Distributed Computing

A-07. Queueing and Scheduling

Queues: The Simplest Model

A Little Bit of Queueing Theory

- Let's consider a simple case, for the moment: we have a single server and a queue of **jobs** it has to serve (web pages, computations, …)

(image by Tsaitgast on Wikipedia, CC-BY-SA 3.0 license)

- λ and μ are the average frequencies (**rates**) at which jobs respectively join the queue and leave the system when they are complete: in time dt, the probability a job arrives or leaves when being served are respectively λdt and μdt
- Jobs are served in a First-In-First-Out fashion

The Simplest Model

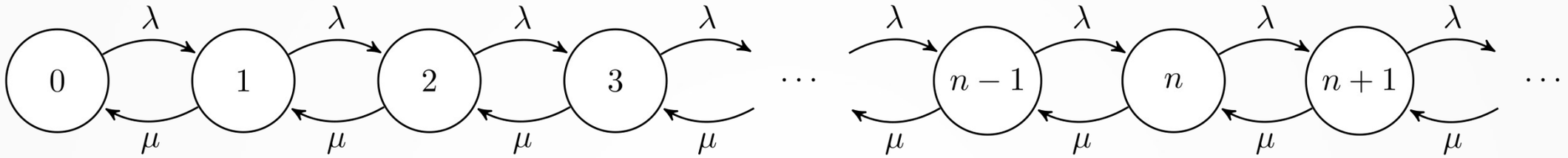
Reference: Harrison & Patel, 1992, chapters 4-5

- This is called **M/M/1** (**Kendall** notation) queue because
  - Jobs arrive in a **memoryless** fashion: no matter what happened until now, the probability of seeing a new job in a time unit **never changes**
  - Jobs are served in a **memoryless** fashion: the probability of a job finishing does not depend on how long it's been served, or anything else
  - Just **one** server

Wait--Why This Model?

- “All models are wrong, but some are useful” (George Box)
- Real systems are **not** like this, but some of the insight **does apply** to real-world use cases
  - The memoryless property makes it **much easier** to derive closed-form formulas
  - (Some of) the insight we get will be **useful**
  - We can compare & contrast with **simulations**
  - And we need to verify whether simulations are representative too

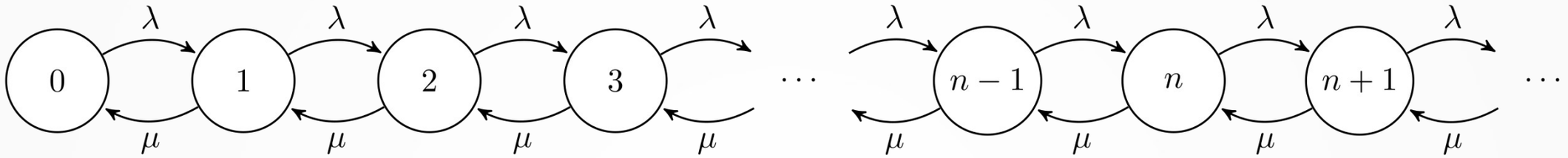
Let's Analyze M/M/1



(image by Gareth Jones on Wikipedia, CC-BY-SA 3.0 license)

- How does this model behave **on the long run?**
  - What is the probability of having *x* jobs in the queue?
  - What is the **amount of time** a typical job will wait before being served?
- We will start by looking at the first question, looking for an **equilibrium probability distribution**

M/M/1 Equilibrium



- To be an equilibrium we need
  - $\lambda<\mu$, otherwise the number of elements in the queue will keep growing
  - That in any moment the probability of moving from state *i* to *i+1* is the same of moving in the opposite direction:

$$\lambda\, p_i\, dt = \mu\, p_{i+1}\, dt$$

$$p_{i+1} = \frac{\lambda}{\mu}\, p_i$$

Some Easy Algebra

- Let's simplify and say $\mu=1$ (just change the time unit)

$$p_{i+1}=\lambda p_i$$

hence

$$p_1=\lambda p_0, p_2=\lambda^2 p_0, \ldots, p_i=\lambda^i p_0$$

- Since this is a probability distribution, its sum must be one. We can then solve everything:

$$\sum_{i=0}^{\infty} \lambda^i p_0=1; \frac{1}{1-\lambda} p_0=1; p_0=1-\lambda; p_i=(1-\lambda)\lambda^i$$

- The average queue length is

$$L=\sum_{i=0}^{\infty} i p_i=(1-\lambda)\sum_{i=0}^{\infty}(i\lambda^i)=(1-\lambda)\frac{\lambda}{(1-\lambda)^2}=\frac{\lambda}{1-\lambda}$$

Little's Law

- Beautifully simple: L equals the **average time spent in the system *W*** times the arrival rate:

$$L = \lambda W$$

- Why? Consider this: if a time unit spent in the system by a job "costs" 1€, then jobs would spend *W*€ on average.
- In an average time unit, the system will collect *L*€ (because on average *L* jobs are in queue); in equilibrium and on average λ jobs will arrive and leave the system, spending a total of λW.

So We've Got Our Result

- The average time spent in the system for an M/M/1 FIFO queue is

$$W = \frac{L}{\lambda} = \frac{\left(\frac{\lambda}{1-\lambda}\right)}{\lambda} = \frac{1}{1-\lambda}$$

Multiple Servers

Multi-Server Version

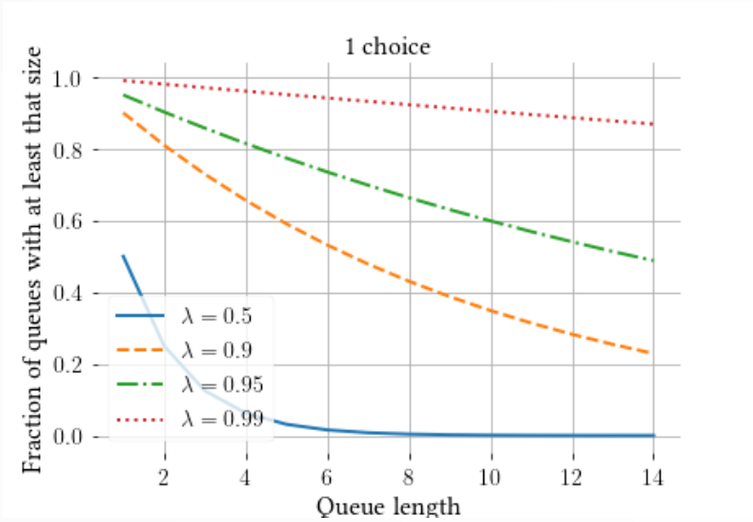
- Rather than a single server, there are *n* servers
- You have a **load balancer** to divide the load between servers
- How to assign load to servers?
  - If randomly, you can handle a load of $n\lambda$ with the performance of a single server having load λ
  - Can we do better?
  - We can assign jobs to the **least loaded server!**

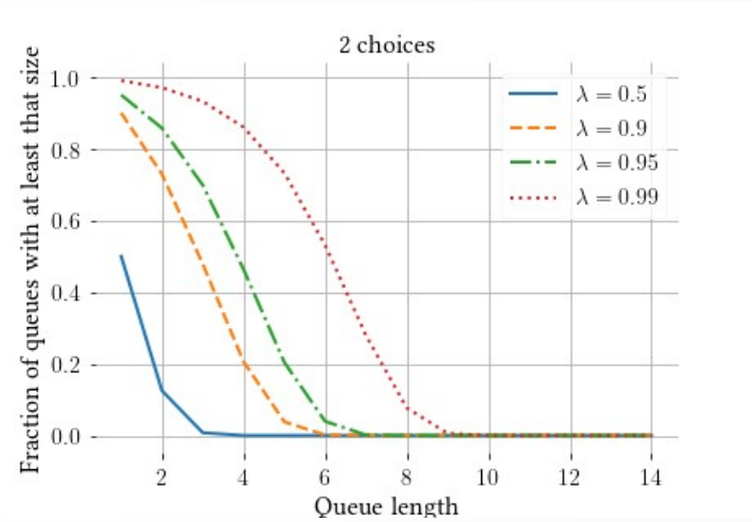
Supermarket Queueing

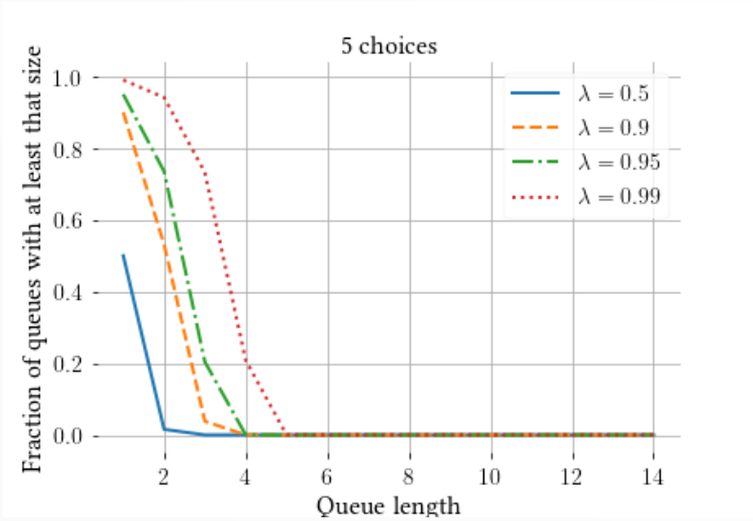
- Idea: rather than querying *n* servers to find out their queue length, just ask a small number *d* of them and assign to the shortest queue
- Mitzenmacher (2001) finds that the fraction of queues with at least I jobs drops from λ^i *to*

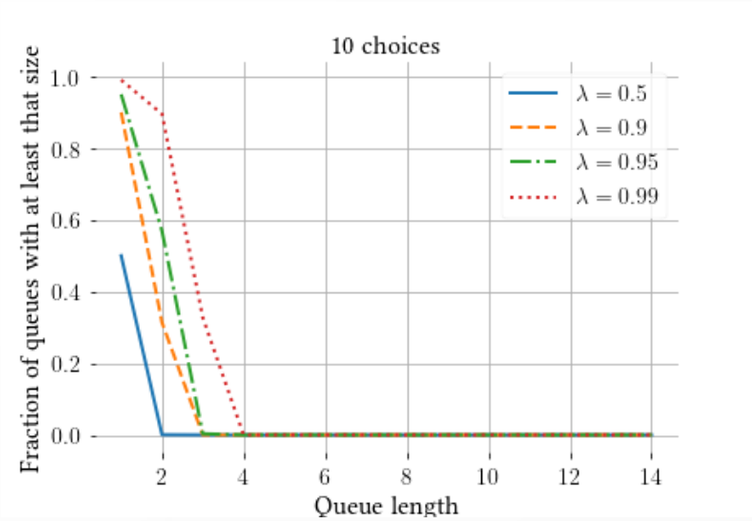
$$\lambda^{\frac{d^i-1}{d-1}}$$

Theoretical Queue Length









Beyond First-In-First-Out

Preemptive Policies

- In the real world, you very often have **many very small jobs** and **a few extremely large ones**
  - The memoryless property doesn't apply!
- You can **preempt** running jobs: pause them and resume them later
- Practical, widely adopted, solution: **round-robin** scheduling, where you run a job for a given timeslot and then pass the turn to another one
  - If *n* jobs are in the queue, each proceeds at speed *1/n*
  - Often adapted with priorities reflecting job importance

Size-Based Scheduling

- If you know **for how long** a job will be running (its **size**), you can do something smarter
  - E.g., you may know how long an algorithm will take or how big is the file you're serving
- To minimize the average time spent **W**, the optimal policy is **Shortest Remaining Processing Time** (SRPT), which always serves the job needing the least work to complete
  - Theoretical possibility of **starvation**: large jobs are never served—not really relevant in practice (Harchol-Balter et al., 2003)
  - Sketch of optimality proof at the blackboard (Schrage, 1968)

Errors in Size Estimation

- SRPT may behave catastrophically
  if jobs' size information is incorrect
  - If large jobs are underestimated, their "remaining processing time" goes below 0 and "blocks" the system until they are finished
  - Particularly problematic with very diverse job sizes
- Simple solution: forget about the "remaining" processing time and schedule first the jobs with smallest estimated size!
  - Shown to work well both in simulations and theory
  - **Assuming** estimation error is **proportional to real size**

Conclusions

Evaluating Distributed Systems

- Mathematical models
  - Good: “hard truths” for the modeled world
  - Bad: needs simplifications: the modeled world is not the real world
- Experiments & measurements on real systems
  - Good: evaluating the “real thing”
  - Bad: overly focused on implementation details, expensive, limited
- Simulations
  - Good: (to some extent) scalable, cheap
  - Bad: trade-off between scalability and precision

Scheduling For Your System

- Apply the “supermarket” technique, scheduling on the machine with the least jobs in queue
- Beware of FIFO! Consider preemption and round-robin scheduling
- If half-decent size estimates are possible, consider prioritizing shortest jobs
  - Make sure mistakes won’t completely stop everything else
  - If malicious behavior is possible, consider its impact though