The Essential Guide To
Queueing Theory

$$P(W_q > 0) = \cfrac{\dfrac{A^M}{M!}\dfrac{M}{M-A}}{\left(\sum\limits_{i=1}^{M-1}\dfrac{A^i}{i!}\right) + \dfrac{A^M}{M!}\dfrac{M}{M-A}}$$

W_q

ρ
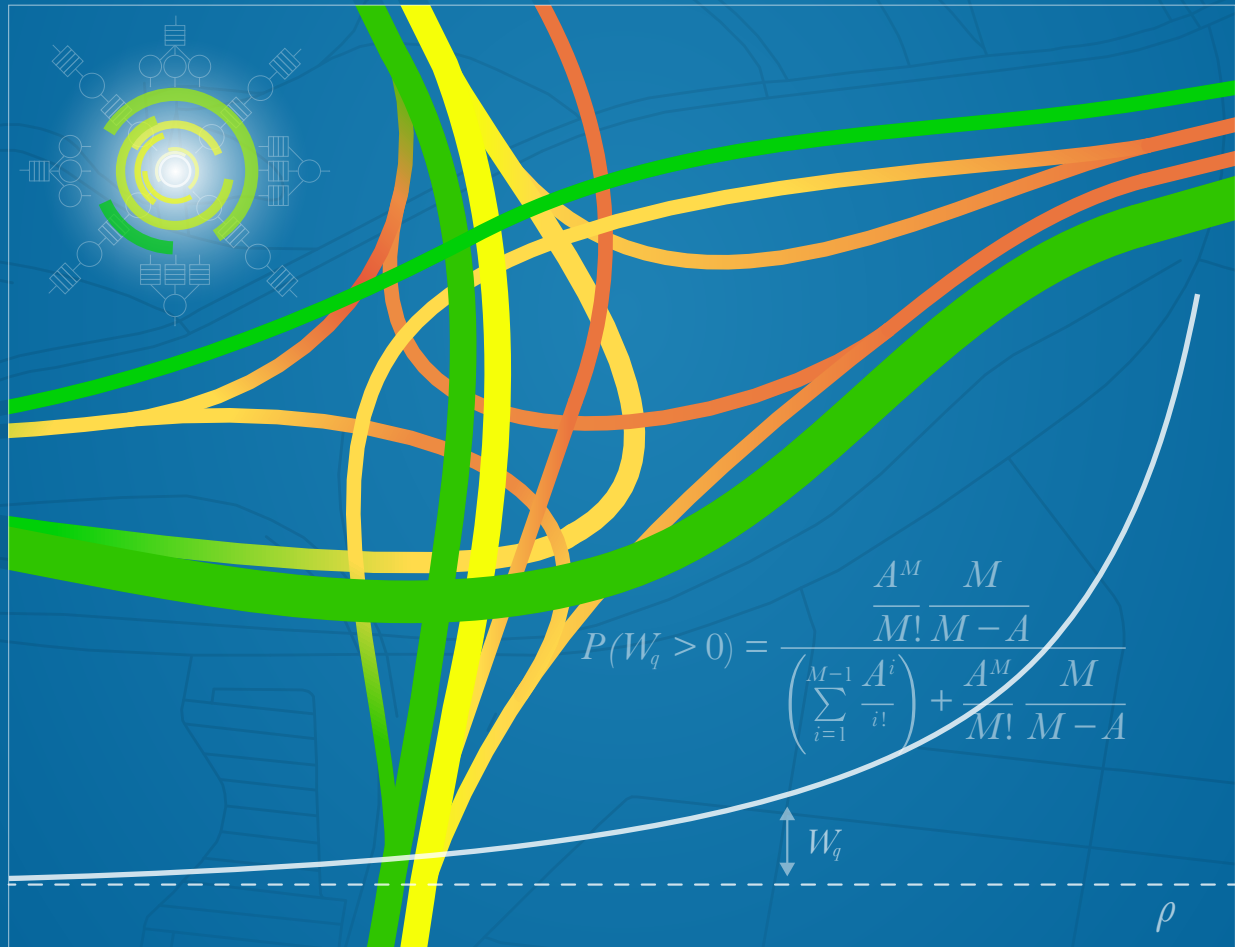
September 8, 2016 • Revision 3

VividCortex

Meet the Author

Baron Schwartz

Baron is well-known for his contributions to the MySQL, PostgreSQL, and Oracle communities. Baron has helped build and optimize database systems for some of the largest Internet properties. He has written several books, including O'Reilly's best-selling High Performance MySQL. Prior to founding VividCortex, Baron managed consulting, support, training, and software engineering at Percona. He has a CS degree from the University of Virginia.

Table of Contents

Queueing Theory is for Everyone

Whether you're an entrepreneur, engineer, or manager, learning about queueing theory is a great way to be more effective. Queueing theory is fundamental to getting good return on your efforts. That's because the results your systems and teams produce are heavily influenced by how much waiting takes place, and waiting is waste. Minimizing this waste is extremely important. It's one of the biggest levers you will find for improving the cost and performance of your teams and systems.

Unfortunately, queueing theory books are often terribly complicated and boring. But it doesn't have to be that way! The concepts are easy to understand, so developing intuition is low-hanging fruit. Plus, queueing theory is super-cool, and it's a pity if you don't learn about it.

There are plenty of 350-page books about queueing theory, and lots of high-level introductions too. But these all either dive deep into pages full of equations and Greek symbols, or gloss over (or just outright lie about) the big, important truths of queueing. We need a middle ground.

My hope is that this book can help you achieve the following:

- Develop awareness of how queueing appears in everyday life. Queueing rules everything around you, but many people probably aren't aware of it.

- Build intuition of important concepts and truths that can guide you in decision making.

- Help you see where your intuition is a trusty guide and where queueing will surprise you.

- Help you understand the landscape of queueing theory from a general framework, so you can quickly navigate more detailed treatises and find the right answers fast.

Armed with the knowledge in this book, you will be able to quickly understand common scenarios such as:

- If web traffic increases 30%, will the server get overloaded?

- Should I buy faster disks, or should I just buy a few more of what I already have?

- What if I specialize tasks in my team and create a handoff between roles?

Knowing how to think about these kinds of questions will help you anticipate bottlenecks. As a result, you'll build your systems and teams to be more efficient, more scalable, higher performance, lower cost, and ultimately provide better service both for yourself and for your customers. And when you need to calculate exactly how much things will bottleneck and how much performance will suffer, you'll know how to do that too.

As for scope, the aim of this book is to help you get answers for common scenarios and point you to the right resources for other cases. I'll go into details where they help illuminate important concepts.

What is Queueing Theory?

Queueing theory is a broad field of study that focuses on the many nuances of how waiting lines behave. It's a fascinating topic if you're of a theoretical bent. But it's highly relevant even if you're the impatient "just answer my question!" type, too.

Queueing theory boils down to answering simple questions like the following:

- How likely is it that things will queue up and wait in line?

- How long will the line be?

- How long will the wait be?

- How busy will the server/person/system servicing the line be?

- How much capacity is needed to meet an expected level of demand?

You can rearrange these questions easily to answer other questions, such as "how likely am I to lose business due to too-long waits or not enough capacity?" and "how much more demand can we satisfy without creating an unacceptably long wait?"

And although I wrote the questions in simple terms, you can also answer sophisticated questions about the characteristics of the answers. For example, "how long will the wait be?" can be answered not only on average, but you can calculate things such as the variability of wait times, the distribution of wait times and how likely it is that someone will get extremely poor service, and so on.

Why Queueing Theory is Tricky

Bad news: queueing theory is based on probability, which is hard and weird. As Eric Bowman tweeted, "Not even God understands queueing theory."[1] Nothing leads you astray faster than trusting your intuition about probability. If your intuition worked, casinos would all go out of business, and your insurance rates would seem reasonable.

In particular, human intuition is *linear*. Double the traffic, we think, and other things will double too. *Wrong!* Queueing systems are *nonlinear*. What's worse is that they're unpredictably nonlinear. Things seem to change gradually, until you reach some tipping point and *boom*, everything gets really bad, really fast. These transitions can be so sudden that they are very surprising. Our intuition isn't geared to expect these disproportionate, sudden changes.

....

[1] https://twitter.com/ebowman/status/728503842611077121

The classic example you might recognize is the so-called hockey-stick curve. This is the most famous queueing theory picture of all time:
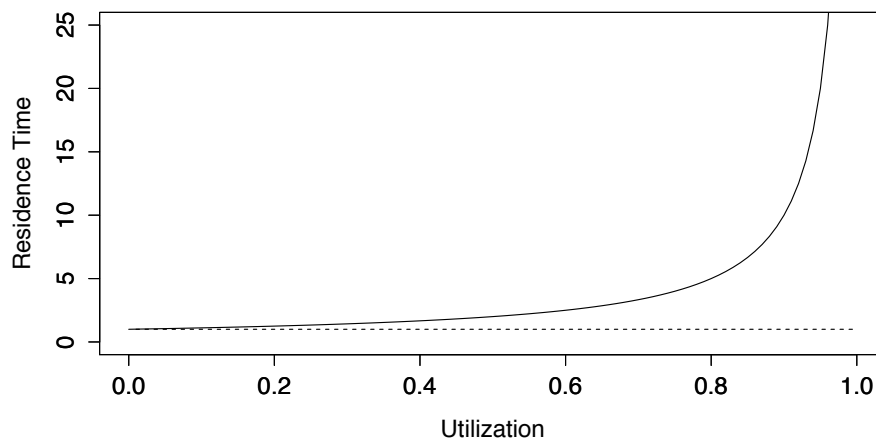
This curve shows how a system's total response time increases as utilization increases. As utilization approaches 100%, response time goes to infinity. The exact shape of this curve depends on the characteristics of the system and the work it's doing, but the general principle applies to pretty much all systems.

You'll get familiar with nonlinearity over the course of this book. You might never be able to accurately estimate *how much* things will be nonlinear. But you'll get pretty good at recognizing situations where someone needs to check whether bad stuff is going to happen. Then you'll do a little bit of math or look things up in a table and find the answers you need.

Queueing Systems in Real Life

Most introductions to queueing theory talk about abstract things and use jargon a lot. They'll usually discuss some real-life scenarios to help you understand the abstract things, but for some reason they always seem to pick bizarre situations I personally have never encountered.[1] Then they

---

[1]   Often with British terminology, something about hiring lorries and abseiling.

VividCortex

jump into discussions of Markovian something or other, and talk about births and deaths. At this point my eyes have usually glazed over.

But life is full of queueing systems!

- Your Kanban board

- Getting a latte at the coffee shop

- The line for the restroom after drinking the latte

- Your computer's CPUs and disks

- At the traffic light

- When the captain says you're third in line for takeoff

- Your load balancer

- Pre-ordering the new iPhone

- Trying to pick the best line at the grocery checkout counter

- Merging onto the freeway

- Connection pooling

Importantly, queueing systems are present in much more subtle scenarios too. Any process for handling units of work has queueing systems built into it. When you ask your graphic designer to produce a beautiful new ebook, for example, queueing is at work.

What is the real meaning of adding a queue into a system? At a higher level, although the mechanics of queues are simple, their ultimate results are often overlooked or misunderstood. To co-opt the old joke, the programmer with a problem thought, "I know, I'll just add a queue." (Now the programmer has unbounded problems.) Let's look at some of the consequences and purposes of queues.[1]

---

[1]   I'm speaking here about queues as a concept, not specific technologies such as RabbitMQ.

Fundamentally, **a queue is a mechanism for increasing availability**. A server is available when it can accept and service a request. In the formal sense, a server is either busy or idle at any point in time. This doesn't match most people's intuition, which is built on concepts such as percent utilization and backlog. But when you take away the queue or buffer, and look at the thing that handles the request, it's either busy working on a request (unavailable for new requests that arrive), or it's idle and ready to accept a request (available). Putting a queue in front of it makes the combined system available for new requests, which otherwise would have nowhere to go and would be rejected.

In other words, **a queue is a lie that says the system is available, when in fact it's not available but might be in the future.**

A consequence of increasing availability is that queues increase utilization. How so? The utilization improvement follows from work being ready to do. If requests can queue, then the system can move from servicing one request to the next with no idle time in between. The alternative, with no queue, is waste and lower utilization. To see why this is so, imagine there's no queue to hold ready-to-do work. Now, newly arriving requests will be rejected, so when the system is finished with its current request, it'll lie idle until the next one happens to arrive. As a result, **a queue is a mechanism for improving system utilization**.

Because queues in front of servers increase their availability and utilization, they also improve their throughput. Throughput is the rate at which work is completed, and without going into formalities at this point, when the server is more utilized it's completing requests back-to-back more often. This increases the rate at which completed requests leave the system.[1]

All good so far—but what's the downside? Latency. **Queues add latency.** Latency is the sum of the time the requests spend waiting in the queue

---

[1] You may have heard that queues don't improve throughput. I've seen this claim, too, but only from people who are confusing the definition of throughput.

and the time they spend being serviced. Intuitively, if they're waiting in the queue, they're waiting longer overall. (This intuition is correct.) Another effect of a queueing system is that keeping the queue itself consumes some resources. In a computer system, queues take resources such as memory and processor cycles; in a grocery store, they take floorspace.

To sum up, then, **a queue is a mechanism that can improve availability, utilization, and throughput, but at the expense of latency and resource consumption**. Queues have other properties too—such as helping add resilience to systems—that we'll examine later. But these are the essential truths.

These principles apply everywhere in queueing systems, which are all around us in real life as well as in the computer systems we're building.

One of the big wins is being able to recognize these systems, particularly in *teams*. Teams have communication structures, and communication—whether it's email or chat at the water cooler—is queueing. Send an email to your colleague; when it's in their inbox it's queued. If a developer doesn't know how to optimize a query and asks a DBA for help, and the DBA is busy, the developer's request is queued. Design your organization with the recognition that *teams are systems too*, and you'll have a chance to get much better results.

Why Does Queueing Happen?

Why do things wait in queues anyway?

It might seem like a dumb question with an obvious answer. "Because there's more work to do than capacity to do it, silly!" But that is not the right answer. Queueing happens even when there's more than enough capacity to do the work. This is part of what's really surprising and counterintuitive about queueing theory. This is so important that it needs

to be emphasized: **Queueing happens even when there's more than enough capacity to do the work.**

Let's take a simple example. Suppose a grocery store has a single checkout line and a single cashier. Suppose an average of one shopper per minute arrives at the line to pay for their groceries. Scanning, bagging and paying takes 45 seconds on average. Will there be queueing and waiting?

Intuition says no, there won't. There will be, on average, 15 seconds of idle time for the cashier every minute. The cashier is only busy 75% of the time! Of course no one has to wait in line, right?

But that's not what really happens. In reality, there are lots of shoppers waiting in line and they have to wait a long time!

Why on earth would this happen? It seems to make no sense at all.

Fundamentally, queueing happens because of three reasons:

1. **Irregular arrivals.** Shoppers don't arrive at the checkout line on a regular schedule. They're sometimes spaced far apart and sometimes close together, so they overlap. An overlap automatically causes queueing and waiting.

2. **Irregular job sizes.** Shoppers don't all complete in 45 seconds. Some of them will take much longer. And when this happens, there's overlap again because new shoppers will arrive and be ready to check out while the existing ones are still in progress.

3. **Waste.** Lost time can never be regained. Shoppers overlap because earlier shoppers didn't have time to finish. But if you look at it another way, perhaps it's not the second shopper's fault. Perhaps the first shopper should have arrived earlier, but wasted time doing something else while the cashier was idle! They missed their chance and made the second shopper have to wait. When the cashier is idle, time is

wasting, and can never be gotten back because the cashier can't be more than 100% utilized. Everything gets delayed permanently, and that shows up as queueing for later arrivals.

In general, irregular arrival times and irregular job sizes are guaranteed to cause queueing. The only time there's no queueing is when the job sizes are uniform, the arrivals are timed evenly, and there's not too much work for the cashier to keep up. Even when the cashier is not busy at all, irregular or bursty arrivals will cause some queueing.

Queueing gets worse when the following are true:

- **High Utilization.** The busier the cashier is, the longer it takes to recover from wasted time.

- **High Variability.** The more variability[1] in arrivals or job sizes, the more waste and the more overlap (queueing) occurs.

- **Few Servers.** Fewer cashiers means less capacity to absorb spikes of arrivals, leading to more wasted time and higher utilization.

The Queueing Theory Framework

All discussions of queueing theory analyze systems and processes in terms of three key concepts:

- **Customers** are the units of work that the system serves. A customer can be a real person, or it can be whatever the system is supposed to process and complete: a web request, a database query, a part to be milled by a machine.

- **Servers** are the things that do the work. This might be the cashier at

---

[1] If you've read Eli Goldratt's *The Goal*, you should recognize the phrase "statistical variations." This book is all about queueing theory even though it doesn't mention it explicitly. The same is true for *The Phoenix Project* by Gene Kim et al.

VividCortex

the grocery store, the web server or database server,[1] or the milling machine.

- **Queues** are where the units of work wait if the server is busy and can't do the work as it arrives.

These are abstract concepts that can be extended to every situation. There are also variations on the names. For example, servers are sometimes called "stations," and sometimes I'll call a customer a "job," a "request," or a "task" in this book.

Graphically, queues are represented as a set of stacked boxes, and servers/stations are shown as circles. The simplest possible system is a single queue leading to a single server:

However, it's possible to have a variety of configurations, such as 1-to-many, many-to-many, many one-to-one, and chained systems:

---
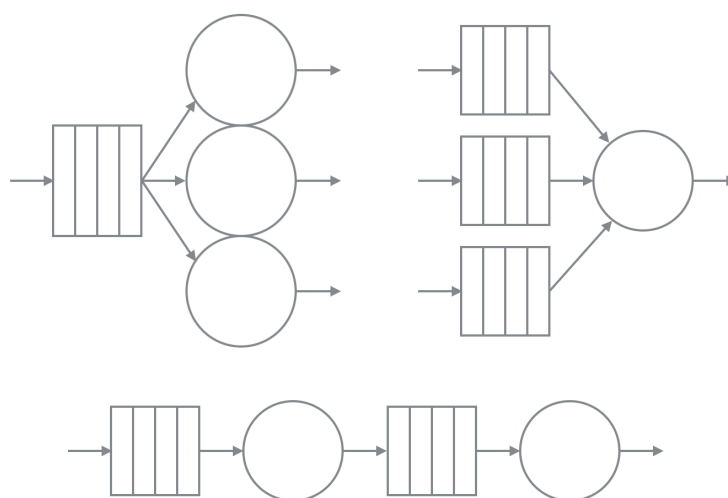
[1]   or its CPU, for that matter.

Queueing theory views every system as interconnected sets of servers and queues, through which customers flow. Once you're aware of this framework, you can decompose any system into networks of queues and servers and analyze them individually or separately.

Now that you know that, what can you measure, analyze and predict about a queueing system? There are a variety of important metrics and parameters:

Metrics	Units	Symbol	Description
Arrival Rate	Customers per time	λ or A	How often new customers arrive at the front of the queue. In a stable system that will reach equilibrium, this is the same as the completion rate, also called the throughput (sometimes denoted X).
Queue Length	Customers	L_q	The average number of customers waiting in the queue.
Wait Time	Time	W_q or W	How long customers have to wait in the queue, on average.
Service Time	Time	S	How long it takes to service customers after they leave the queue, on average.
Service Rate	Services per time	μ	This is the inverse of service time.
Residence Time	Time	R	How long customers are in the system overall, on average. In computer systems we usually call this latency or response time. It's the sum of wait time and service time.
Utilization	Fraction	ρ or U	How much of the time the system is busy serving requests. If the system has multiple servers, it's an average over all of them. This ranges from 0 to 1, inclusive.
Concurrency	Customers	L	The number of customers waiting or in service, on average
Number of Servers	Servers	M	The number of servers the system has (e.g. cashiers, CPUs).

You will find many different names used to describe some of these. There is no universal standard for naming or symbols, and I have only listed

some of the most common ones. You will find others in use, such as sojourn times and delays, sometimes ambiguously. You'll also find different letters and symbols; for example, sometimes s is the symbol for the number of servers.

The results you'll get from queueing theory are mostly about long-term averages in stable systems that reach equilibrium. A stable system is one that is able to keep up with all the incoming work, so all customers eventually get serviced, and queues don't grow to infinity.

Many of these parameters are related through a set of simple laws that are easy to understand. (Parts of queueing theory actually do make sense!) The most important of these is Little's Law, which explains the relationship between a few of the key metrics. It is important to note that Little's Law applies only to stable systems. In general, and with no further assumptions needed about any of the metrics, Little's Law states that

$$L = \lambda R$$

In words, concurrency (the number of customers in the system) is arrival rate[1] times residence time. This relationship can be solved for different variables and used to analyze all or part of a queueing system. For example, you can analyze just the queue:

$$L_q = \lambda W_q$$

You will find Little's Law in different guises absolutely everywhere. Another useful relationship is the Utilization Law, which states that utilization is throughput times service time:

$$\rho = \lambda S$$

Or, if you have M number of servers,

$$\rho = \frac{\lambda S}{M}$$

---

[1]   Or throughput, or departures; in a stable system these are equal in the long term.

Another way to compute utilization:

$$\rho = \frac{\lambda}{\mu}$$

Although they're simple, these formulas are pretty easy to apply incorrectly, so always be sure you double check everything. There are also some subtleties that trip me up repeatedly. For example, the queue length is by definition the number of customers in the queue only and not in service, but the wait time in the queue will include some time for the current customer in service to finish.

The utilization formula $\rho = \lambda/\mu$ is notable because it shows directly what is intuitively clear: if customers are arriving faster than they can be served, then utilization will be more than 100%, the queue will grow to infinity, and the system isn't stable.

Another extremely important equation is the relationship between residence time, wait time, and service time:

$$R = W_q + S$$

Residence time in a queueing system is usually what you'd think of as response time or latency. As this equation shows, it's the sum of two components. Here's our hockey stick diagram again:
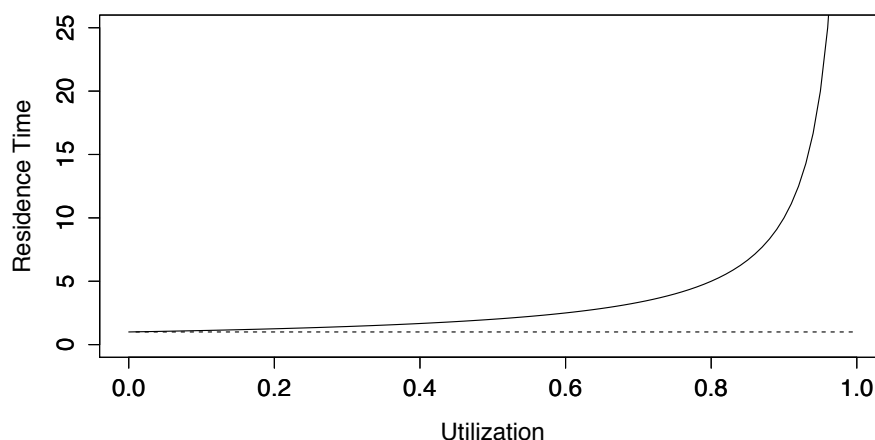
The dashed line in this chart is the service time, which is a floor on how quickly a request will complete, on average. Even with zero queueing, the service time takes some minimal amount of time. The wait time in the queue—the distance between the curve and the dashed line—is responsible for the hockey stick curve. Later I'll show you how to calculate that.

Describing Queueing Systems

As you have seen, there are different kinds of queueing systems, depending on how many queues and servers there are, and how they're connected together. Various configurations of queueing systems are typically described with Kendall's Notation, which gives a convenient shorthand for labeling classes of systems.

These are important because different kinds of systems have very different queueing behavior, which sometimes results in drastically different wait times and queue lengths. If you're going to analyze them, you'll need to be sure you know what kind of system you're working with, so you can pick the right analytical tools.

Kendall's Notation makes that easy. It is a set of slash-delimited letters and numbers that indicates the following:

- How arrival rates behave

- How service times behave

- The number of servers

- Special parameters that are only needed in unusual cases and are usually left off. Just so you know what they are, they relate to whether the system has finite capacity and might reject arrivals, how big the population of customers is, and whether the queue does anything special to select which customer gets service next.

You'll commonly see a few kinds of queueing systems in Kendall's Notation:

- M/M/1

- M/M/m (sometimes called M/M/c)

- M/G/1

- M/G/m

These letters mean the following.

- In the first position, the arrival rate is almost always specified as M, which stands for Markovian or Memoryless. This means that the customers arrive randomly and independently of each other, with average rate λ. These arrival events are said to be generated by a Poisson process, and the time between arrivals is exponentially distributed, with mean $1/\lambda$.

- The second position describes how the service times are distributed—that is, how long it takes to service each customer, with mean μ. This is often M as well, which means they are exponentially distributed, but sometimes it is G, which means the distribution is "general" and no specific distribution is assumed. Sometimes you'll see authors assume a Gaussian (normal) distribution, but it doesn't have to be Gaussian.

- In the third place, you will find the number of servers. It'll either be an exact number, such as M/M/1 where there's explicitly 1 server, or you'll see an m or c as a variable placeholder, meaning a generalized system is being discussed and you'll plug the number of servers into the analysis.

The exponential distribution is very common in queueing analysis. Not only does it seem to describe real-world systems quite often, but it has

nice mathematical properties that simplify analysis, and it's the safest bet when the real distribution is unknown. If you're not familiar with it, here's a chart of its PDF,[1] or probability density function $\lambda e^{-\lambda x}$:
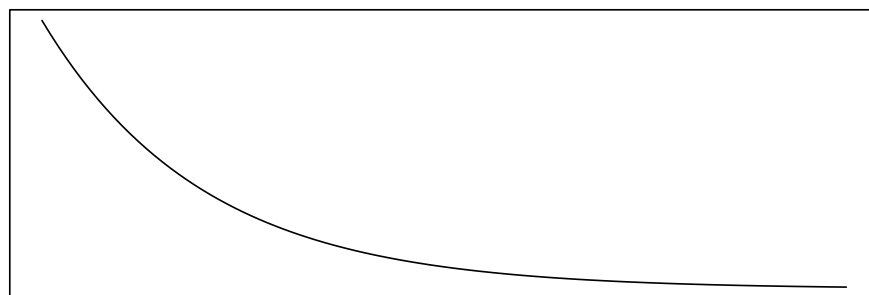

The exponential distribution's useful properties include easy-to-calculate mean, standard deviation, and quantiles. The Wikipedia page gives the formulas for these in the sidebar.

Most of the time you'll work with M/M/m queuing systems. Lots of real-life situations are known to result in independent and randomly spaced arrivals, and it's usually a pretty safe default assumption unless you know otherwise. And if you think about it, lots of real-life service times are sort of exponentially distributed too, with most jobs completing pretty quickly but occasional outliers taking a long time.

One real-world example of an exponential distribution is reaction speed when an Olympic runner hears the starting gun. Conceptually, the gunshot is the request, the runner's nervous system is the server, and starting off the blocks is the completed request. How long does this take? Intuitively you'd expect most people to have about the same reaction time, with a few being faster and slower. No one has a less-than-zero reaction time. The result is an exponential curve (with slightly different shape because of different parameters) like the following chart. This shows people's response time on a website, compared to Usain Bolt's starting time at the 2016 Rio Olympics.

[1]   The PDF is what you get when you plot a histogram of a distribution.
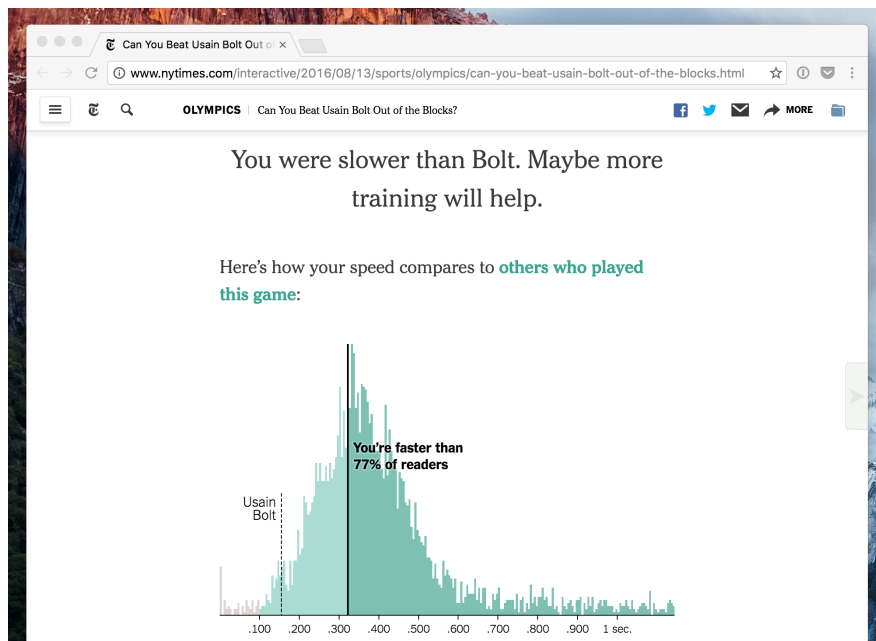
Let's think about a few kinds of queueing systems and see what they are.

- The line at my local coffee shop is M/M/2. There's a single line, and people arrive randomly and independently (well, mostly... sometimes a few people arrive together), so that's an M. Their orders usually are pretty quick, but as you know, occasionally someone has trouble making up their mind and it takes a really long time, so M seems appropriate for the second letter. And there are two cashiers.

- The security area at my local airport is best described as multiple M/M/1 queues. This applies to the ID check process as well as the bag check. In both cases I get in a line and I can't jump into an adjacent one, and there's a single person or single X-ray belt servicing me.

The M/M/m queueing system, and the exponential distribution, is sort of the moral equivalent of the Gaussian (normal) distribution in statistics. It shows up everywhere, it's nice to work with it, and it's well analyzed.

Calculating Queue Lengths and Wait Times

The usual use for queueing analysis is to answer questions about queue lengths and wait times. There are lots of other reasons to apply queueing theory, too, such as minimizing total cost of service and the like. But many of these applications center around queue lengths and wait times.

As I showed earlier, both queue lengths and wait times go to infinity as a server gets close to 100% utilization. But what, exactly, is the shape of that hockey stick curve? How do you calculate it?

Finding exact answers can be tricky, because you have to figure out what kind of system you're dealing with. In the simplest M/M/1 case, your total time checking out is:

$$R = \frac{S}{1 - \rho}$$

Therefore, your residence time is proportional to $1/(1 - \rho)$, which is known as the *stretch factor*, a normalized measure of how "stretched" your residence time is relative to your service time. It is easy to see that this is a fast-growing curve; in fact it's the hockey stick curve. You can experiment interactively with it on Desmos.

You've probably heard rules of thumb about server utilization. Even experts repeat these. For example, a very senior performance engineer was quoted recently as saying that a server running at 80% CPU utilization is "fairly active." But is it? It really depends on the number of CPUs in the server (which we'll get to later) and the desired response time. For many servers, 80% utilization is about 20% too much idle time.

Notice that the residence time doubles with every halving of $1 - \rho$. If S is 0.25, for example, then

- At 50% utilization, R is 0.5

- At 75% it is 1.0

- At 87.5% it is 2.0, and so on.

This leads to a useful rule of thumb to remember for a single-server queueing system: *residence time is inversely proportional to the server's idle capacity.* Halve the idle capacity, double the response time.

Note that if you want to determine any other desired metric about the queueing system, the laws given above make it possible. For example, if you want to calculate the queue length at a given utilization, you can find the time in the queue by just rearranging Little's Law, the Utilization Law, and the relationship between residence time, service time, and queue time. There are some subtleties that often trip people up about these—forgetting to account for time waiting for the current job to finish, for example—so I'll include them here.

The average number of requests in the system as a function of server utilization, is

$$L = \frac{\rho}{1 - \rho}$$

The average length of the queue itself, excluding requests in service, is

$$L_q = \frac{\rho^2}{1 - \rho}$$

The expected waiting time in the queue, before a request starts to be serviced, is

$$W_q = \frac{\rho S}{1 - \rho}$$

You can also approximate service times and service time distributions for specific queueing configurations. Estimating approximate response times (latencies) for a queueing system is a useful skill, and it turns out it's not difficult to do in a napkin-math sort of way that's easy to remember. It's derived from the quantiles of the exponential distribution's cumulative distribution function, $-ln(1-F)/\lambda$. You can find the details in Dr. Neil Gunther's blog post. For M/M/1 queueing systems, the heuristics are as follows:

$$R_{50} \approx \frac{2}{3}R$$

$$R_{80} \approx \frac{5}{3}R$$

$$R_{90} \approx \frac{7}{3}R$$

$$R_{95} \approx \frac{9}{3}R$$

Where the quantile (median or 50th percentile, 80th percentile, 90th percentile, 95th percentile) is approximately the corresponding fraction of the mean response time. You'll notice the denominators are all 3.

The Erlang Queueing Formulas

The simple equation $R = S/(1-\rho)$ that begins the previous section is for an M/M/1 queue with a single server. Computing queue lengths and wait times for multiple servers requires Erlang's formulas. Agner Erlang was a pioneer in telecommunications, and developed a variety of equations for predicting things like how many telephone lines would be needed to carry the expected volume of calls.

The Erlang formulas apply to a variety of different kinds of queueing systems. The one I've seen referenced most often is the Erlang C formula, because it applies to M/M/m queues. It uses an *Erlang* as a dimensionless unit of service demand or load, sometimes called the traffic intensity. An Erlang is another way of expressing average concurrency. For example, as the Wikipedia page explains, a single phone circuit can carry 60 minutes of calls in an hour; if a group of circuits receives 100 6-minute calls in an hour, the traffic intensity in that hour will be 600 minutes, or 10 Erlangs. If there are more Erlangs of demand than servers, the queue is unstable.

The Erlang C formula commonly appears in several forms. I will show only one. Given the traffic intensity A in Erlangs, the following is the probability that a customer will find all servers busy and will have to wait in the queue:

$$P(W_q > 0) = \frac{\dfrac{A^M}{M!}\dfrac{M}{M-A}}{\left(\displaystyle\sum_{i=1}^{M-1} \dfrac{A^i}{i!}\right) + \dfrac{A^M}{M!}\dfrac{M}{M-A}}$$

You can derive related forms with some substitutions and rearrangement, using Little's Law and the like. My purpose here is not to show all of these, because you can find or solve those on your own. It is to call your attention to the factorials and summation. *This equation requires iterative computations to solve.* That makes it hard to use in, for example, a spreadsheet.

The Erlang equations are correct, but they're not easy to apply. Fortunately, there's a middle ground.

Approximations to Erlang Formulas

When the Erlang formulas aren't convenient, one solution is to use precomputed tables of results, which are readily available. Another is equations that approximate the Erlang formulas, some of which are exact[1] in special cases such as single-server queueing systems.

Before I show you any formulas, let's pause for a moment and think about how a single-queue, multi-server system should behave. In this configuration, whoever's next in line can be served by *any* of the servers—whichever one becomes idle first.[2]

So when there are M servers that are all equally capable of serving a customer, then it's less likely that all of them are busy at the moment a customer arrives, and wait time in the queue should be shorter.

Recall the earlier equation for the residence time in an M/M/1 queueing system:

$$R = \frac{S}{1 - \rho}$$

One way to extend this for multiple-server queues is as follows.

$$R \approx \frac{S}{1 - \rho^M}$$

Neil Gunther explores this in detail in Chapter 2 (specifically, Section 2.7) of *Analyzing Computer System Performance With Perl::PDQ* (Springer, 2005). Intuitively, the justification for adding the exponent in the denominator is as follows:

---

[1] For example, the Pollaczek–Khinchine formula is an exact solution for the average queue length of an M/G/1 queue. You can find lots of other such formulas on Wikipedia or elsewhere.

[2] As a rule of thumb, this is applicable to things such as CPUs and coffee shop baristas. It is not applicable to situations such as disk I/O, where a request must be served by the disk that has the requested data, and can't be served by just any available disk. It's also not applicable to team situations where a person with specialized knowledge or privileges (such as a DBA or sysadmin) is required to handle a task. Hint: specialization in teams is problematic.

- When $M = 1$, it's exactly the same equation.

- When $M > 1$, the denominator becomes larger because ρ is less than 1. Thus the resulting R is smaller, which is what we expect.

Although this passes the sniff test, the results are only approximate. (They are exact for 1 and 2 servers.) This equation can underestimate the residence time by up to about 10% in some cases.

In 1977, Hirotaka Sakasegawa wrote a paper that describes the derivation and behavior of a more accurate approximation[1] for M/M/m and M/G/m queueing systems. Note that this one is for queue length, not residence time.

$$L_q \approx \frac{\rho^{\sqrt{2(M+1)}}}{1 - \rho} \times \frac{CV_{IAT}^2 + CV_{ST}^2}{2}$$

The numerator of the right-hand term is the coefficient of variation (standard deviation divided by mean) of the inter-arrival time and service time, respectively. The right-hand term disappears in M/M/m queueing systems, because the coefficient of variation of the exponential distribution is 1. (Nifty!)

Key properties of these formulas, which reinforce the major themes of this book (helping to build intuition about queueing) are:

- Increasing utilization ρ increases queue wait time.

- Increasing the number of servers M decreases queue wait time.

- Increasing variability in inter-arrival times and/or service times increases queue wait time.

As before, it is possible to use these approximations with relationships such as Little's Law to derive any desired metric.

....

[1]   See this GitHub repository for a table of results and comparisons to Gunther's heuristic formula and the Erlang C formula, as well as the paper in PDF format.

Many Queues or One Combined Queue?

Suppose you want to create greater efficiency at the airport security checkpoint. There is floorspace to queue even the largest number of travelers you would ever receive. There are 2 agents checking documents and bags. What's the most efficient configuration of lines and checking stations: a combined line, or a separate line for each agent?

If you don't know the answer to this already, I'm glad you are reading this book. I'll work through the math, assuming M/M/m queues. Suppose 240 people arrive at the airport per hour, and it takes 15 seconds to check each person, on average. Here are the results:

Metric	Combined	Separate
Utilization	50%	50%
Avg Queue Length	0.33	0.5
Avg Residence Time	20 sec	30 sec

The combined queue is better, and it always will be better. With more lines it becomes pretty dramatic. Suppose the airport's traffic grows 4-fold and they hire 4 times as many agents:

Metric	Combined	Separate
Utilization	50%	50%
Avg Queue Length	0.06	0.5
Avg Residence Time	15.24 sec	30 sec

Proper system design can create huge efficiency or waste!

This illustrates the effect of specialization. In the combined-queue configuration, we're letting travelers go to the next available agent. Agents are generalists. In the separate-queue configuration, agents are

specialized: they can only check documents for travelers in their own lines. This assumes travelers aren't allowed to skip lines and can't leave a queue after they enter it, which is a normal set of assumptions in most queueing theory analysis.

In general, having lots of servers serving each queue pushes the so-called "knee" of the hockey stick curve toward higher utilization, letting you utilize resources better. Here's an illustration of the effect:
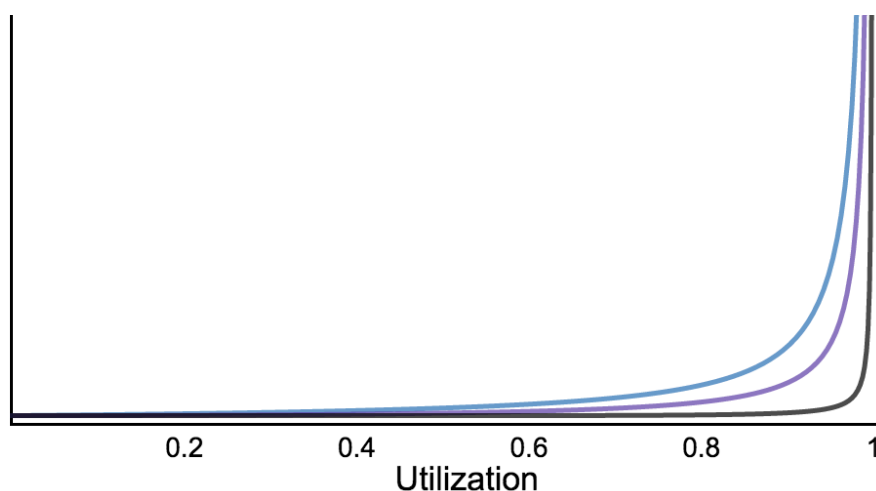
This chart plots the response time curves of systems with 1, 2, and 16 servers. The many-server curves remain flat far longer than the others. You can explore this chart on Desmos if you like.

The chart also illustrates a rule of thumb you'll probably hear a lot in certain circles: the "knee" of the response time curve.[1] It is often said that the knee is at 75% utilization, and you shouldn't run your servers past that or performance will be bad. The chart illustrates why this folklore is wrong. The knee depends on the configuration. For example, if it's a computer server with 64 processors, you'd be severely underutilizing it if you only ran it at 75% CPU utilization.

....

[1]   The location of the "knee" is completely arbitrary. One non-arbitrary definition I've seen is Cary Millsap's, which states that the knee is where the curve is tangent to a line extended from the origin. Another is that the knee occurs where the wait time in the queue is as long as the service time. In general, the knee is actually an optical illusion.

As an aside, one of the biggest wins we see for our customers at VividCortex is giving *developers* production visibility into database performance. This creates immense efficiency because it breaks down silos of access to information about database performance. That removes a bottleneck in the DevOps "queueing system" of engineers and operators communicating with each other, and creates huge return on investment.

Tradeoffs Between Cost and Quality of Service

Queueing theory explains why there's a difficult, nonlinear tradeoff between cost and quality of service. If you're in charge of the servers (for example, purchasing phone lines and hiring staff for a call center), you want to make sure you don't overprovision, because that's wasteful. On the other hand, you also don't want to underprovision, because that'll frustrate your customers.

The balance between cost and quality of service is usually considered in one of three ways:

1. **Quality Driven**. If quality of service is the priority, then you provision lots of extra capacity in order to get it. You might do this when realtime response is required for special purposes, for example.

2. **Efficiency Driven**. If you care most about efficiency, then you provision as much capacity as you need, and customers have terrible wait times. You might do this if you're planning the call center staffing for an airline or insurance company's customer service line. Just as a random example.

3. **Quality and Efficiency Driven**. If you want to balance quality and efficiency, you aim for this, often abbreviated the QED regime.

It's not difficult to figure out how many servers you need for the quality or efficiency goals. You can do this by solving for a desired utilization or by just holding constant the "padding" of spare capacity above the offered load, respectively. The QED regime is a bit trickier, though.

It turns out that there's a really neat approximate solution that works very well in the real world and is useful to help develop your intuition further. It's typically called the *square root staffing rule*. In a nutshell, it says that **the spare capacity needed to deliver a given quality of service grows proportionately to the square root of the traffic.**

As a thought experiment, imagine that you have 10 web servers behind a load balancer, each of them 80% utilized. You don't know exactly what performance an end user experiences, but you're satisfied with it. You expect traffic to triple for the holiday season. How many servers do you need?

To solve this problem, note that you effectively have 8 servers fully utilized and 2 servers of spare capacity. Your 8 servers need to triple, and your spare capacity needs to grow by a factor of $\sqrt{3}$. So the answer is $3 \cdot 8 + 2\sqrt{3}$, or 27.46. You should round this up to the nearest whole server, or 28 servers.

This rule of thumb is great for developing intuition about how capacity, utilization, and quality of service grow with traffic. It gets more accurate as the offered load grows, and works best above a load of 5 or so.

If you want to solve for a specific desired quality of service, that's also simple. The references at the end of this book contain more information about this.

Applying Queueing Theory in the Real World

Many books on queueing theory have extensive examples and problem sets. This is nice, but in my personal experience, I have found queueing theory, especially Erlang queueing models, very difficult to apply to analysis of real systems.

As an example, I've tried to use Erlang formulas when capacity planning for database servers. I got absolutely nowhere because I found it impossible to measure service times. Not only did I not know S, but I could not compute utilization, nor could I validate that service times were exponentially distributed,[1] so I wasn't sure the Erlang C model was really the right one to use.

None of this is to say that queueing theory isn't practical. It's enormously practical, but sometimes it's either overkill or there are roadblocks to applying it.

To sum up, queueing theory can be hard to apply for the following reasons:

- Lack of visibility into what kind of queuing systems lie inside "black box" systems

- Inability to measure and validate inputs and preconditions for a model, particularly service times

- The difficulty of computing the Erlang formulas

- Dirty or noisy systems, or poor measurements, that don't seem to conform to models

....

[1] The dirty little secret of service times is that they usually seem to be log-normal or gamma distributed, if they even have a clean unimodal distribution at all.

So what is queueing theory good for, then?

In my opinion, queueing theory is best applied *conceptually* in day-to-day life. That is, it's a great framework for having intelligent opinions and discussions about how systems will behave, because as Neil Gunther says, "Common sense is the pitfall of performance analysis." Queueing theory helps you understand whether things are likely to perform worse than intuition suggests. And when you find such a situation, it might help explain why.

I have applied queueing theory in some real-life use cases. For example, VividCortex's Adaptive Fault Detection algorithm is based, in part, on queueing theory. But even in that example, queueing theory was more of a conceptual framework for a successful approach to the problem, than an analytical solution to it.

My approach to queueing theory might not match everyone's, but that's why I wrote this book.

Conclusions

Queueing theory has many practical and important results, as I've illustrated. In my opinion, however, the most important outcomes of studying queueing theory are the following:

- Learning to see queueing systems everywhere. Especially in human systems and the design of teams, processes, and flow of tasks and communications.

- Building better intuition about which variables influence queueing the most. It's hard to think about nonlinear systems, but if you know they're really sensitive to small changes in particular variables or in specific ranges of values, you'll be better equipped.

- Understanding the different queueing configurations and their behaviors.

- Understanding which relationships are linear and straightforward (Little's Law, Utilization Law, etc) and which depend on subtle aspects of system behavior such as arrival processes and their probability distributions.

If you reach the end of this book and understand only the following big-picture points, I'll be more than satisfied:

- A queue is a mechanism to **improve availability, utilization, and throughput, but at the expense of latency and resource consumption.**

- To reduce queueing and waste, try to **reduce utilization and variability, increase the number of servers, and avoid specialization.**

This is not everything there is to know about queueing theory, but in my opinion it's everything you need to know.

Further Reading

This book isn't meant to stand on its own. You should use it as a jumping-off point for doing further research. Here's a selection of the best reference material I've found.

- *Fundamentals of Queueing Theory* by Gross & Harris.

- *Practical Queueing Analysis* by Mike Tanner.

- *Analyzing Computer System Performance With Perl::PDQ* by Neil Gunther, which despite its name is much more about queueing theory than about Perl.

- *Probability, Statistics, and Queueing Theory* by Allen.

- *Capacity Planning for Web Performance* by Menascé and Almeida.

- For more on the QED regime and the square-root staffing rule, see Heavy-traffic limits for queues with many exponential servers by Halfin & Whitt (1981), or a more recent paper, What You Should Know About Queueing Models To Set Staffing Requirements in Service Systems, by Ward Whitt.

- For more on approximations to the Erlang formulas, see this GitHub repository.

About VividCortex

VividCortex is a SaaS database performance monitoring product. The database is the heart of most applications, but it's also the part that's hardest to scale, manage, and optimize even as it's growing 50% year over year. VividCortex has developed a suite of unique technologies that significantly eases this pain for the entire IT department. Unlike traditional monitoring, we measure and analyze the system's work and resource consumption. This leads directly to better performance for IT as a whole, at reduced cost and effort.

Related Resources From VividCortex
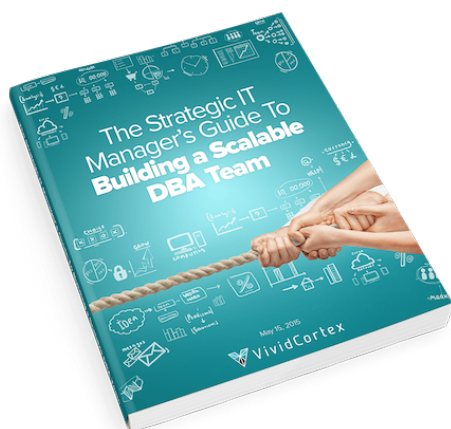
The Strategic IT Manager's Guide To Building A Scalable DBA Team

Case Study: SendGrid

VividCortex has been instrumental in finding issues. It's the go-to solution for seeing what's happening in production systems.