Lecture 31 — Introduction to Queueing Theory

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 4, 2019

Literally the theory of queues – what makes queues appear, how will they behave, and how do we make them go away?

Queueing theory has played a role in your life whether you know it or not.

This is how tech support at Rogers or Bell or Telus or whomever decides just how many customer service agents to have available at any given time.

"Your call is important to us; please hold while we ignore it."

Queueing theory is applicable to lots of fields, including industrial design, call centres, telecom systems, and computers executing transactions.

Questions Queueing Theory Helps Answer

- Given a choice between a single machine with speed s or n machines, each with speed s/n, which should we choose?
- If the arrival rate and service rate double, how does the mean response time change?
- Should we try to balance load or is that a waste of time/effort?
- Can we give priority to certain operations without harming another category of job?
- How do job size variability and heavy-tailed workloads affect our choices of scheduling policy?
- If 12 servers is enough to handle 9 jobs per second, do we need 12 000 servers if we have an arrival rate if 9 000 jobs per second?

- Server - The banking centre fulfilling customer requests.
- Customer - Initiator of service requests.
- Wait time - The time a customer spends waiting in line.
- Service time - The time from when a teller starts to serve a customer up to the time when the next customer is called forward.
- Arrival rate - The rate at which customers arrive.
- Service rate - the rate at which customer requests are serviced.

- Utilization - The fraction of the teller's time used actually handling customer requests (not idling).
- Queue length - The total number of customers waiting, or currently with a teller, or both.
- Response time - The sum of wait and service time for a single visit.
- Residence time - The total response time if a customer visits several tellers (or the same one multiple times).
- Throughput - The rate at which customers get their requests serviced and dealt with.

Symbol	Semantics
S	Service time
V	Number of visits to the server
D	Service demand
R	Response time
R'	Residence time
X	Throughput
λ	Arrival rate
U	Utilization
W	Wait time
N	Total queue length (waiting and/or being serviced)

It's actually (allegedly) a trade-off: if I have to wait too long to do my banking, I could always take my business elsewhere.

Minimizing customer wait time makes customers happy, so that's something the bank should want.

It would also be nice if the bank trains its tellers well, so they can complete all operations, even unusual ones, quickly and efficiently, reducing service time.

Overstaffed is bad and understaffed is also bad.

Back to the realm of computers: you have lots of queues in your computer.

The CPU uses a time-sharing scheduler to run as many concurrent programs as possible.

A router has a queue for packets (data) that has a maximum size, and if this is exceeded, packets will be simply dropped.

Queueing theory gives us a formal framework with which to grapple with our problems instead of just guessing.

Imagine we have a system with one CPU that serves a queue of jobs in First-Come-First-Served (FCFS) order with an arrival rate λ of 3 jobs per second.

Each job takes some amount of time and resources.

Suppose the average service rate μ is 5 jobs per second.

The system is not overloaded – 3 jobs per second arriving is less than 5 jobs per second being serviced.

Suppose now that your boss says that tomorrow the arrival rate will double.

If you do nothing: 6 jobs arriving per second, on average, to a system that can service, on average, 5 jobs per second.

You have been allocated some budget to replace the CPU with a faster one.

FCFS CPU

$\lambda = 3$

$\mu = 5$
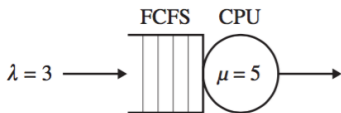
If $\lambda \to 2\lambda$,
by how much
should μ increase?

Goal: Customers should not notice the increase in arrival rate.

So, should we (1) double the CPU speed; (2) more than double the CPU speed; or (3) less than double the CPU speed?

The answer is (3) - we don't need to double the CPU speed.

We can see later in formal terms why this is the case, but think for a minute about what it is?

If we double the service rate and double the arrival rate, we actually get half the mean response time...

There are always $N = 6$ jobs running at a time.

As soon as a job completes, a new one is started (this is called a closed system).

Each job goes through to be processed on one of two servers, each of which has a service time μ of 1 job per 3 seconds.

$N = 6$ jobs

Server 1

$\mu = \frac{1}{3}$

$\frac{1}{2}$

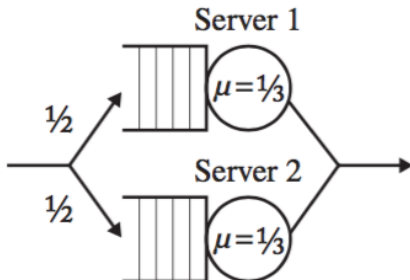Server 2

$\mu = \frac{1}{3}$

$\frac{1}{2}$
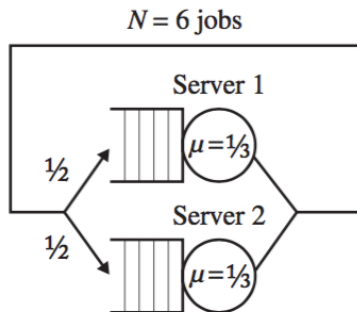
Bad news: sometimes, improvements do nothing.

If we replace server 1 with one which is $2\times$ faster (so 2 jobs per 3 seconds), does that help? *Nope. Not really.*

Does raising N help? *Nope, negligible effect.*

Strangely, dropping N to 1 means the server replacement makes a difference, if you can call that improvement.

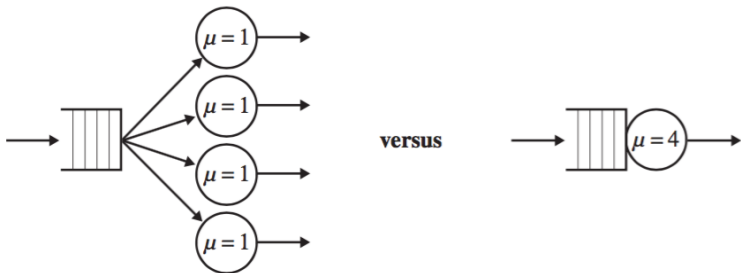What if it's an open system where arrival times are independent of completion?

In this case, yes, replacing server 1 makes a huge difference!

Do we want one fast server or *n* slower ones?

What is better if we want to minimize the mean response time when we have non-preemptable jobs?

One big factor is the variability of the job sizes.

Imagine you are at the grocery store and most people have 12 items or fewer and there's one guy who's buying 85 items.

You don't want to be standing in line with milk and eggs behind someone who is trying to buy six of everything, do you?

So if there's high variability, you probably want multiple servers.

What if the load is low?

Chances are you would prefer the one fast server instead of having some number of servers doing nothing.

What if jobs are interruptible (preemptible)?

You could always use a single fast machine to simulate n slow machines, so a single fast machine is at least as good as the alternative.

Imagine your typical "server farm" - you have n servers that are all responsible for handling incoming requests.
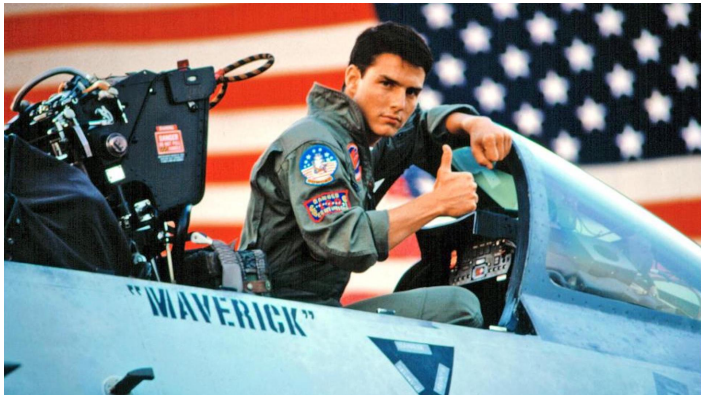
Let's imagine all servers are the same (or close enough).

What we typically see in load balancing is assignment of tasks to servers via some dispatcher:
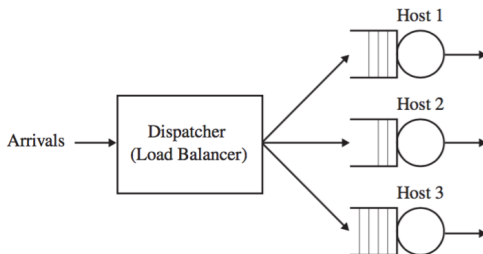
There are a few different task assignment policies - ways in which we can assign work to servers:

- Random
- Round-Robin
- Shortest-Queue
- Size-Interval-Task-Assignment
- Least-Work-Left
- Central-Queue

"You'll never say hello to you
Until you get it on the red line overload
You'll never know what you can do
Until you get it up as high as you can go"

- Kenny Loggins

Earlier I mentioned it would probably be bad to see 6 jobs arriving per second to a system that can handle 5 per second.

This doesn't seem like rocket science, but it bears repeating.

In our discussion we require that $\lambda \leq \mu$ and assume that $\lambda < \mu$.

That is to say, we are not overloaded.

Remember now that the values for λ and μ are averages.

It could happen that temporarily we "fall behind" a bit, but then make up for it a little later on...

Think about the long term, though—if we are not at least keeping up then this will eventually get out of hand.

How badly? Well, in the limit, the queue length goes to infinity.

Let's represent time with t and define $N(t)$ as the number of jobs in the system at time t.

$A(t)$ represents arrivals by time t and $D(t)$ represents departures by time t.

$$E[N(t)] = E[A(t)] - E[D(t)] \geq \lambda t - \mu t = t(\lambda - \mu)$$

If arrivals exceed departures $t(\lambda - \mu)$ also goes to infinity.

Raising μ is generally desirable.

This is, after all, programming for performance—the faster we complete work, the more work we can get done in the same amount of time.

Improving the service rate, however, does not necessarily improve the throughput.

We've assumed that the arrival rate is less than the service rate.

So we have enough capacity to handle all incoming work.

So the limiting factor on completed work is actually arriving work.

We have the capacity to do at least what is arriving and possibly a bit more.

Adding more work capacity doesn't mean more work gets done if there isn't any more work to do.

So raising μ increases the maximum possible throughput, but does not necessarily increase the actual throughput.

In a closed system: one in which there is always more work to do and as soon as one item is finished the next one enters the queue.

In that case, we are running at capacity all the time, so actually μ is the controlling factor – the throughput is exactly the service rate.
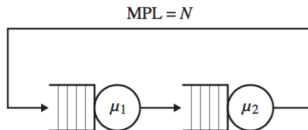
What is the throughput here? Intuition suggests $\min(\mu_1, \mu_2)$, right?

Sometimes. This is okay if the slower server is always busy, but that's not always the case.

What if N is 1? Okay, that's a bit of an exception case though.

What about N being 2? Then the slower server has some work to do at all times right?

Nope, sadly not.

Sometimes the slow server is faster than the fast server, because μ_1 and μ_2 are just averages.

And averages can be misleading!

The average family might have 2.3 children (or whatever the figure is), but you can't exactly have 0.3 of a child...

Some smart folks at IBM wanted to know, given the arrival rate λ, what was the mean job size, $E[S]$ (which is $1/\mu$).

Our experiment should be a hundred runs of sending a single job into the system and averaging the values.

This is okay, but does not reflect reality where we have things like caching of data and multiple concurrent jobs.

The open system strategy is: ramp up λ.

Keep piling more jobs on the system.

At some point the system will not be able to keep up. Once the completion rate levels off, we hit the limit and we have a value for μ.

The closed system strategy: set it up so there is always work to do.

In closed systems, there's often consideration given to *think time* – this is what happens when the user is on the command line and dispatching work to do.

To keep the system totally busy in the stress test, we need think time to be zero—so additional work is always available.

And then we can simply measure the jobs completing per second, giving us μ directly.