closely:

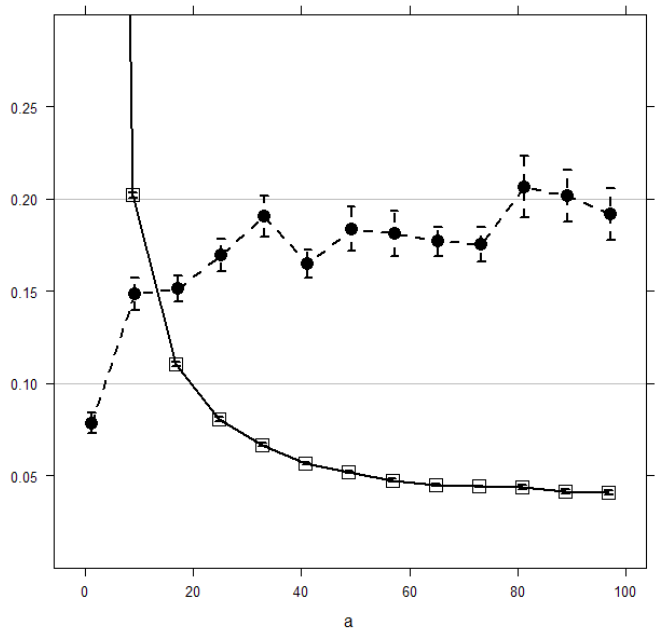
* Theory doesn't just tell you the final answer: it tells you lots more.
E.g. it tells you the mean duration of a visit to a state, and the
chance of jumping in either direction, and the equilibrium distribution.
Measure these in your simulator, and compare.

* If your simulator produces meaningless output (eg negative waiting times),
it's a good idea to set a random seed (random.seed("Damon") in Python)
so that every run produces exactly the same output. This is helpful for
stepping back and seeing where you went wrong.

I was generally disappointed by the debugging effort that students put in.

marks already listed above

Set $\lambda_T = 30$, $\lambda_S = 30$, $f_T = 1.8$, $f_s = 1$, $C = 100$, $A = 10$. Use your simulator to measure the mean completion time for the two classes of flow. Then repeat your simulations for a range of values of A, from 1 to 100. Plot your results. Remember to include error bars.



mean waiting time for takers

mean waiting time for shavers.

Error bars show 95% confidence intervals.

When you plot graphs, you don't have to blindly show every single piece of data. You should choose axes so that the most informative/interesting data is shown clearly.

If your graph has an unexpected shape, you should comment & investigate. Here, we can reasonably expect that: as the rate cap A increases,
* "taker" completion times decrease
* "shaver" completion times increase.

I selected a range of values for $A \in [1, 100]$. There is no point plotting every single integer.

It's interesting to observe much more variability for shavers than for takers. It would be good, in your answer, to speculate about why you think this might be.

10 marks (simulation)

Also run a simulation in which the link runs true processor sharing, i.e. the capacity is shared equally between all active flows, with no rate caps. This represents the status quo.

As noted earlier, my simulator does processor-sharing, when I use the throughput function

```
def procshare(c): return lambda nt,ns: (float(c)/(nt+ns) if nt>0 else 0, float(c)/(nt+ns) if ns>0 else 0)
```

5 marks (theory)

Compute these quantities. How do they compare to your simulation results?

Note that here the simulator and the theory should agree exactly. This constitutes another validation of my simulator.

```
> # Comparison of procshare sim with procshare theory. Should be ft/(c-lambda*f), fs/(c-lambda*f)
> meanci(df$wt[df$expt=='procshare'])
          n       lower        mean       upper
15.0000000   0.1020004   0.1124244   0.1228483
> meanci(df$ws[df$expt=='procshare'])
           n       lower        mean       upper
15.00000000   0.05670076   0.06177784   0.06685492
> 1.8/(100-60*1.4)
[1] 0.1125
> 1/(100-60*1.4)
[1] 0.0625
```

simulation output for taker waiting time

simulator output for sharer waiting time

theoretical prediction is in very close agreement.

Some students got very confused, and calculated $\frac{f_S}{c-\lambda_S f_S}$, $\frac{f_T}{c-\lambda_T f_T}$.
This is incorrect.
Little's Law, as we learnt in lectures, applies to any black box.

takers + sharers → [ ] → mean occupancy $= \frac{\rho}{1-\rho}$ where $\rho = \frac{\lambda f}{c} = \frac{60 \times 1.4}{100}$.

or takers → [ ] →

or sharers → [ ] →

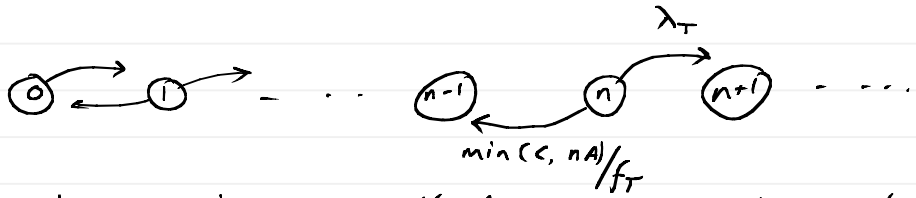
but in these two cases, we don't have any theory which lets us calculate the mean occupancy of the black box.

So, you have to use the formulae I gave in the question.

15 marks

Draw a state space diagram for the number of active takers. Find the equilibrium distribution and the mean number of active takers $\bar{n}_T$. You may find question 4 on example sheet 4 to be helpful. Assuming that there are always exactly $\bar{n}_T$ active takers, calculate the mean number of sharers. Show your theoretical predictions on the same graph that you used to show your simulation results.

Takers



This is exactly the scenario from Sh4Q4, so we can just apply the conclusion:

(ii) Calculate the equilibrium distribution π. Show that

$$\pi_n = \begin{cases} \pi_0 \left(\frac{\lambda}{A/m}\right)^n \frac{1}{n!} & \text{if } n \le \lfloor \alpha \rfloor \\ \pi_0 \left(\frac{\lambda}{A/m}\right)^{\lfloor \alpha \rfloor} \frac{1}{\lfloor \alpha \rfloor !} \left(\frac{\lambda}{C/m}\right)^{n-\lfloor \alpha \rfloor} & \text{if } n > \lfloor \alpha \rfloor \end{cases}$$

where $\alpha = C/A$ is the multiplexing ratio, and $\lfloor \alpha \rfloor$ is the floor of α, i.e. $\lfloor \alpha \rfloor \le \alpha < \lfloor \alpha \rfloor + 1$.

(iii) *[Difficult algebra]* Show that the mean number of active flows is

$$\frac{e}{e+g} F(\rho\alpha, \lfloor \alpha \rfloor) + \frac{g}{e+g}\left(\lfloor \alpha \rfloor + \frac{1}{1-\rho}\right)$$

where $e = 1/E(\rho\alpha, \lfloor \alpha \rfloor)$, $g = \rho/(1-\rho)$, $E(\rho, C)$ is the blocking probability for an Erlang link with traffic load ρ and C circuits, and $F(\rho, C)$ is the mean number of active circuits on that Erlang link.

The parameters we should use are $\rho = \frac{\lambda_T f_T}{C}$, $\alpha = \frac{C}{A}$.

Here is the code to compute these quantities:

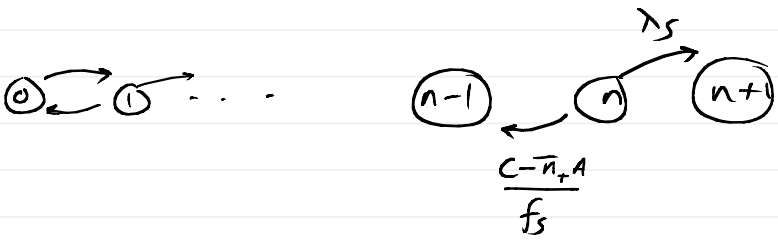
```
erlang <- function(rho,C) dpois(C,rho)/ppois(C,rho)
frlang <- function(rho,C) rho*(1-erlang(rho,C))
q4flows <- function(rho,alpha) {
      e <- 1/erlang(rho*alpha,floor(alpha))
      g <- rho/(1-rho)
      e/(e+g) * frlang(rho*alpha,floor(alpha)) + g/(e+g) * (floor(alpha)+1/(1-rho))
      }
```

I took the erlang formula from notes §3.2.
I derived the F formula using Little's Law, see model answer to Sh7Q4.

Or, you could use the formula $E(\rho, c) = \frac{\rho^c/c!}{\sum_{i=0}^{c} \rho^i/i!}$
and you could get $F(\rho, c)$ by
$F(\rho, c) = \mathbb{E}\#\text{circuits busy} = \sum_{n=0}^{c} n \times \mathbb{P}(\text{circuits are busy}) = \sum_{n=0}^{c} n \times \frac{\rho^n/n!}{\sum_{i=0}^{c} \rho^i/i!}$

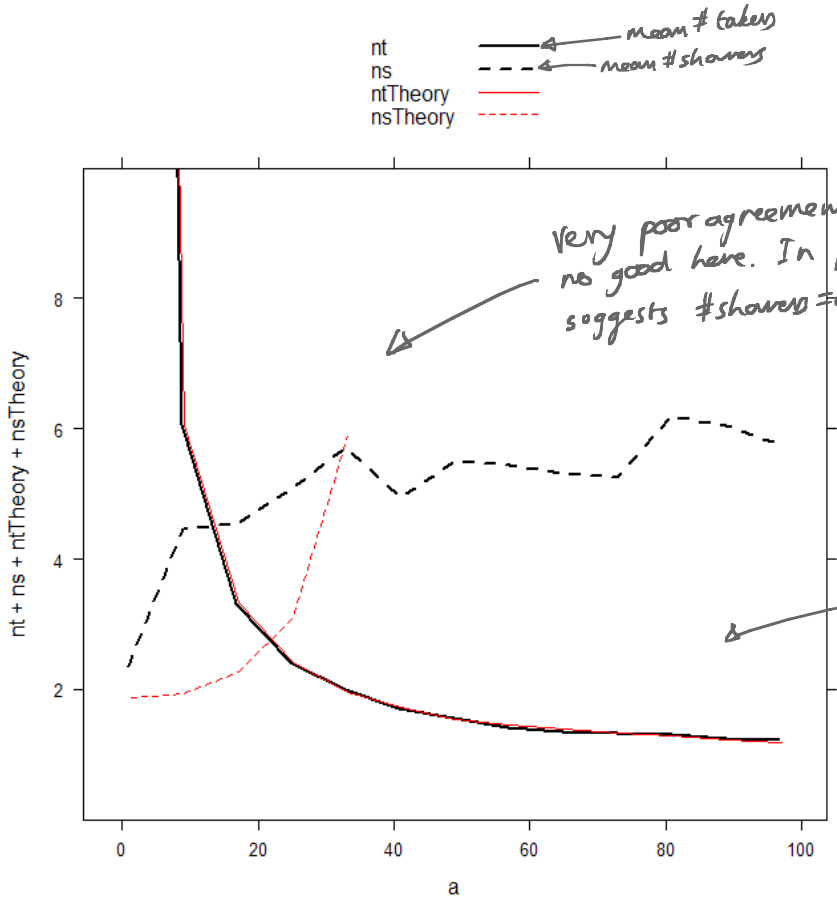
Or, you could calculate π_n for n in some big range (e.g. $n = 0$ up to 1000) from the formula, then use
mean # active flows $= \sum_{n=0}^{1000} n \times \pi_n$.

Sharers



This is exactly the model of a processor-sharing link;

mean # flows $= \frac{\rho}{1-\rho}$ where $\rho = \frac{\lambda_s f_s}{C - \bar{n}_t A}$; if $\rho > 1$ then unstable $\Rightarrow$ mean # flows $= \infty$.



Very poor agreement. The fixed point approximation is no good here. In particular, the approximation suggests #shared $= \infty$ when $\frac{\lambda_s f_s}{C - \bar{n}_t A} > 1$; but it looks stable on this plot for a much wider range of A.

The theory and the sim agree perfectly — as they should, since the theory is an exact calculation and not an approximation.

Very few students worked out how to apply the theory of SheetQ4. Those who did, did it correctly.

This question forces you to think deeply about the link between sim & theory. The approximation we made forces us into conclusions about instability which are not experimentally true. Therefore the approximation is no good.

20 style marks, for overall presentation.

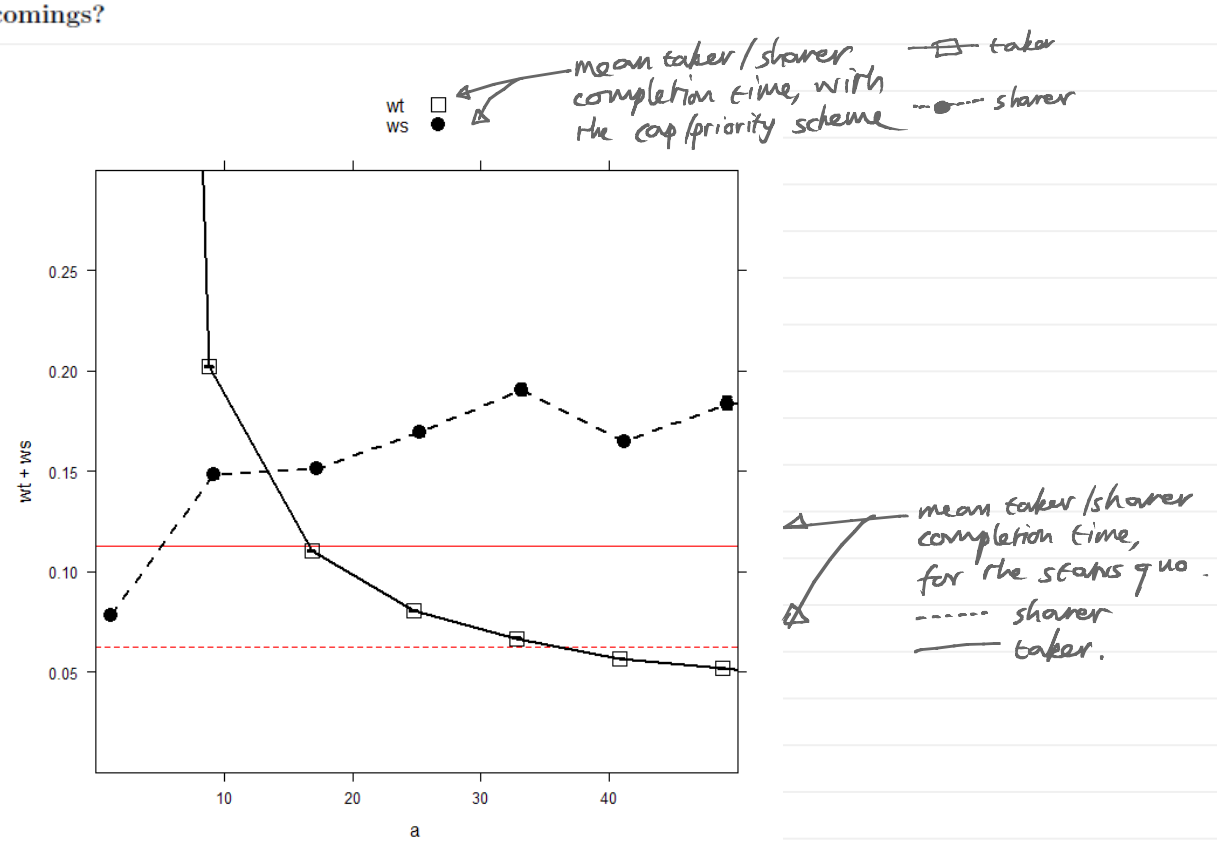
Repeat your experiment with a range of other parameter values. You should pay particular attention to parameter values where the theoretical approximation suggests the system will be unstable. Explain why you chose the parameter values you did. You should include any extra observations or mathematical analyses that shed light on your findings.

There were no dedicated marks for this. However, students who do a good exploration of the parameter space, touching on the issue of stability that I described above, got overall "style marks".

Style marks are also awarded for conciseness, clarity, clear explanations, good plots.

5 marks

What does this model tell you about network neutrality? Is it a useful contribution to the debate about μTorrent's actions? If not, what are its major shortcomings?



Compared to the status quo, the takers only get enhanced service with cap/priority when $A \geq 0.16\,C$, ie a contention ratio of $1/0.16 = 6.25$.
The sharers, on the other hand, get worse service whatever the value of A; at $A = 0.16\,C$ it's more than twice as bad as the status quo, for them.

This cap/priority scheme would only be beneficial to users if A was very large, and the "takers" paid a lot more than they are paying now, so the ISP could invest in extra capacity, so that the sharers don't suffer.

For a fixed level of capacity, it seems that everyone gets better service if they stick with the status quo. Therefore the Slashdot comment at the top of this coursework is entirely mistaken.

The most obvious limitation of the model is that it doesn't take account of the fact that users may have many flows open, so the rate cap should really be per-user rather than per-flow.

This is not an opportunity for you to waffle.

You need to think: does this model tell us anything about the debate?
We have looked at two schemes: cap/priority, and processor sharing.
This implies a question: which of the two is better? for whom?

This question mirrors the question raised by the quotes: would it be good for BitTorrent to switch to UDP & for ISPs to impose caps, or are we better off sharing as with TCP?

I was disappointed that very few students were able to draw the link between the theory/simulation work, and the real-world problem.

```
#-------------------------------------------------------------------------------
# Load in some standard functions, and define some basic probability functions

import time
import heapq
import math
from random import random
Inf = float('inf')
def rexp(lambd):
    """Generate an exponential random variable with rate lambd"""
    if lambd==0: return Inf
    u = random()
    return (-1.0/lambd) * math.log(u)
def mean(d):
    """Given a dictionary of value,prob terms representing a distribution, find the mean"""
    return sum(k*v for k,v in d.iteritems())

#-------------------------------------------------------------------------------
## A link with two classes of user
#
# My code here is generic, not specifically about takers and sharers --
# you pass the code a function called throughputFunction, which calculates the throughput that
# users of each class will receive.

class ProcShareLink:
    """A link shared between two classes of users, called A and B, which get throughputs (thA,thB)=func(#A,#B)"""
    def __init__(self,throughputFunction):
        self.throughputFunction = throughputFunction
        self.waitingA = []
        self.waitingB = []
        self.workdoneperA = 0.0
        self.workdoneperB = 0.0
    def arrivalA(self,flowsize):
        heapq.heappush(self.waitingA,self.workdoneperA+flowsize)
    def arrivalB(self,flowsize):
        heapq.heappush(self.waitingB,self.workdoneperB+flowsize)
    def advancetime(self,by,isdeparture=False):
        nA,nB = len(self.waitingA),len(self.waitingB)
        throughputA,throughputB = self.throughputFunction(nA,nB)
        self.workdoneperA += throughputA*by
        self.workdoneperB += throughputB*by
        if isdeparture:
            depB = nB>0 and (nA==0 or (self.waitingB[0]-self.workdoneperB < self.waitingA[0]-self.workdoneperA))
            if depB: heapq.heappop(self.waitingB)
            else: heapq.heappop(self.waitingA)
    def timeuntilnextcompletion(self):
        if len(self.waitingA)+len(self.waitingB)==0: return Inf
        nA,nB = len(self.waitingA),len(self.waitingB)
        throughputA,throughputB = self.throughputFunction(nA,nB)
        timetodepartA = (self.waitingA[0]-self.workdoneperA)/throughputA if throughputA>0 else Inf
        timetodepartB = (self.waitingB[0]-self.workdoneperB)/throughputB if throughputB>0 else Inf
        return min(timetodepartA,timetodepartB)

def runsim(lambdaA,lambdaB,filesizeA,filesizeB,throughputFunction):
    link = ProcShareLink(throughputFunction)
    def randomInterarrival(): return rexp(lambdaA+lambdaB)
    def randomIsB(): return random() < (0.0+lambdaB)/(lambdaA+lambdaB)
    def randomFilesizeA(): return rexp(1.0/filesizeA)
    def randomFilesizeB(): return rexp(1.0/filesizeB)
    #
    simtime = 0.0
    nextarrival = simtime + randomInterarrival()
    trace = [(0,0,0)]  # store the trace as a list of (simtime,#A,#B)
    for i in range(10000):
        nextdeparture = simtime + link.timeuntilnextcompletion()
        if nextarrival<nextdeparture:
            link.advancetime(nextarrival-simtime)
            if randomIsB(): link.arrivalB(randomFilesizeB())
            else: link.arrivalA(randomFilesizeA())
            simtime = nextarrival
            nextarrival = simtime + randomInterarrival()
        else:
            link.advancetime(nextdeparture-simtime, isdeparture=True)
            simtime = nextdeparture
        trace.append( (simtime,len(link.waitingA),len(link.waitingB)) )
    #
    # Find the fraction of time spent at each level of #A, and at each level of #B
    # Return the mean #A and the mean #B
    timeWithAat,timeWithBat = {},{}
    tottime = trace[-1][0]
    t,nA,nB = trace[0]
    for t2,nA2,nB2 in trace[1:]:
        timeWithAat[nA] = (t2-t)/tottime + (timeWithAat[nA] if nA in timeWithAat else 0)
        timeWithBat[nB] = (t2-t)/tottime + (timeWithBat[nB] if nB in timeWithBat else 0)
        t,nA,nB = t2,nA2,nB2
    return sum(k*v for k,v in timeWithAat.iteritems()), sum(k*v for k,v in timeWithBat.iteritems())



#-------------------------------------------------------------------------------
## Now we specialize to takers and sharers, with the throughput function specified in the coursework.

# The command ratecap(c,a) returns a function f, where f(nt,ns) is the throughput that takers and sharers
# get respectively when there are nt takers and ns sharers. For example, ratecap(100,10)(4,10)=(10,6)
# (In Python, the lambda command defines an anonymous function.)
def ratecap(c,a): return lambda nt,ns: (min(float(c)/nt,a) if nt>0 else 0, max(c-nt*a,0)/float(ns) if ns>0 else 0)
def procshare(c): return lambda nt,ns: (float(c)/(nt+ns) if nt>0 else 0, float(c)/(nt+ns) if ns>0 else 0)

# Validation 1: no takers, so sharers should be pure processor sharing
res1 = [("validate1",0,30,-1,1,100,-1, runsim(0,30,-1,1,ratecap(100,1))) for r in range(15)]
# Validation 2: very small takers, so sharers should be nearly pure processor sharing
```

```
res2 = [("validate2",30,30,.01,1,100,1, runsim(30,30,.01,1,ratecap(100,1))) for r in range(15)]
# Validation 3: large rate cap for the takers, so they should be pure processor sharing
res3 = [("validate3",30,30,1.8,1,100,100, runsim(30,30,1.8,1,ratecap(100,100))) for r in range(15)]
# Validation 4: pretty large rate cap for the takers, so they should be nearly pure processor sharing
res4 = [("validate4",30,30,1.8,1,100,95, runsim(30,30,1.8,1,ratecap(100,95))) for r in range(15)]
# Main experiment: 15 runs at a range of values of a, for lambdat=30,lambdas=30,ft=1.8,fs=1,c=100
res5 = [("ratecap",30,30,1.8,1,100,a, runsim(30,30,1.8,1,ratecap(100,a))) for a in range(1,100,8) for r in range(15)]
# Test of pure proc sharing
res6 = [("procshare",30,30,1.8,1,100,-1, runsim(30,30,1.8,1,procshare(100))) for r in range(15)]

allres = [res1,res2,res3,res4,res5,res6]
# Write it all out to a log, to load into R for plotting
f = open('cw2-out.csv','wt')
print >>f, 'expt,lambdat,lambdas,ft,fs,c,a,nt,ns'
for r in allres:
    for expt,lambdat,lambdas,ft,fs,c,a,(nt,ns) in r:
        print >>f, ','.join([str(x) for x in [expt,lambdat,lambdas,ft,fs,c,a,nt,ns]])
f.close()


#-------------------------------------------------------------------------------
## R code for plotting & analysis

# Load in a library with plotting routines, and set graphics style
library(djwutils)
gp <- col.whitebg()
gp$superpose.line$col <- c('black','black','red','red')
gp$superpose.line$lwd <- c(2,2,1,1)
gp$superpose.line$lty <- c(1,2)
gp$superpose.symbol$pch <- c(0,19)
gp$superpose.symbol$col <- c('black','black','red','red')
gp$superpose.symbol$cex <- 1.5
trellis.par.set(gp)

# load the data
df <- read.csv('cw2-out.csv')
df$wt <- df$nt/df$lambdat
df$ws <- df$ns/df$lambdas

# a utility function, to show mean and confidence interval
meanci <- function(x) c(n=length(x),
                        lower=mean(x)-1.96*sd(x)/sqrt(length(x)-1),
                        mean=mean(x),
                        upper=mean(x)+1.96*sd(x)/sqrt(length(x)-1))

# Summary of how many observations there are in each experiment, to remind ourself
xtabs(~expt, data=df)
unique(df[,c('expt','lambdat','lambdas','ft','fs','c','a')])

# Validation 1: we expect #sharers = rho/(1-rho), wait=fs/(C-lambdas*fs)
meanci(df$ws[df$expt=='validate1'])
1/(100-30*1)

# Validation 2: we expect a similar answer
meanci(df$ws[df$expt=='validate2'])

# Validation 3: we expect #takers = rho/(1-rho), wait=ft/(c-lambda*ft)
meanci(df$wt[df$expt=='validate3'])
1.8/(100-30*1.8)

# Validation 4: we expect a similar answer
meanci(df$wt[df$expt=='validate4'])

# Plot of main experimental outcome
bwplot2(wt+ws~a, data=df, subset=expt=='ratecap',
        err=.95,type='b', ylim=c(0,0.3), panel=function(...) { panel.abline(h=c(0,.1,.2),col='grey75'); panel.bwplot2(...) },
        auto.key=TRUE)

# Comparison of procshare sim with procshare theory. Should be ft/(c-lambda*f), fs/(c-lambda*f)
meanci(df$wt[df$expt=='procshare'])
meanci(df$ws[df$expt=='procshare'])
1.8/(100-60*1.4)
1/(100-60*1.4)

# Comparison of ratecap sim with ratecap theory.
erlang <- function(rho,C) dpois(C,rho)/ppois(C,rho)
frlang <- function(rho,C) rho*(1-erlang(rho,C))
q4flows <- function(rho,alpha) {
    e <- 1/erlang(rho*alpha,floor(alpha))
    g <- rho/(1-rho)
    e/(e+g) * frlang(rho*alpha,floor(alpha)) + g/(e+g) * (floor(alpha)+1/(1-rho))
    }
dfrc <- df[df$expt=='ratecap',]
dfrc$ntTheory <- q4flows(dfrc$lambdat*dfrc$ft/dfrc$c,dfrc$c/dfrc$a)
dfrc$nsTheory <- dfrc$lambdas*dfrc$fs/(dfrc$c-dfrc$ntTheory*dfrc$a-dfrc$lambdas*dfrc$fs)
dfrc$nsTheory[dfrc$nsTheory<0] <- NA
#
bwplot2(nt+ns+ntTheory+nsTheory~a, data=dfrc,
        err='none', type='l', ylim=c(0,10),
        auto.key=list(lines=TRUE,points=FALSE))

# Comparison of status quo to ratecap scheme
bwplot2(wt+ws~a, data=df, subset=expt=='ratecap',
        err='none',type='b', ylim=c(0,.3), xlim=c(0,50),
        panel=function(...) {
            panel.abline(h=1/(100-60*1.4),col='red',lty=2,lwd=1) # sharers
            panel.abline(h=1.8/(100-60*1.4),col='red',lty=1,lwd=1) # takers
            panel.bwplot2(...) },
        auto.key=TRUE)
```