Chapter 1

Motivating Examples of the Power of Analytical Modeling

1.1 What is Queueing Theory?

Queueing theory is the theory behind what happens when you have lots of jobs, scarce resources, and subsequently long queues and delays. It is literally the "theory of queues": what makes queues appear and how to make them go away.

Imagine a computer system, say a web server, where there is only one job. The job arrives, it uses certain resources (some CPU, some I/O), and then it departs. Given the job's resource requirements, it is very easy to predict exactly when the job will depart. There is no delay because there are no queues. If every job indeed got to run on its own computer, there would be no need for queueing theory. Unfortunately, that is rarely the case.

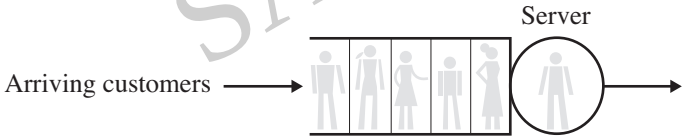

Figure 1.1: *Illustration of a queue, in which customers wait to be served, and a server. The picture shows one customer being served at the server and five others waiting in the queue.*

Queueing theory applies anywhere that queues come up (see Figure 1.1). We have all had the experience of waiting in line at the bank, wondering why there are not more tellers, or waiting in line at the supermarket, wondering why the express lane

is for 8 items or less rather than 15 items or less, or whether it might be best to actually have *two* express lanes, one for 8 items or less and the other for 15 items or less. Queues are also at the heart of any computer system. Your CPU uses a time-sharing scheduler to serve a queue of jobs waiting for CPU time. A computer disk serves a queue of jobs waiting to read or write blocks. A router in a network serves a queue of packets waiting to be routed. The router queue is a finite capacity queue, in which packets are dropped when demand exceeds the buffer space. Memory banks serve queues of threads requesting memory blocks. Databases sometimes have lock queues, where transactions wait to acquire the lock on a record. Server farms consist of many servers, each with its own queue of jobs. The list of examples goes on and on.

The goals of a queueing theorist are twofold. The first is *predicting* the system performance. Typically this means predicting mean delay or delay variability or the probability that delay exceeds some Service Level Agreement (SLA). However, it can also mean predicting the number of jobs that will be queueing or the mean number of servers being utilized (e.g., total power needs), or any other such metric. Although prediction is important, an even more important goal is finding a superior system *design* to improve performance. Commonly this takes the form of capacity planning, where one determines which additional resources to buy to meet delay goals (e.g., is it better to buy a faster disk, or a faster CPU, or to add a second slow disk). Many times, however, without buying any additional resources at all, one can improve performance by just deploying a smarter scheduling policy or different routing policy to reduce delays. Given the importance of smart scheduling in computer systems, all of Part VII of this book is devoted to understanding scheduling policies.

Queueing theory is built on a much broader area of mathematics called stochastic modeling and analysis. Stochastic modeling represents the service demands of jobs and the interarrival times of jobs as random variables. For example, the CPU requirements of UNIX processes might be modeled using a Pareto distribution [85], whereas the arrival process of jobs at a busy web server might be well modeled by a Poisson process with Exponentially distributed inter-arrival times. Stochastic models can also be used to model dependencies between jobs, as well as anything else that can be represented as a random variable.

Although it is generally possible to come up with a stochastic model that adequately represents the jobs or customers in a system and its service dynamics, these stochastic models are not always analytically tractable with respect to solving for performance. As we discuss in Part IV, *Markovian assumptions*, such as assuming Exponentially distributed service demands or a Poisson arrival process, greatly simplify the analysis; hence much of the existing queueing literature relies on such Markovian assumptions. In many cases these are a reasonable approximation. For

example, the arrival process of book orders on Amazon might be reasonably well approximated by a Poisson process, given that there are many independent users, each independently submitting requests at a low rate (although this all breaks down when a new Harry Potter book comes out). However, in some cases Markovian assumptions are very far from reality; for example, in the case in which service demands of jobs are highly variable or are correlated.

While many queueing texts downplay the Markovian assumptions being made, this book does just the opposite. Much of my own research is devoted to demonstrating the impact of workload assumptions on correctly predicting system performance. I have found many cases where making simplifying assumptions about the workload can lead to very inaccurate performance results and poor system designs. In my own research, I therefore put great emphasis on integrating measured workload distributions into the analysis. Rather than trying to hide the assumptions being made, this book *highlights* all assumptions about workloads. We discuss specifically whether the workload models are accurate and how our model assumptions affect performance and design, as well as look for more accurate workload models. In my opinion, a major reason why computer scientists are so slow to adopt queueing theory is that the standard Markovian assumptions often don't fit. However, there are often ways to work around these assumptions, many of which are shown in this book, such as using phase-type distributions and matrix-analytic methods, introduced in Chapter 21.

1.2 Examples of the Power of Queueing Theory

The remainder of this chapter is devoted to showing some concrete examples of the power of queueing theory. Do *not* expect to understand everything in the examples. The examples are developed in much greater detail later in the book. Terms like "Poisson process" that you may not be familiar with are also be explained later in the book. These examples are just here to highlight the types of lessons covered in this book.

As stated earlier, one use of queueing theory is as a *predictive tool*, allowing one to predict the performance of a given system. For example, one might be analyzing a network, with certain bandwidths, where different classes of packets arrive at certain rates and follow certain routes throughout the network simultaneously. Then queueing theory can be used to compute quantities such as the mean time that packets spend waiting at a particular router i, the distribution on the queue buildup at router i, or the mean overall time to get from router A to router B in the network.

We now turn to the usefulness of queueing theory as a *design tool* in choosing the

best system design to minimize response time. The examples that follow illustrate that system design is often a *counterintuitive* process.

Design Example 1 – Doubling Arrival Rate

Consider a system consisting of a single CPU that serves a queue of jobs in First-Come-First-Served (FCFS) order, as illustrated in Figure 1.2. The jobs arrive according to some random process with some average arrival rate, say $\lambda = 3$ jobs per second. Each job has some CPU service requirement, drawn independently from some distribution of job service requirements (we can assume any distribution on the job service requirement for this example). Let's say that the average service rate is $\mu = 5$ jobs per second (i.e., each job on average requires $1/5$ of a second of service). Note that the system is not in overload ($3 < 5$). Let $\mathbf{E}[T]$ denote the mean response time of this system, where response time is the time from when a job arrives until it completes service, a.k.a. sojourn time.

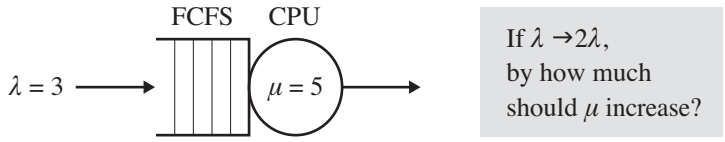

Figure 1.2: *A system with a single CPU that serves jobs in FCFS order.*

Question: Your boss tells you that starting tomorrow the arrival rate will double. You are told to buy a faster CPU to ensure that jobs experience the same mean response time, $\mathbf{E}[T]$. Response time is the time from when a job arrives until it completes. That is, customers should not notice the effect of the increased arrival rate. By how much should you increase the CPU speed? (a) Double the CPU speed; (b) More than double the CPU speed; (c) Less than double the CPU speed.

Answer: (c) Less than double.

Question: Why not (a)?

Answer: It turns out that doubling CPU speed together with doubling the arrival rate will generally result in cutting the mean response time in half! We prove this in Chapter 13. Therefore, the CPU speed does not need to double.

Question: Can you immediately see a rough argument for this result that does not involve any queueing theory formulas? What happens if we double the service rate and double the arrival rate?

Answer: Imagine that there are two types of time: Federation time and Klingon time. Klingon seconds are faster than Federation seconds. In fact, each Klingon second is equivalent to a half-second in Federation time. Now, suppose that in the Federation, there is a CPU serving jobs. Jobs arrive with rate λ jobs per second and are served at some rate μ jobs per second. The Klingons steal the system specs and reengineer the same system in the Klingon world. In the Klingon system, the arrival rate is λ jobs per Klingon second, and the service rate is μ jobs per Klingon second. Note that both systems have the same mean response time, $\mathbf{E}[T]$, except that the Klingon system response time is measured in Klingon seconds, while the Federation system response time is measured in Federation seconds. Consider now that Captain Kirk is observing both the Federation system and the Klingon reengineered system. From his perspective, the Klingon system has twice the arrival rate and twice the service rate; however, the mean response time in the Klingon system has been halved (because Klingon seconds are half-seconds in Federation time).

Question: Suppose the CPU employs time sharing service order (known as Processor-Sharing, or PS for short), instead of FCFS. Does the answer change?

Answer: No. The same basic argument still works.

Design Example 2 – Sometimes "Improvements" Do Nothing

Consider the batch system shown in Figure 1.3. There are always $N = 6$ jobs in this system (this is called the multiprogramming level). As soon as a job completes service, a new job is started (this is called a "closed" system). Each job must go through the "service facility." At the service facility, with probability $1/2$ the job goes to server 1, and with probability $1/2$ it goes to server 2. Server 1 services jobs at an average rate of 1 job every 3 seconds. Server 2 also services jobs at an average rate of 1 job every 3 seconds. The distribution on the service times of the jobs is irrelevant for this problem. Response time is defined as usual as the time from when a job first arrives at the service facility (at the fork) until it completes service.

Question: You replace server 1 with a server that is twice as fast (the new server services jobs at an average rate of 2 jobs every 3 seconds). Does this "improvement" affect the average response time in the system? Does it affect the throughput? (Assume that the routing probabilities remain constant at $1/2$ and $1/2$.)

Answer: Not really. Both the average response time and throughput are hardly affected. This is explained in Chapter 7.

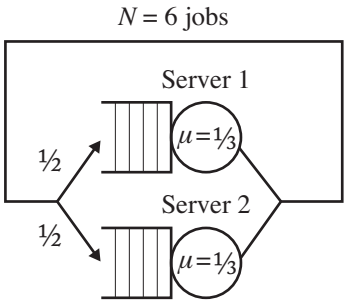

Figure 1.3: *A closed system.*

Question: Suppose that the system had a higher multiprogramming level, N. Does the answer change?

Answer: No. The already negligible effect on response time and throughput goes to zero as N increases.

Question: Suppose the system had a lower value of N. Does the answer change?

Answer: Yes. If N is sufficiently low, then the "improvement" helps. Consider, for example, the case $N = 1$.

Question: Suppose the system is changed into an open system, rather than a closed system, as shown in Figure 1.4, where arrival times are independent of service completions. Now does the "improvement" reduce mean response time?

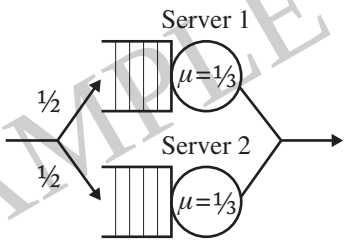

Figure 1.4: *An open system.*

Answer: Absolutely!

Design Example 3 – One Machine or Many?

You are given a choice between one fast CPU of speed s, or n slow CPUs each of speed s/n (see Figure 1.5). Your goal is to minimize mean response time. To start, assume that jobs are *non-preemptible* (i.e., each job must be run to completion).

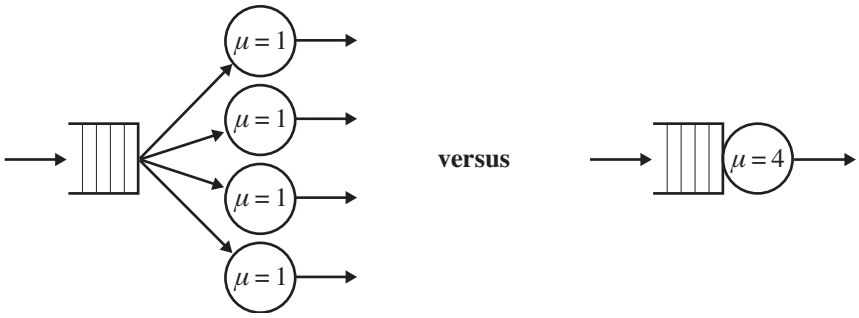

Figure 1.5: *Which is better for minimizing mean response time: many slow servers or one fast server?*

Question: Which is the better choice: one fast machine or many slow ones?

Hint: Suppose that I tell you that the answer is, "It depends on the workload." What aspects of the workload do you think the answer depends on?

Answer: It turns out that the answer depends on the variability of the job size distribution, as well as on the system load.

Question: Which system do you prefer when job size variability is high?

Answer: When job size variability is high, we prefer many slow servers because we do not want short jobs getting stuck behind long ones.

Question: Which system do you prefer when load is low?

Answer: When load is low, not all servers will be utilized, so it seems better to go with one fast server.

These questions are revisited many times throughout the book.

Question: Now suppose we ask the same question, but jobs are *preemptible*, that is, they can be stopped and restarted where they left off. When do we prefer many slow machines as compared to a single fast machine?

Answer: If your jobs are preemptible, you could always use a single fast machine to simulate the effect of n slow machines. Hence a single fast machine is at least as good.

The question of many slow servers versus a few fast ones has huge applicability in a wide range of areas, because anything can be viewed as a resource, including CPU, power, and bandwidth.

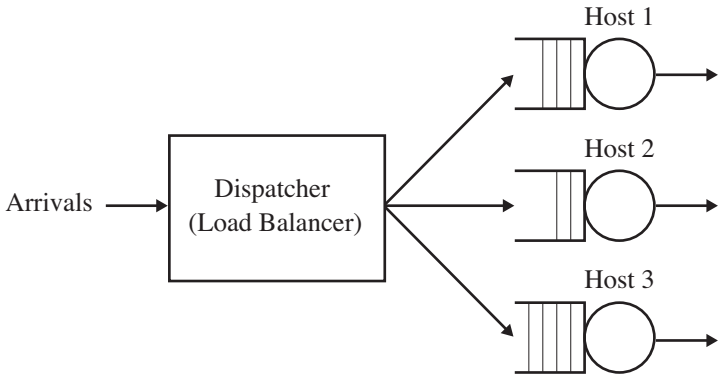

Figure 1.6: *A distributed server system with a central dispatcher.*

For an example involving power management in data centers, consider the problem from [70] where you have a fixed power budget P and a server farm consisting of n servers. You have to decide how much power to allocate to each server, so as to minimize overall mean response time for jobs arriving at the server farm. There is a function that specifies the relationship between the power allocated to a server and the speed (frequency) at which it runs – generally, the more power you allocate to a server, the faster it runs (the higher its frequency), subject to some maximum possible frequency and some minimum power level needed just to turn the server on. To answer the question of how to allocate power, you need to think about whether you prefer many slow servers (allocate just a little power to every server) or a few fast ones (distribute all the power among a small number of servers). In [70], queueing theory is used to optimally answer this question under a wide variety of parameter settings.

As another example, if bandwidth is the resource, we can ask when it pays to partition bandwidth into smaller chunks and when it is better not to. The problem is also interesting when performance is combined with price. For example, it is often cheaper (financially) to purchase many slow servers than a few fast servers. Yet in some cases, many slow servers can consume more total power than a few fast ones. All of these factors can further influence the choice of architecture.

Design Example 4 – Task Assignment in a Server Farm

Consider a server farm with a central dispatcher and several hosts. Each arriving job is immediately dispatched to one of the hosts for processing. Figure 1.6 illustrates such a system.

Server farms like this are found everywhere. Web server farms typically deploy

a front-end dispatcher like Cisco's Local Director or IBM's Network Dispatcher. Supercomputing sites might use LoadLeveler or some other dispatcher to balance load and assign jobs to hosts.

For the moment, let's assume that all the hosts are identical (homogeneous) and that all jobs only use a single resource. Let's also assume that once jobs are assigned to a host they are processed there in FCFS order and are non-preemptible.

There are many possible *task assignment policies* that can be used for dispatching jobs to hosts. Here are a few:

Random: Each job flips a fair coin to determine where it is routed.

Round-Robin: The ith job goes to host $i \mod n$, where n is the number of hosts, and hosts are numbered $0, 1, \ldots, n-1$.

Shortest-Queue: Each job goes to the host with the fewest number of jobs.

Size-Interval-Task-Assignment (SITA): "Short" jobs go to the first host, "medium" jobs go to the second host, "long" jobs go to the third host, etc., for some definition of "short," "medium," and "long."

Least-Work-Left (LWL): Each job goes to the host with the least total remaining work, where the "work" at a host is the sum of the sizes of jobs there.

Central-Queue: Rather than have a queue at each host, jobs are pooled at one central queue. When a host is done working on a job, it grabs the first job in the central queue to work on.

Question: Which of these task assignment policies yields the lowest mean response time?

Answer: Given the ubiquity of server farms, it is surprising how little is known about this question. If job size variability is low, then the LWL policy is best. If job size variability is high, then it is important to keep short jobs from getting stuck behind long ones, so a SITA-like policy, which affords short jobs isolation from long ones, can be far better. In fact, for a long time it was believed that SITA is always better than LWL when job size variability is high. However, it was recently discovered (see [91]) that SITA can be far worse than LWL even under job size variability tending to infinity. It turns out that other properties of the workload, including load and fractional moments of the job size distribution, matter as well.

Question: For the previous question, how important is it that we know the size of jobs? For example, how does LWL, which requires knowing job size, compare with Central-Queue, which does not?

Answer: Actually, most task assignment policies do not require knowing the size of jobs. For example, it can be proven by induction that LWL is equivalent to Central-Queue. Even policies like SITA, which by definition are based on knowing the job size, can be well approximated by other policies that do not require knowing the job size; see [83].

Question: Now consider a different model, in which jobs are preemptible. Specifically, suppose that the servers are Processor-Sharing (PS) servers, which time-share among all the jobs at the server, rather than serving them in FCFS order. Which task assignment policy is preferable now? Is the answer the same as that for FCFS servers?

Answer: The task assignment policies that are best for FCFS servers are often a disaster under PS servers. In fact, for PS servers, the Shortest-Queue policy is near optimal ([80]), whereas that policy is pretty bad for FCFS servers if job size variability is high.

There are many open questions with respect to task assignment policies. The case of server farms with PS servers, for example, has received almost no attention, and even the case of FCFS servers is still only partly understood. There are also many other task assignment policies that have not been mentioned. For example, *cycle stealing* (taking advantage of a free host to process jobs in some other queue) can be combined with many existing task assignment policies to create improved policies. There are also other metrics to consider, like minimizing the variance of response time, rather than mean response time, or maximizing fairness. Finally, task assignment can become even more complex, and more important, when the workload changes over time.

Task assignment is analyzed in great detail in Chapter 24, after we have had a chance to study empirical workloads.

Design Example 5 – Scheduling Policies

Suppose you have a *single* server. Jobs arrive according to a Poisson process. Assume anything you like about the distribution of job sizes. The following are some possible service orders (scheduling orders) for serving jobs:

First-Come-First-Served (FCFS): When the server completes a job, it starts working on the job that arrived earliest.

Non-preemptive Last-Come-First-Served (LCFS): When the server completes a job, it starts working on the job that arrived last.

Random: When the server completes a job, it starts working on a random job.

Question: Which of these non-preemptive service orders will result in the lowest mean response time?

Answer: Believe it or not, they all have the same mean response time.

Question: Suppose we change the non-preemptive LCFS policy to a Preemptive-LCFS policy (PLCFS), which works as follows: Whenever a new arrival enters the system, it immediately preempts the job in service. How does the mean response time of this policy compare with the others?

Answer: It depends on the variability of the job size distribution. If the job size distribution is at least moderately variable, then PLCFS will be a huge improvement. If the job size distribution is hardly variable (basically constant), then PLCFS policy will be up to a factor of 2 worse.

We study many counterintuitive scheduling theory results toward the very end of the book, in Chapters 28 through 33.

More Design Examples

There are many more questions in computer systems design that lend themselves to a queueing-theoretic solution.

One example is the notion of a *setup cost*. It turns out that it can take both significant time and power to turn *on* a server that is *off*. In designing an efficient power management policy, we often want to leave servers *off* (to save power), but then we have to pay the setup cost to get them back on when jobs arrive. Given performance goals, both with respect to response time and power usage, an important question is whether it pays to turn a server off. If so, one can then ask exactly how many servers should be left on. These questions are discussed in Chapters 15 and 27.

There are also questions involving optimal scheduling when jobs have priorities (e.g., certain users have paid more for their jobs to have priority over other users' jobs, or some jobs are inherently more vital than others). Again, queueing theory is very useful in designing the right priority scheme to maximize the value of the work completed.

However, queueing theory (and more generally analytical modeling) is *not* currently all-powerful! There are lots of very simple problems that we can at best only analyze approximately. As an example, consider the simple two-server network shown in Figure 1.7, where job sizes come from a general distribution. No

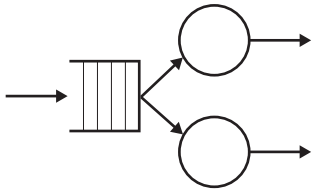

Figure 1.7: *Example of a difficult problem: The M/G/2 queue consists of a single queue and two servers. When a server completes a job, it starts working on the job at the head of the queue. Job sizes follow a general distribution, G. Even mean response time is not well understood for this system.*

one knows how to derive mean response time for this network. Approximations exist, but they are quite poor, particularly when job size variability gets high [77]. We mention many such open problems in this book, and we encourage readers to attempt to solve these!