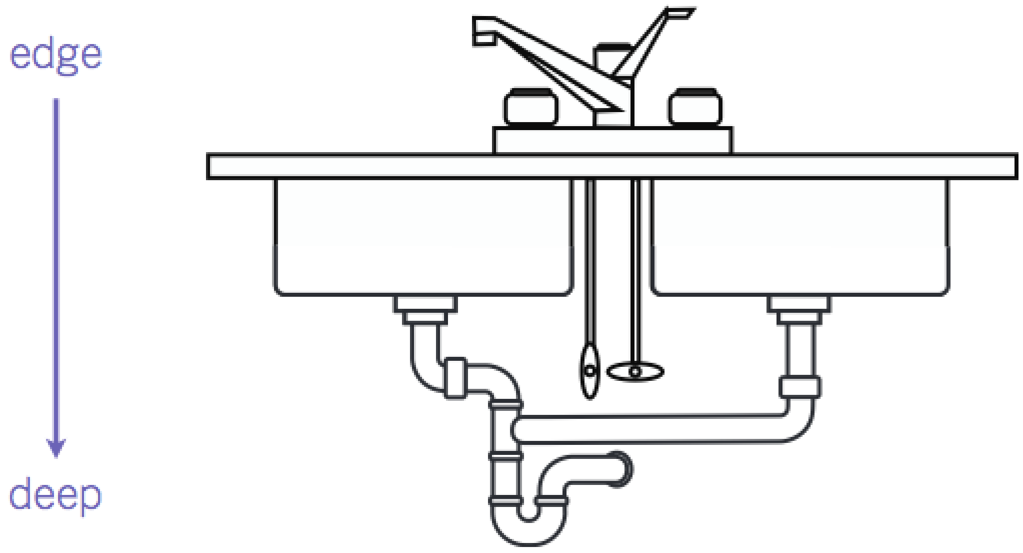
2014/11/19

Queues Don't Fix Overload

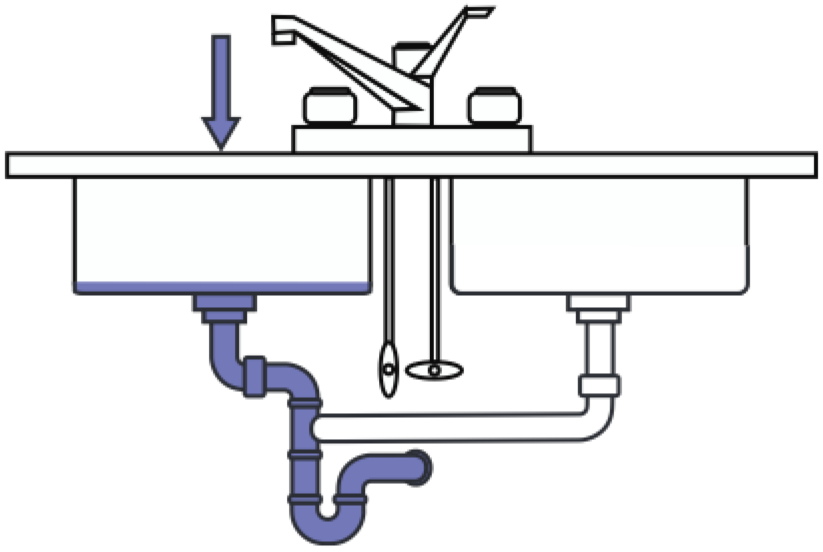
OK, queues.

People misuse queues all the time. The most egregious case being to fix issues with slow apps, and consequently, with overload. But to say why, I'll need to take bits of talks and texts I have around the place, and content that I have written in more details about in Erlang in Anger.

To oversimplify things, most of the projects I end up working on can be visualized as a very large bathroom sink. User and data input are flowing from the faucet, down 'till the output of the system:

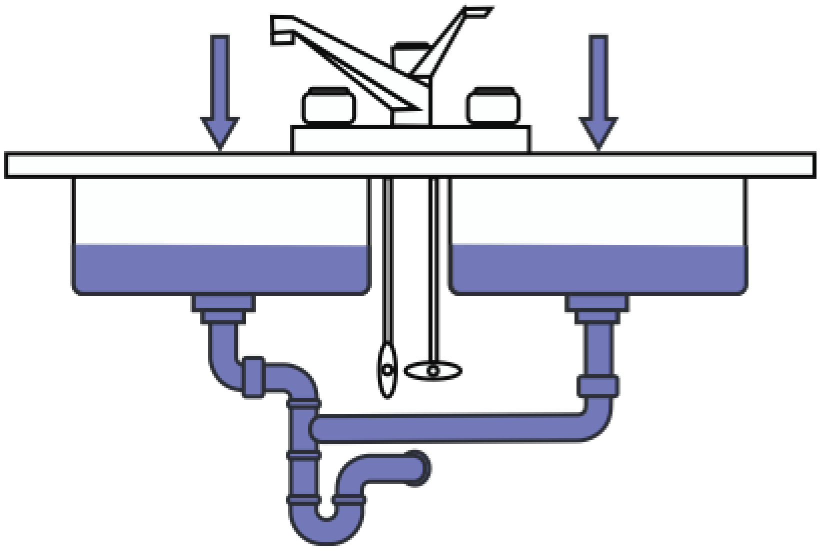


So under normal operations, your system can handle all the data that comes in, and carry it out fine:



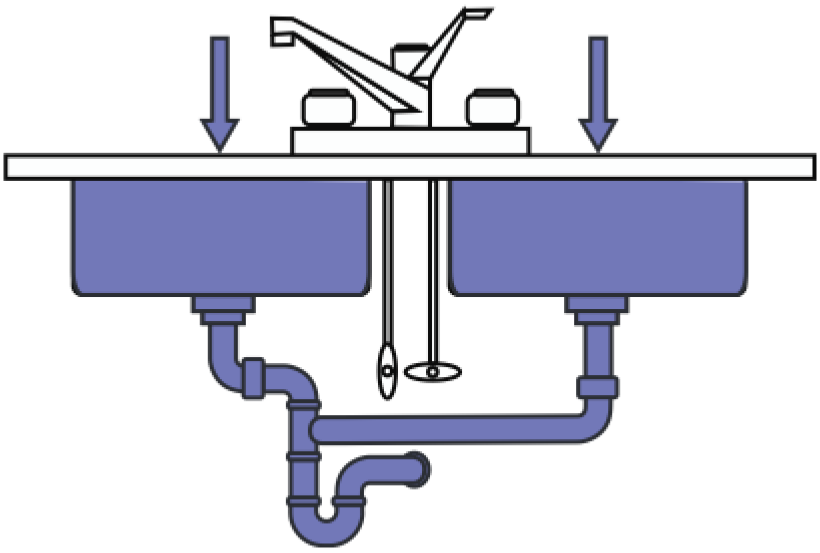
Water goes in, water goes out, everyone's happy. However, from time to time, you'll see temporary overload on your system. If you do messaging, this is going to be around sporting events or events like New Year's Eve. If you're a news site, it's gonna be when a big thing happens (Elections in the US, Royal baby in the UK, someone says they dislike French as a language in Quebec).

During that time, you may experience that temporary overload:



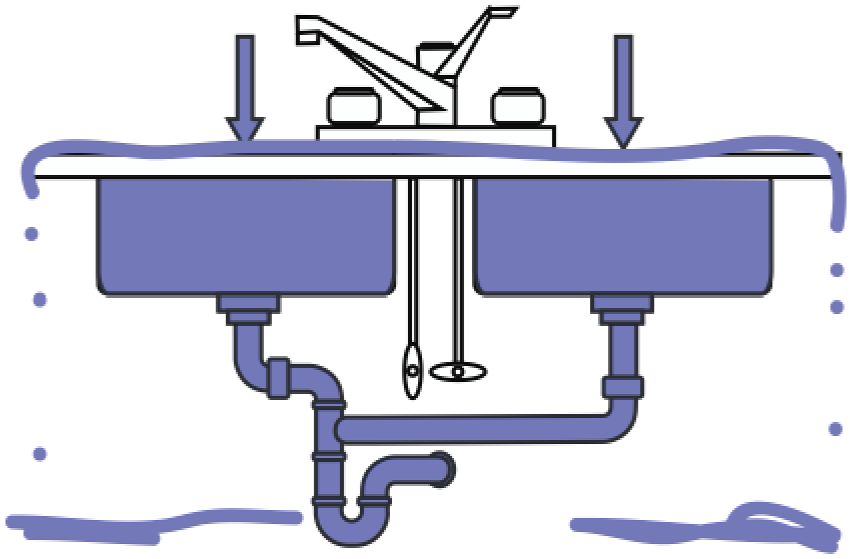
The data that comes out of the system is still limited, and input comes in faster and faster. Web people will use stuff like caches at that point to make it so the input and output required gets to be reduced. Other systems will use a huge buffer (a queue, or in this case, a sink) to hold the temporary data.

The problem comes when you inevitably encounter prolonged overload. It's when you look at your system load and go "oh crap", and it's not coming down ever. Turns out Obama doesn't want to turn in his birth certificate, the royal baby doesn't look like the father, and someone says Quebec should be better off with Parisian French, and the rumor mill is going for days and weeks at a time:



All of a sudden, the buffers, queues, whatever, can't deal with it anymore. You're in a critical state where you can see smoke rising from your servers, or if in the cloud, things are as bad as usual, but more!

The system inevitably crashes:

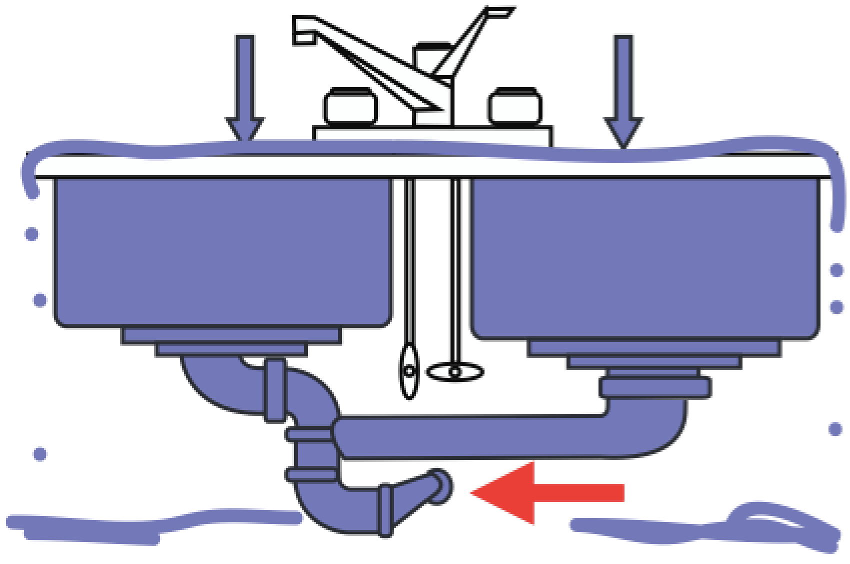


Woops, everyone is dead, you're in the office at 3am (who knew so many people in the US, disgusted with their "Kenyan" president, now want news on the royal baby, while Quebec people look up 'royale with cheese baby' for some reason) trying to keep things up.

You look at your stack traces, at your queue, at your DB slow queries, at the APIs you call. You spend weeks at a time optimizing every component, making sure it's always going to be good and solid. Things keeps crashing, but you hit the point where every time, it takes 2-3 days more.

At the end of it, you see a crapload of problems still happening, but they're a week apart between each failure, which slows down your optimizing in immense ways because it's incredibly hard to measure things when they take weeks to go bad.

You go "okay I'm all out of ideas, let's buy a bigger server." The system in the end looks like this, and it's still failing:



Except now it's an unmaintainable piece of garbage full of dirty hacks to make it work that cost 5 times what it used to, and you've been paid for months optimizing it for no god damn reason because it still dies when overloaded.

The problem? That red arrow there. You're hitting some hard limit that even through all of your profiling, you didn't consider properly. This can be a database, an API to an external service, disk speed, bandwidth or general I/O limits, paging speed, CPU limits, whatever.

You've spent months optimizing your super service only to find out at some point in time, you went past its optimal speed without larger changes, and the day your system got to have an operational speed greater than this hard limit, you've doomed yourself to an everlasting series of system failures.

The disheartening part about it is that you discover that once your system is popular, has people using it and its APIs, and changing it to be better is very expensive and hard. Especially since you'll probably have to revisit assumptions you've made in its core design. Woops.

So what do you need? You'll need to pick *what has to give* whenever stuff goes bad. You'll have to pick between blocking on input (back-pressure), or dropping data on the floor (load-shedding). And that happens all the time in the real world, we just don't want to do it as developers, as if it were an admission of failure.

Bouncers in front of a club, water spillways to go around dams, the pressure mechanism that keeps you from putting more gas in a full tank, and so on. They're all there to impose a system-wide flow control to keep operations safe.

In [non-critical] software? Who cares! We never shed load because that makes stakeholders angry, and we never think about back-pressure. Usually the back-pressure in the system is implicit: 'tis slow.

A function/method call to something ends up taking longer? It's slow. Not enough people think of it as back-pressure making its way through your system. In fact, slow distributed systems are often the canary in the overload coal mine. The problem is that everyone just stands around and goes "durr why is everything so slow??" and devs go "I don't know! It just is! It's hard, okay!"

That's usually because somewhere in the system (possibly the network, or something that is nearly impossible to observe without proper tooling, such as TCP incast), something is clogged and everything else is pushing it back to the edge of your system, to the user.

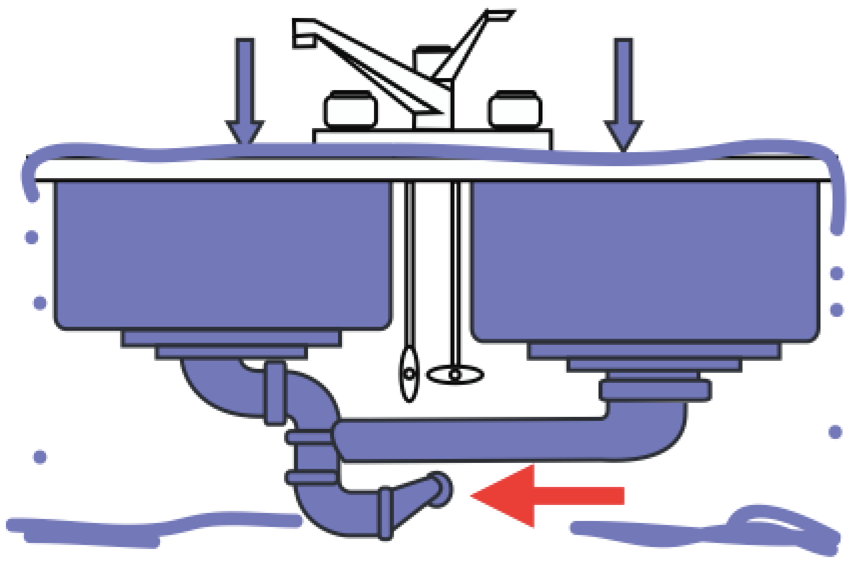
And that back-pressure making the system slower? It slows down the rate at which users can input data. It's what is likely keeping your whole stack alive. And you know when people start using queues? Right there. When operations take too long and block stuff up, people introduce a freaking queue in the system.

And the effects are instant. The application that was sluggish is now fast again. Of course you need to redesign the whole interface and interactions and reporting mechanisms to become asynchronous, but man is it fast!

Except at some point the queue spills over, and you lose all of the data. There's a serious meeting that then takes place where everyone discusses how this could possibly have happened. Dev #3 suggests more workers are added, Dev #6 recommends the queue gets persistency so that when it crashes, no requests are lost.

"Cool," says everyone. Off to work. Except at some point, the system dies again. And the queue comes back up, but it's already full and uuugh. Dev #5 goes in and thinks "oh yeah, we could add more queues" (I swear I've seen this unfold back when I didn't know better). People say "oh yeah, that increases capacity" and off they go.

And then it dies again. And nobody ever thought of that sneaky red arrow there:



Maybe they do it without knowing, and decide to go with MongoDB because it's "faster than Postgres" (heh). Who knows.

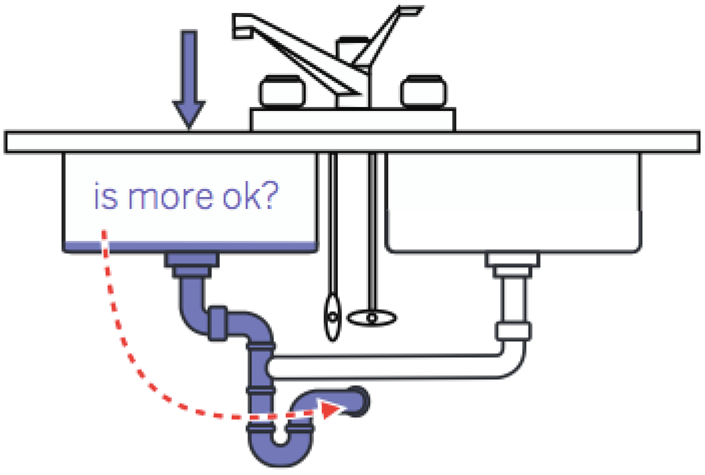
The real problem is that everyone involved used queues as an optimization mechanism. With them, new problems are now part of the system, which is a nightmare to maintain. Usually, these problems will come in the form of ruining the end-to-end principle by using a persistent queue as a fire-and-forget mechanisms or assuming tasks can't be replayed or lost. You have more places that can time out, require new ways to detect failures and communicate them back to users, and so on.

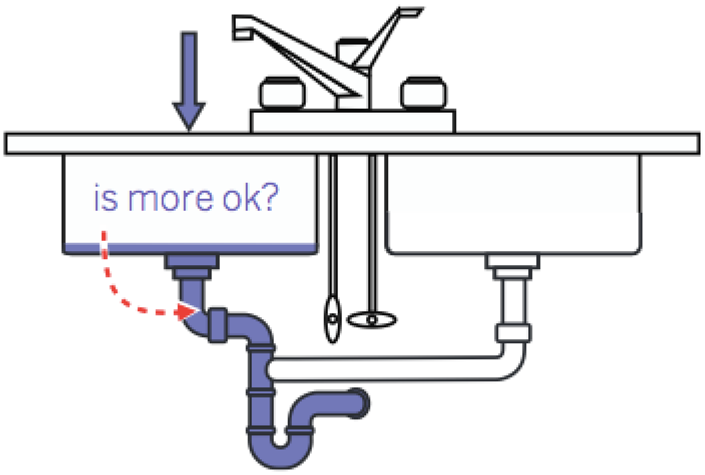
Those can be worked around, don't get me wrong. The issue is that they're being introduced as part of a solution that's not appropriate for the problem it's built to solve. All of this was just premature optimization. Even when everyone involved took measures, reacted to real failures in real pain points, etc. The issue is that nobody considered what the true, central business end of things is, and what its limits are. People considered these limits locally in each sub-component, more or less, and not always.

But someone should have picked what had to give: do you stop people from inputting stuff in the system, or do you shed load. Those are inescapable choices, where inaction leads to system failure.

And you know what's cool? If you identify these bottlenecks you have for real in your system, and you put them behind proper back-pressure mechanisms, your system won't even have the right to become slow.

Step 1. Identify the bottleneck. Step 2: ask the bottleneck for permission to pile more data in:





Depending on where you put your probe, you can optimize for different levels of latency and throughput, but what you're going to do is define proper operational limits of your system.

When people blindly apply a queue as a buffer, all they're doing is creating a bigger buffer to accumulate data that is in-flight, only to lose it sooner or later. You're making failures more rare, but you're making their magnitude worse.

When you shed load and define proper operational limits to your system, you don't have these. What you may have is customers that are as unhappy (because in either case, they can't do what your system promises right), but with proper back-pressure or load-shedding, you gain:

- Proper metrics of your quality of service
- An API that will be designed with either in mind (back-pressure lets you know when you're in an overload situation, and when to retry or whatever, and load-shedding lets the user know that some data was lost so they can work around that)
- Fewer night pages
- Fewer critical rushes to get everything fixed because it's dying all the time
- A way to monetize your services through varying account limits and priority lanes

- You act as a more reliable endpoint for everyone who depends on you

To make stuff usable, a proper idempotent API with end-to-end principles in mind will make it so these instances of back-pressure and load shedding should rarely be a problem for your callers, because they can safely retry requests and know if they worked.

So when I rant about/against queues, it's because queues will often be (but not always) applied in ways that totally mess up end-to-end principles for no good reason. It's because of bad system engineering, where people are trying to make an 18-wheeler go through a straw and wondering why the hell things go bad. In the end the queue just makes things worse. And when it goes bad, it goes really bad, because everyone tried to close their eyes shut and ignore the fact they built a dam to solve flooding problems upstream of the dam.

And then of course, there's the use case where you use the queue as a messaging mechanism between front-end threads/processes (think PHP, Ruby, CGI apps in general, and so on) because your language doesn't support inter-process communications. It's marginally better than using a MySQL table (which I've seen done a few times and even took part in), but infinitely worse than picking a tool that supports the messaging mechanisms you need to implement your solution right.