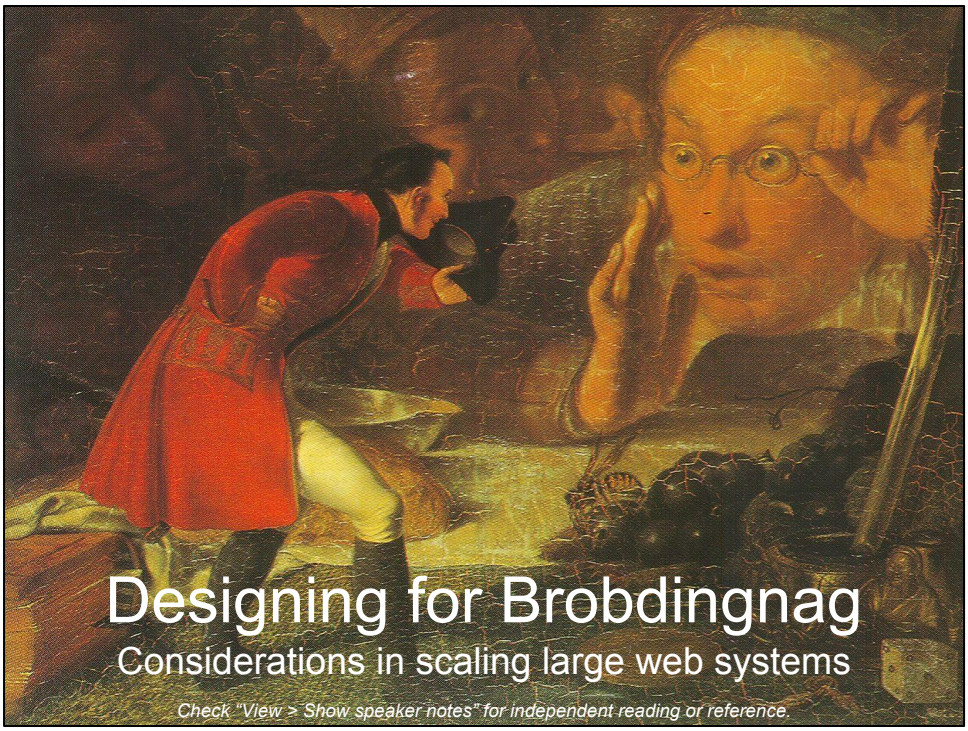


This presentation is designed for reading and reference outside the talk proper; please enlarge the speaker notes window.

Hello, and welcome to the first tech-talk of today's SRE Classroom in Dublin.

There's a lot in this talk, but I'll pause for questions after each major section, and we'll have time for more discussion at the end.

In particular, please tell me if our blind spots about scale manifest themselves: working in SRE tends to create slightly divergent perceptions of "small" and "large" in systems. :o)

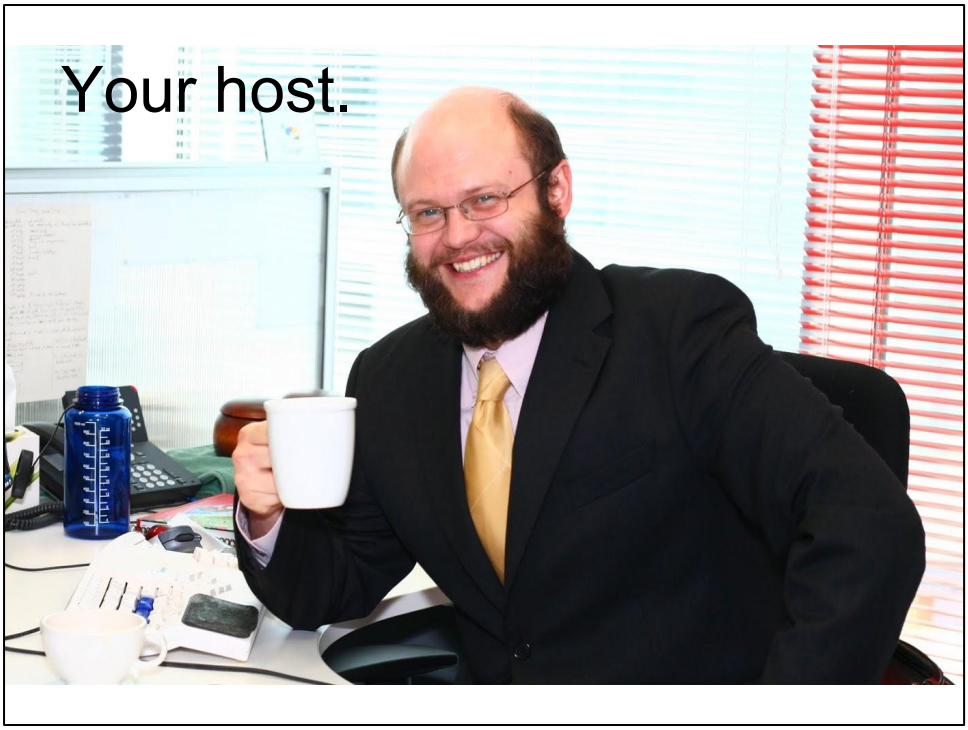
Brobdingnag, by the way, is a land of giants visited by Lemuel Gulliver in his famous *Travels*. The author Jonathan Swift was dean at St. Patrick's Cathedral, which isn't far away.

This is a 50-minute talk intended for software and systems engineering SRE candidates who may lack exposure to large-scale systems.

It is based on presentations at the London and Mountain View SRE Classroom events by Matt Brown (mattbrown@google.com), John Neil (jneil@google.com) and Robert Spier (rspier@google.com). I also include ideas from Jeff Dean's (jeff@google.com) talk Building Software Systems At Google and Lessons Learned.

Many thanks to Niall Richard Murphy (niallm@google.com), Alex Perry (alexperry@google.com), Laura Nolan (lnolan@google.com) and Pete Nuttall (psn@google.com) for assistance, ideas and review.

Image from “Gulliver in Brobdingnag” by Richard Redgrave. http://en.wikipedia.org/wiki/File:Szene_aus_Gulliver%27s_Reisen_-_Gulliver_in_Brobdingnag.jpg. Public domain.



My name's Cian Synnott, cian@google.com, and I joined Google in 2005.

Before Google, I worked for some Irish Internet service providers. At Google, I've been a corporate sysadmin; a "product" SRE in Ads; and an "infrastructure" SRE in Storage.

Image from Ads SRE Dublin's "formal friday", July 2009. By Alexey Klimov, neuro@google.com. Used with permission.

Each of these jobs has been in a different environment, none of them as serene as Dublin's Grand Canal in snow.

The environments we work in comprise many different factors - for example: types of software; numbers of instances; platforms available; vendor vs. locally built software; customers; teams and company culture. They all contribute.

From the point of view of a systems designer, each environment presents different opportunities and constraints.

Today, we're focusing on technical designs in large-scale software environments, and patterns which we've found work well.

These can be relatively difficult to get exposure to.

This won't be an exhaustive treatment, but we hope to give you:

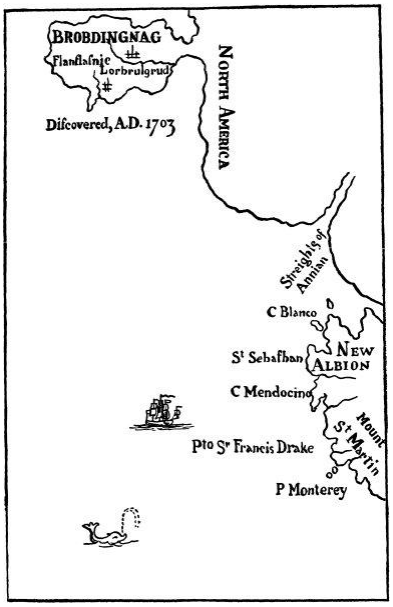
- an appreciation of the kinds of constraints and opportunities we have at scale;
- some tools for approaching design;
- and a better idea of the kinds of systems SREs engage with.

Note that we could talk about subtleties, trade-offs and mostly-trues in practically every slide, so I've chosen to restrain my hedging and be fairly definite in the interests of clarity.

In any case, thinking along these lines and about these issues should help you avoid many pitfalls.

Image “Grand Canal in Snow”, by and copyright Niall Richard Murphy, http://500px.com/niallm/sets/dublin. Used with permission.

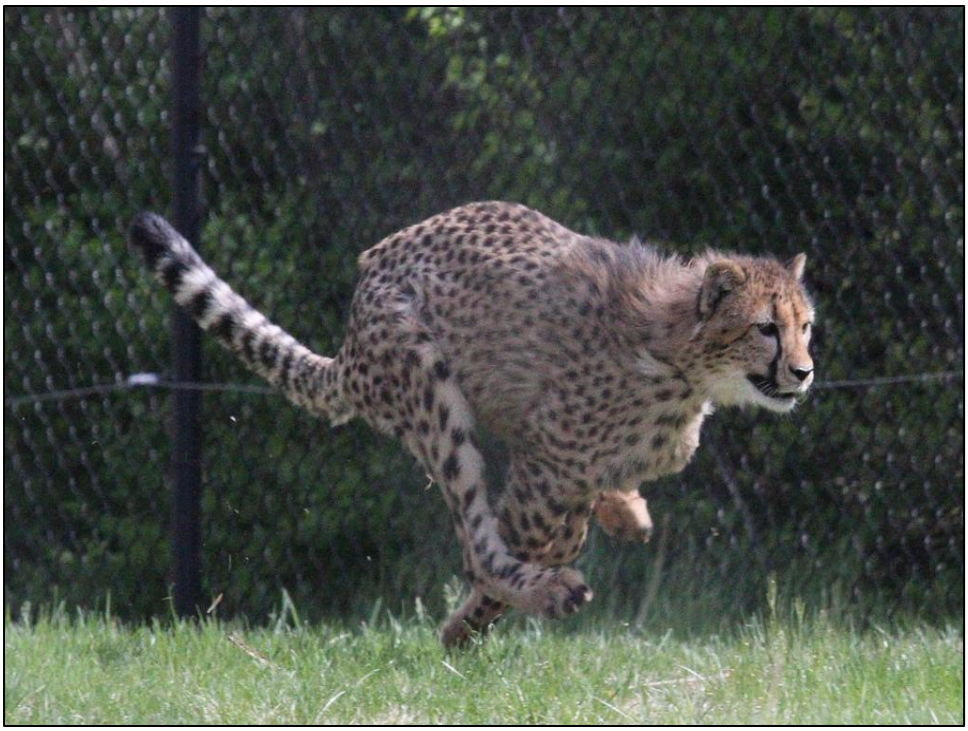
You are here. →



So what do we define as a large-scale environment in SRE? What features can we expect to see?

1. Big numbers. We're dealing with large volumes of data - at rest or in flight, often both.

Image from "Map of Brobdingnag" by Hermann Moll http://en.wikipedia.org/wiki/File:Brobdingnag_map.jpg. Public domain.



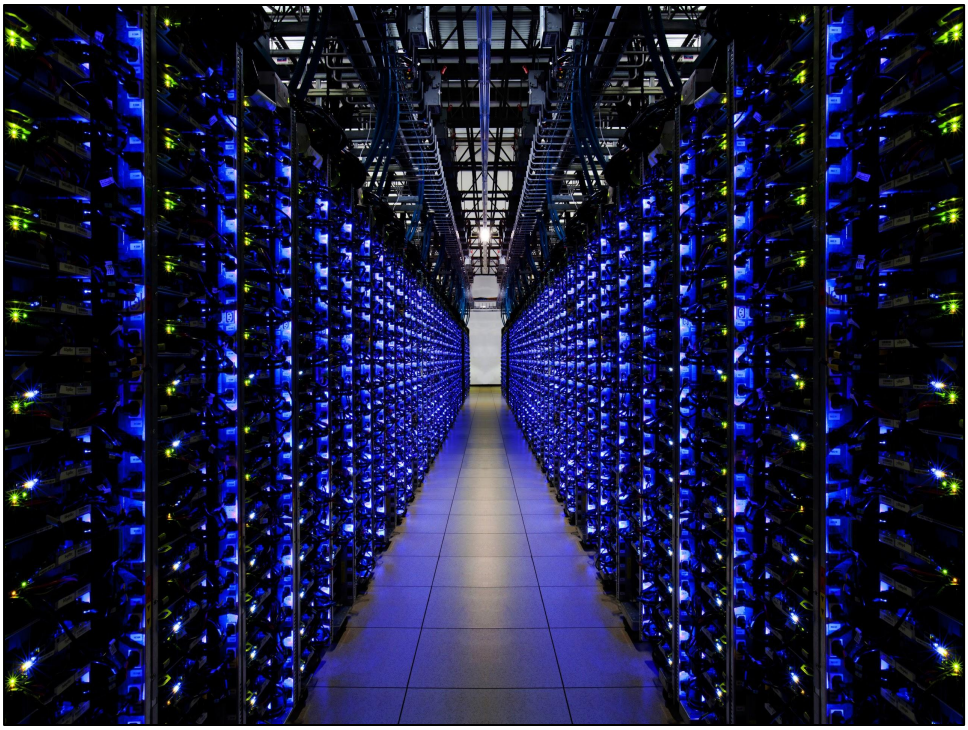
2. Speed matters.

Web user experience research repeatedly shows that users really care about speed.

Keeping latency low is the main driver of many design elements in large web systems.

But we need to keep costs low, too: most large web services have small profit margins, so we can't throw a pile of money at reducing latency; it tends to be expensive and difficult to "retrofit".

These together drive, for example, the high degree of "fan out" in backend designs and a related "horizontal scaling" model; we'll cover these later.



3. Hundreds or thousands of connected machines.

Systems that can manage large volumes of data or concurrent requests are necessarily distributed.

SRE-supported systems at Google typically tend towards the thousands (or more).

We expect to deal with multiple datacenters: this offers us a host of physical and networking constraints, but also an amazing platform to build on.

It’s worth noting that unlike most other large compute deployments, we don't build ours to be single purpose.

Image from Google’s facility at Douglas County, Georgia. http://www.google.com/about/datacenters/gallery/#/locations/douglas-county.

4. Failure is normal.

If you take nothing else away from today, this is it.

At all levels of large-scale distributed systems, things fail.

At the physical level, power supplies explode; RAM goes bad; CPUs exhibit strange behaviour; disks crash; hunters shoot down optical fiber.

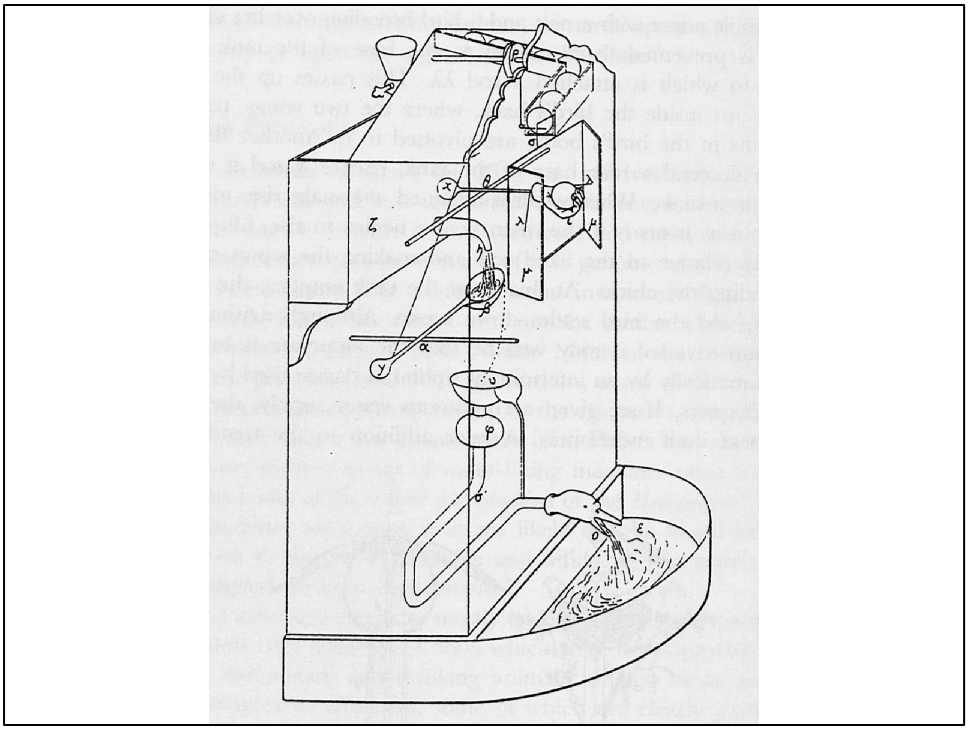
Remember that all those computers depend on building, power, HVAC and networking infrastructure.

But that's just the start, really. The software systems we build, even those for handling failure, fail too!

Niall will be speaking about this in some detail later.

There are topics I'll cover that harmonize or overlap slightly with his, but in general I'll avoid detailed discussion of failure modes.

Image from the train wreck at Montparnasse Station, at Place de Rennes side (now Place du 18 Juin 1940), Paris, France, 1895. http://en.wikipedia.org/wiki/File:Train_wreck_at_Montparnasse_1895.jpg. Public domain.



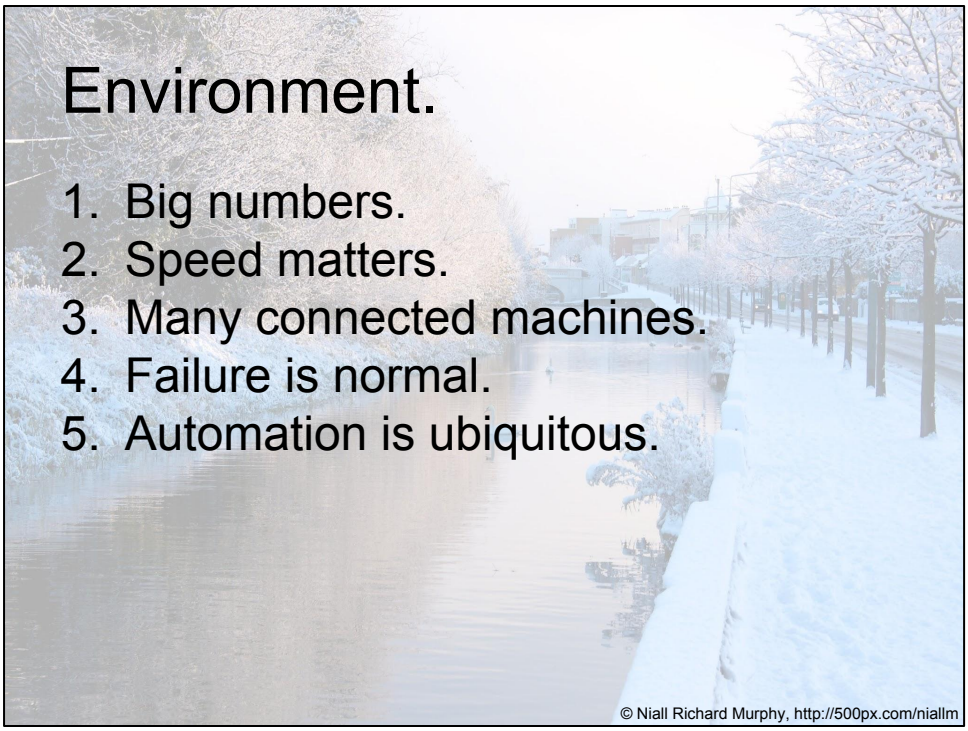
5. Automation is ubiquitous.

Large scale software systems tend to (need to!) be highly automated. Humans typing at terminals will only scale so far. How far? Not sure exactly, but we're way past that.

This can get a bit scary, in terms of layering and whole-system understanding: we can end up writing programs to configure programs to monitor programs that run Google.

It can also be a lot of fun.

Image from Reconstruction of a washstand with escapement mechanism, the earliest known, as described by the Greek engineer Philo of Byzantium (3rd century BC) by Carra de Vaux, B. http://en.wikipedia.org/wiki/File:Washstand_by_Philo_of_Byzantium.png. Public domain.



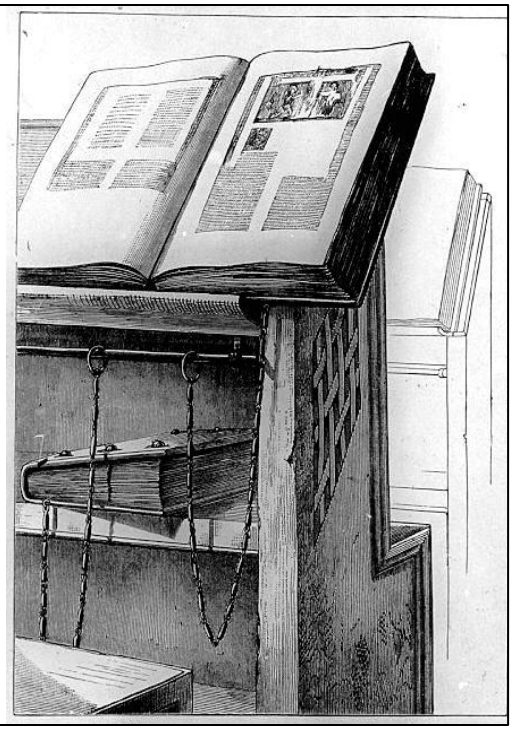
Recap: features of the large-scale environment.

I want to especially call out number 5. Much of what we will discuss about replication, “sharding” etc. cannot reasonably be achieved without high levels of automation.

Questions?

Image “Grand Canal in Snow”, by and copyright Niall Richard Murphy, http://500px.com/niallm/sets/dublin. Used with permission.

Terminology.



I'll be throwing various terms around during the talk:

A **server** is an individual process or binary running on a machine.

A **machine** is a physical or virtual piece of hardware running an operating system.

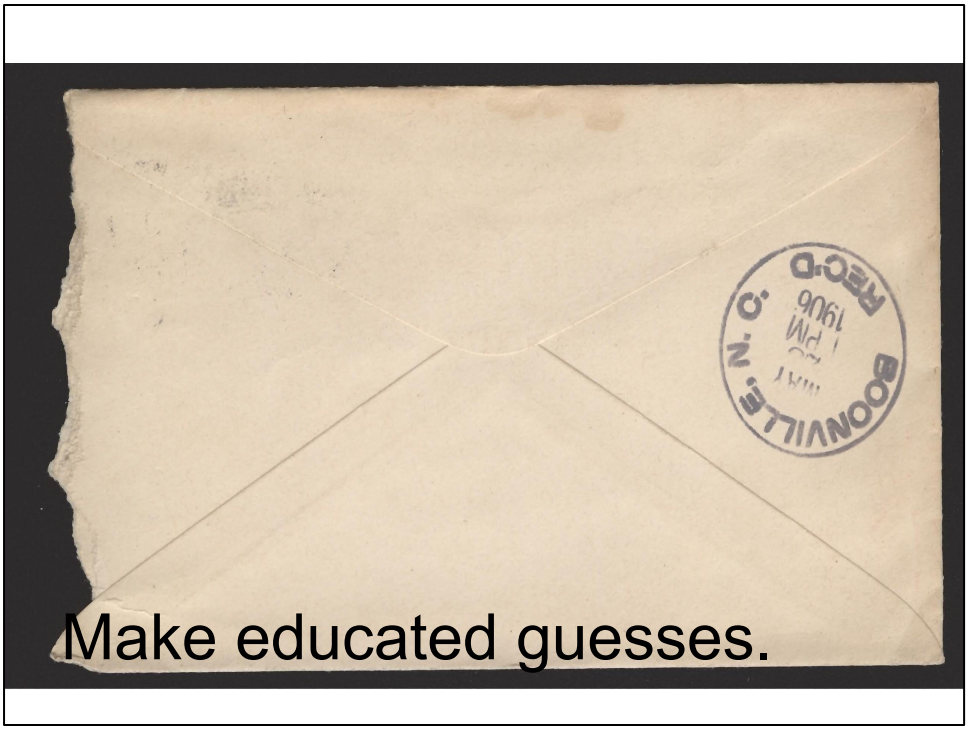
We use the term **queries per second** or **qps** to talk about request volume; "rps" might be more generic, but we're a search company. :o)

Image from Press with chained book in the Library of Cesena, Italy; original drawing has caption "part of a bookcase at Cesena to shew the system of chaining"; by John Willis Clark. http://en.wikipedia.org/wiki/File:Milkau_B%C3%BCcherschrank_mit_angekettetem_Buch_aus_der_Bibliothek_von_Cesena_109-2.jpg. Public domain.

First, a few important ideas:

1. estimation;
2. simplicity;
3. transparency.

Image from Cornerstone, St. Vincent De Paul Church, Bywater neighborhood of New Orleans; Photo by Infrogmation. http://en.wikipedia.org/wiki/File:BywaterStVincentCornerstone2.jpg. Creative Commons Attribution 2.5 Generic.



You've seen the "Latency numbers every programmer should know" handout, with some worked examples.

In choosing between two designs, or four, or in constraining the solution domain enough that we can start to make progress, being able to estimate what *might* work is invaluable.

This kind of "back of the envelope" calculation requires that we know the characteristics of the building blocks we're working with - in a distributed system, often these are in terms of latency, availability or durability.

Note that as the components we're working with become more complex - for example, an underlying distributed filesystem or lock service - this can be difficult and may require more research.

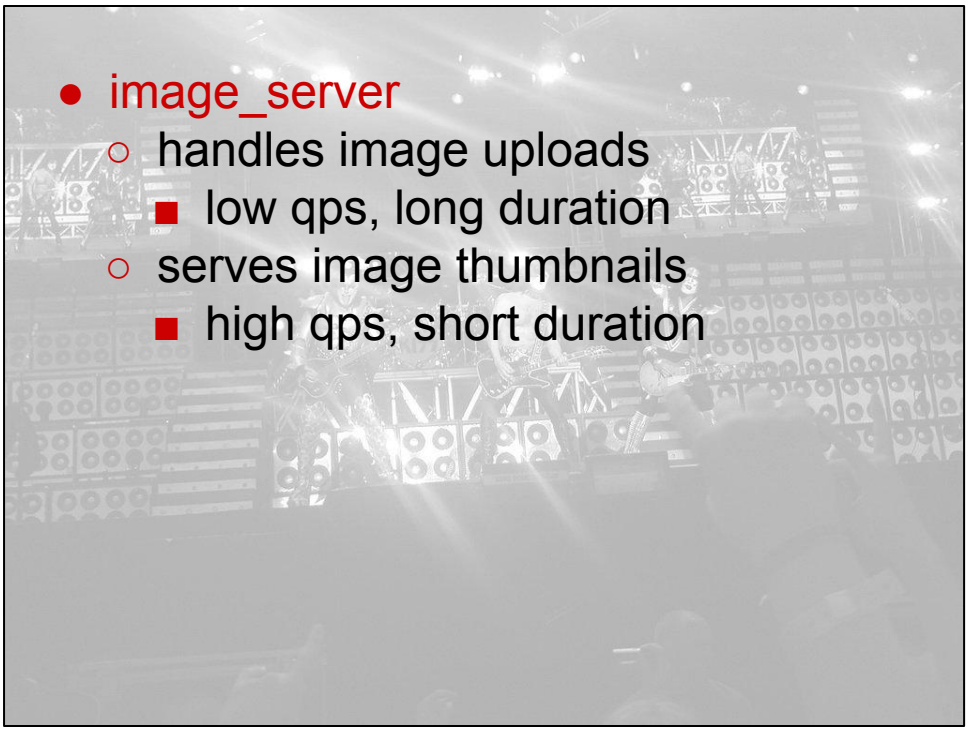
Image from Back of an envelope mailed in the U.S. in 1906, showing a receiving office postmark. http://en.wikipedia.org/wiki/File:Envelope_-_Boonville_Address-002.jpg. Public domain.

The KISS principle is not exactly a new idea in design. What does it mean for us?

Servers should do one thing and do it well.

In particular, we've found it's wise to avoid mixing request types in one server.

Image from Kiss in Stockholm in 2008 by Wikipedia user FrehleyKISS. http://en.wikipedia.org/wiki/File:KISS_Stockholm_2008_4.JPG. Public domain.



Say we're designing a system in which we need to upload images and serve thumbnails.

Our first design iteration might have these two functions as different handlers in one server.

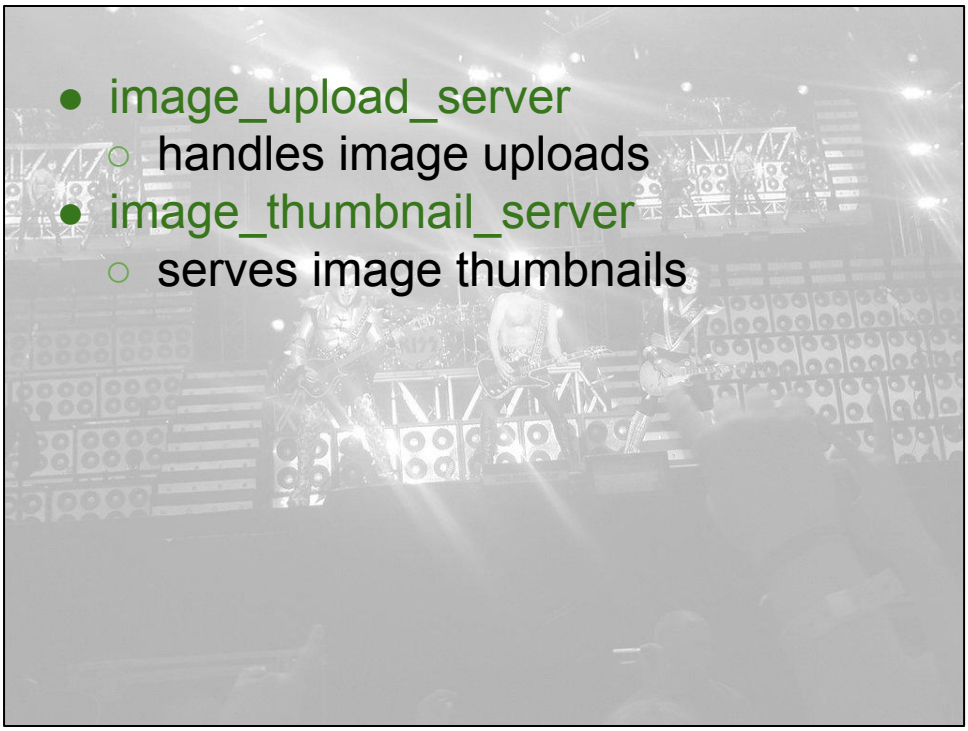
But they have different traffic characteristics, and the mix of requests coming in can change.

This makes it hard to reason about the server's capacity and operation.

With a monolithic, tightly-bound server like this -- and this is obviously a trivial example, we've seen much worse, many more functions all wrapped up together -- we're constrained to scaling in full instances, even if it's only one request type that's hitting a bottleneck.

That is, we might hit capacity in some dimension on one request path, e.g. disk I/O on upload, long before we hit it on the other. Or as another example, in a garbage collected language like Java, the changing memory profiles between requests can make it hard for the VM to anticipate and keep up with what's going on -- making collection take more time, reducing our capacity, and slowing requests.

Image from Kiss in Stockholm in 2008 by Wikipedia user FrehleyKISS. http://en.wikipedia.org/wiki/File:KISS_Stockholm_2008_4.JPG. Public domain.



Instead, split traffic out by request type.

We now have consistent behaviour per server, which is easy to reason about.

In particular, it's easy to monitor these for capacity and scale them independently as they hit resource bottlenecks.

Another benefit is that we get some isolation between parts of our site now; if for some reason a bug in the upload path starts crashing our upload servers, we keep serving thumbnails from the other.

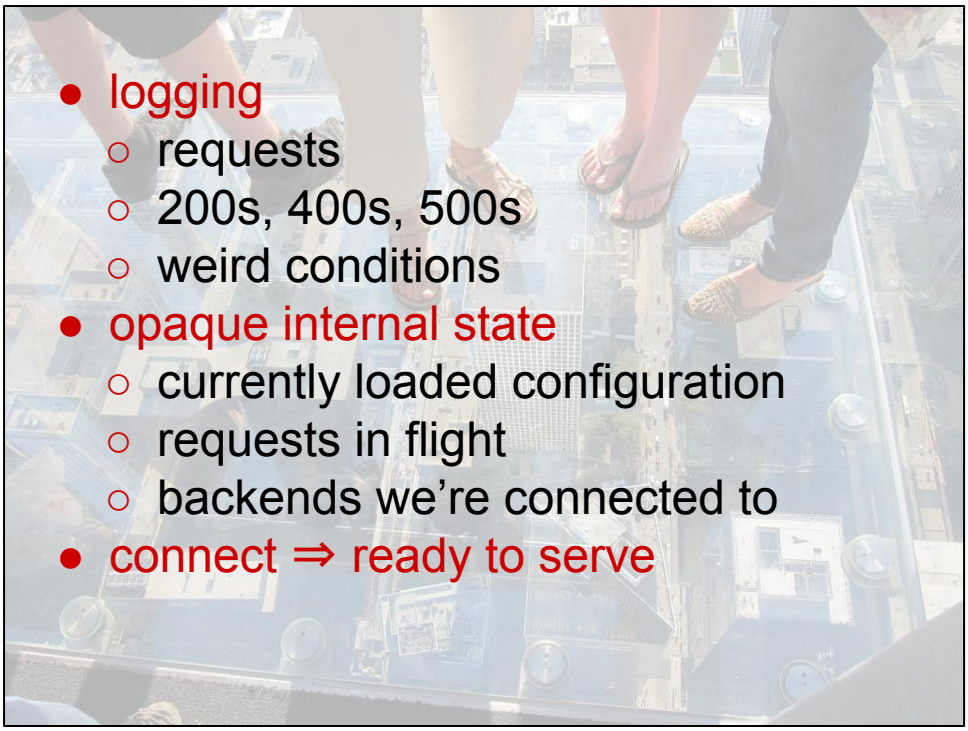
Deploying break-fix code independently to the upload server will be easier too.

The downside is that the two may have to communicate somehow, so there's extra work in maintaining and testing the interface between our two jobs.

Image from Kiss in Stockholm in 2008 by Wikipedia user FrehleyKISS. http://en.wikipedia.org/wiki/File:KISS_Stockholm_2008_4.JPG. Public domain.

No part of our system should be a black box - that is, where requests go in, responses come out, and we don’t know what happens inside.

We handle the vendor issue by reinventing the wheel, which leads to all the concomitant failures and successes you might expect.



First, at the level of individual servers, the traditional way to record interesting events at runtime is logging - INFO, WARN, ERROR, etc.

Lots of things qualify as interesting events. Unfortunately, logs tend to be extremely noisy, and provide metrics after collection and analysis rather than as they happen.

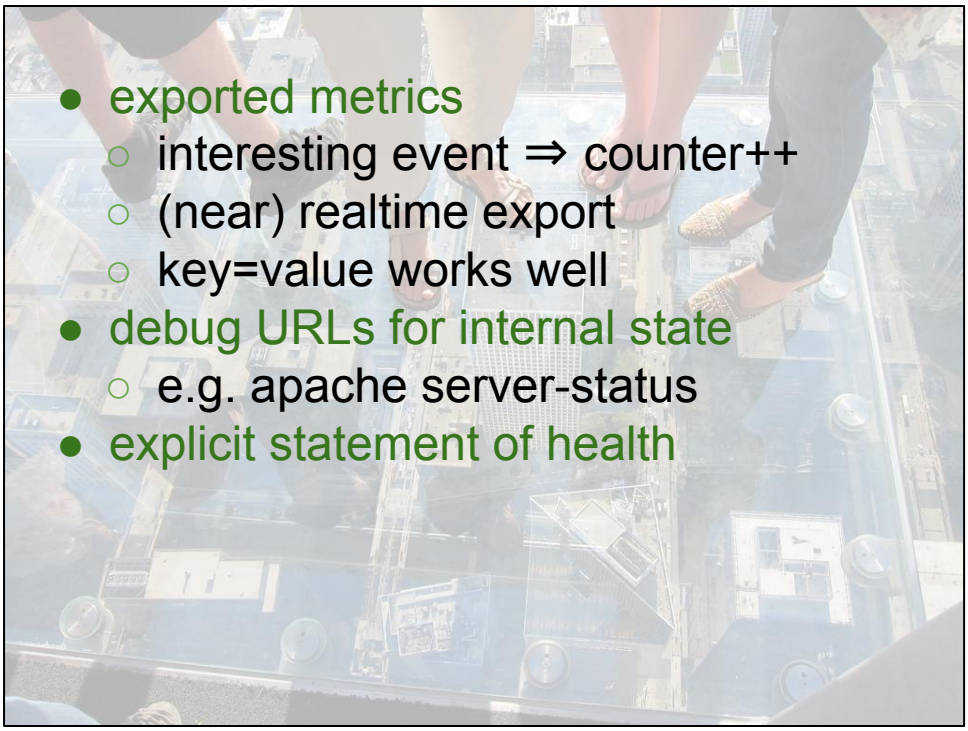
Making things worse, most software does not allow inspection of its internal state.

What we might desperately want to know when debugging a problem, we need to infer by reading code, calling the authors, or sorting out the signal from the noise in a 2GiB log file.

Logs are a great way to ship things that require a degree of structure, durability and analysis - for example ad impressions - but they're a pretty poor aid to live debugging.

Finally, load-balancers and other automation often assume that being able to connect to a server means it's ready to serve requests.

Unless we're being very careful with initialization of internal data structures, etc. before listening on a port, that's probably not the case.



Keep track of those interesting events as metrics.

Export these metrics to the monitoring system. Here we typically have a URL that exports key-value pairs.

Allow engineers to inspect internal state directly. Again, we can do this using debug URLs: it's incredibly useful and saves a lot of time when debugging difficult problems.

Finally, have our server explicitly signal whether it believes it is healthy and able to process requests. So if we're initializing, or get into a bad state, we can signal that. This can be done within an RPC protocol or simply with a health-check URL.

All of that is at the level of a server.

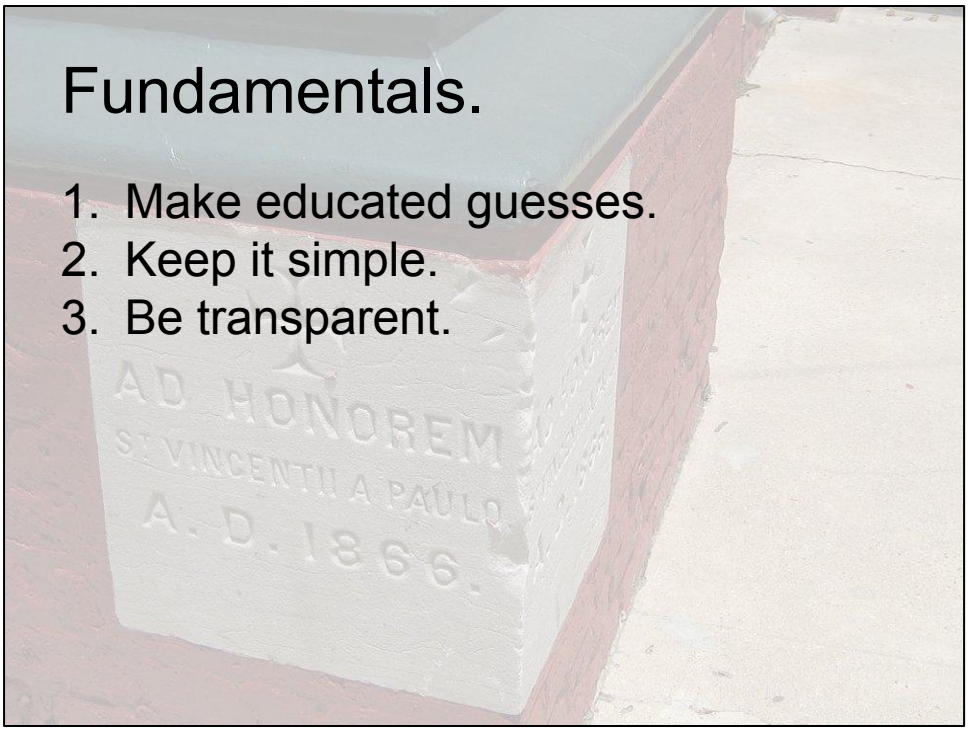
At the level of the whole system, we need to gather the metrics exported by these individual components into useful whole-system views.

For example: we should, at a glance, be able to compare a particular system's request, error, and cache hit rates.

We won't have time to treat monitoring in detail today, but we'll touch on some more related topics towards the end of the talk.

Image from Chicago SkyDeck - glass box bottom by UCFFool. http://www.flickr.

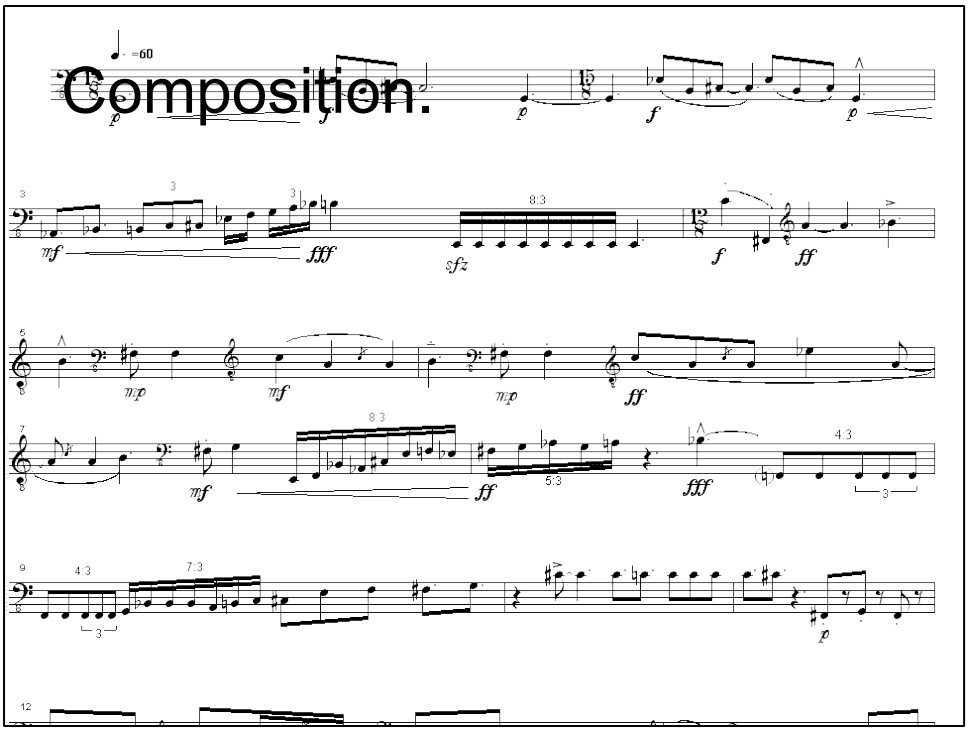
com/photos/ucffool/8060999441/. Creative Commons Attribution 2.0 Generic.



Recap: fundamental ideas.

Practice with back-of-the-envelope stuff. It's fun and enlightening.
Keep designs simple, and unless we have a good reason don't mix requests with different characteristics.
Make software transparent to monitoring and inspection.

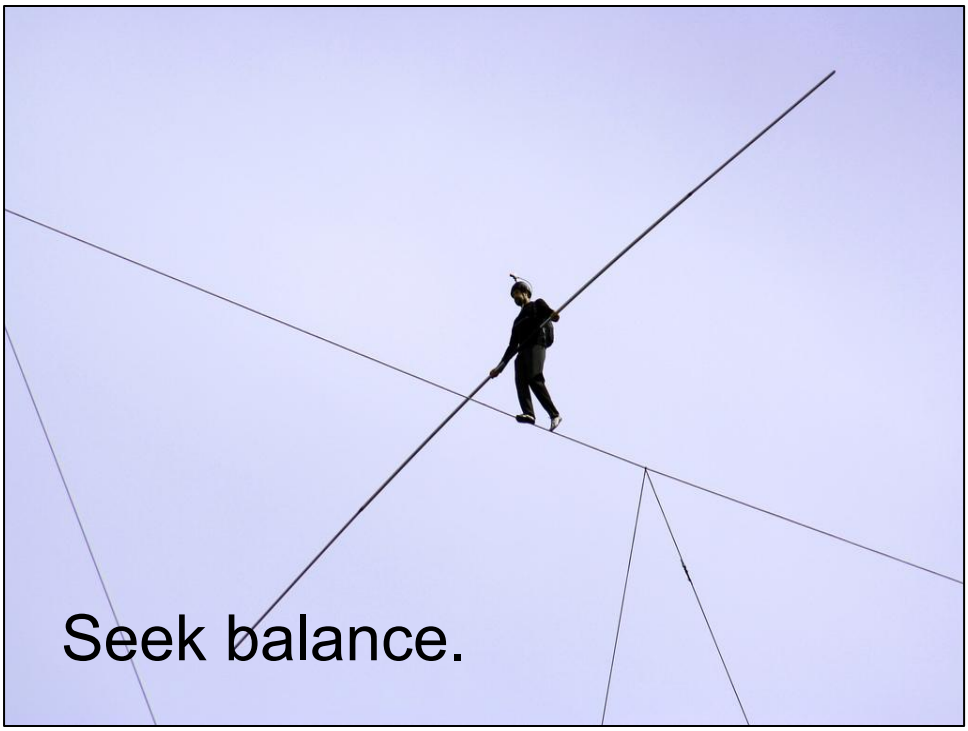
Questions?



Next, we're going to talk about some ways we can assemble large-scale systems:

- Issues in balancing load;
- Some ideas about handling state;
- Problems of consensus and coordination.

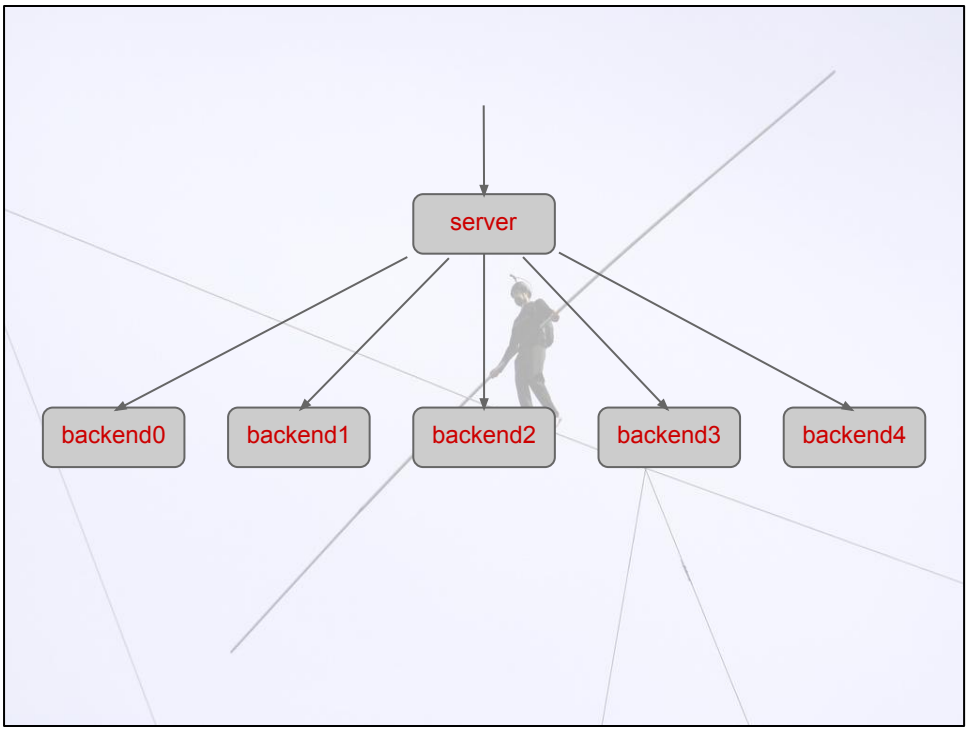
In each case, we'll try and give examples and illustrations, but note that this is a whistle-stop tour: each of these topics can sustain a great deal of thought and argument.



I'll assume some familiarity with the basics of load balancing - for example, how one might use DNS & connection balancing to get requests from users to some set of frontends.

Note that in large web systems, we usually control the client beyond that point - so we have more options in terms of server discovery, request balancing, throttling, etc. on the backend.

We'll focus on structure here; Niall will speak later about some of the fascinating failure-modes of request balancing.

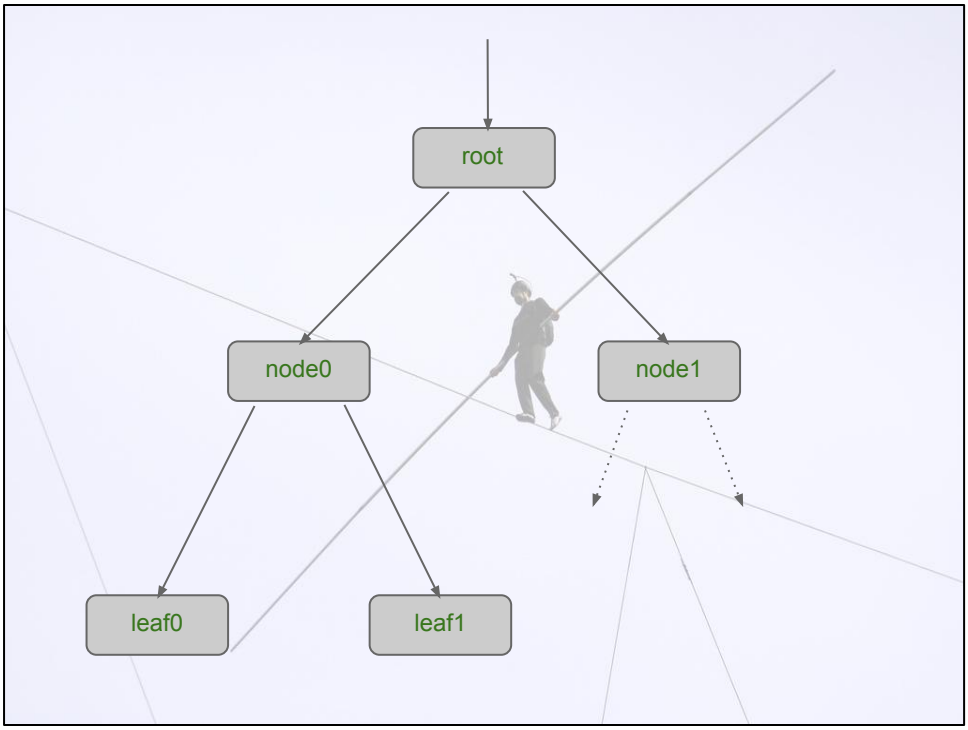


Typically we see a lot of “fan-out” in the backend request structure of large web systems.

This supports both performance - we break up and balance out the total load of processing user requests - and the separation of concerns or request types we noted under “keep it simple” above.

However, this can lead to the problem of backend response “incast”, where the fan-in of (possibly large) responses to thousands of backend RPCs overwhelms the root of the request tree.

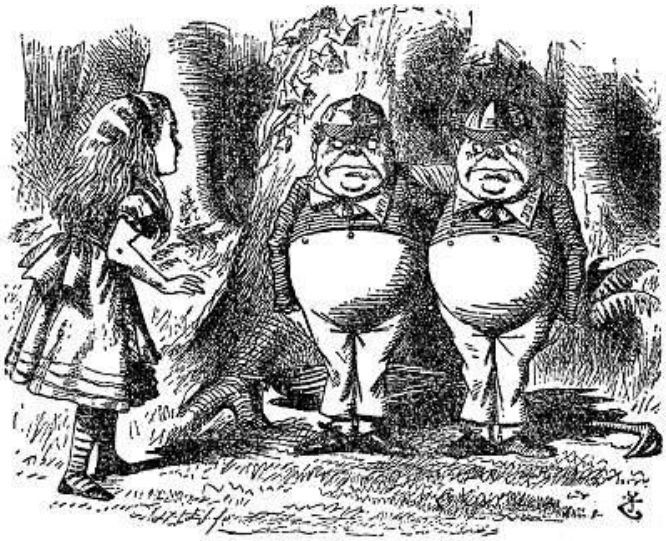
This can cause a bottleneck for example at the network - leading to dropped packets and concomitant poor performance - or at the CPU as the root server attempts to combine these results into its own response.



A tree distribution of requests and responses often works better. It works best when nodes in the tree can process, combine and/or reduce child responses before forwarding them to a parent.

Costs are distributed across more machines; fan-in at any one node is more manageable.

Note that we need to watch for the introduction (intended or not) of cycles into the request tree as the system evolves. See notes on feedback in Niall's talk on failure later.



Replicate everything.

We design servers to run as many replicas, rather than single instances.

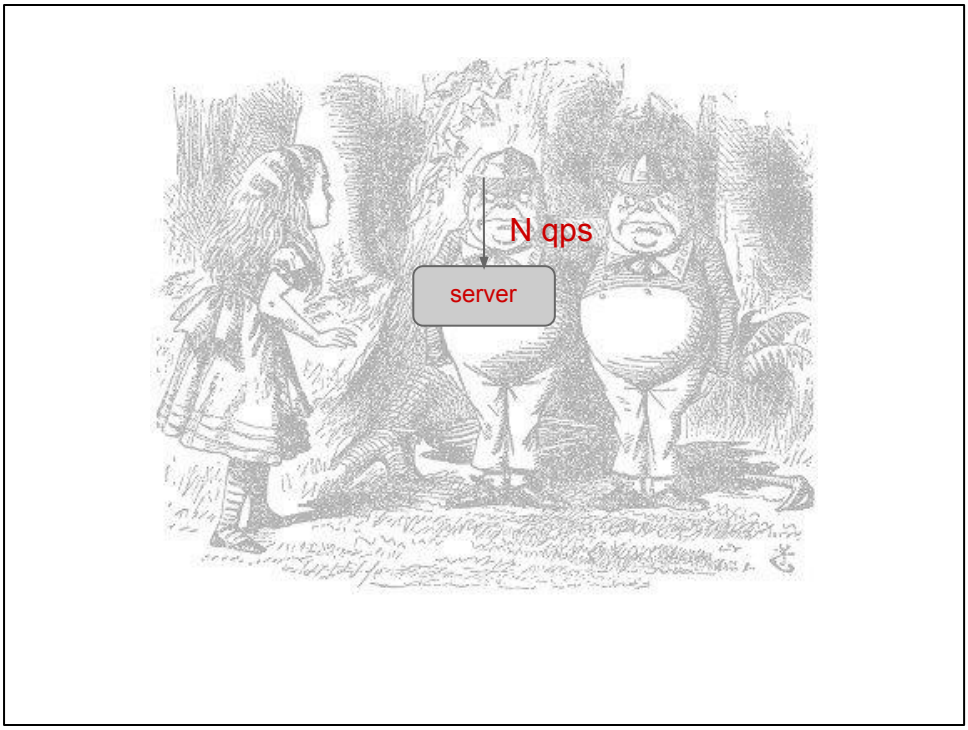
We talk about “horizontal” and “vertical” scaling:

- Horizontal scaling is balancing load across lots of identical servers, each of which can be relatively small in terms of machine footprint.
- Vertical scaling is running few ever-larger server instances on big, typically expensive machines.

We try to scale horizontally rather than vertically - we’re going to have to anyway, both due to the practical limits of our machines and network, and for high availability.

Another advantage of horizontal scaling is that we can add capacity in smaller increments, which is a good thing from an economics and utilization perspective.

Image from illustration of Tweedledum (centre) and Tweedledee (right) and Alice (left); by John Tenniel. http://en.wikipedia.org/wiki/File:Tennieldumdee.jpg. Public domain.



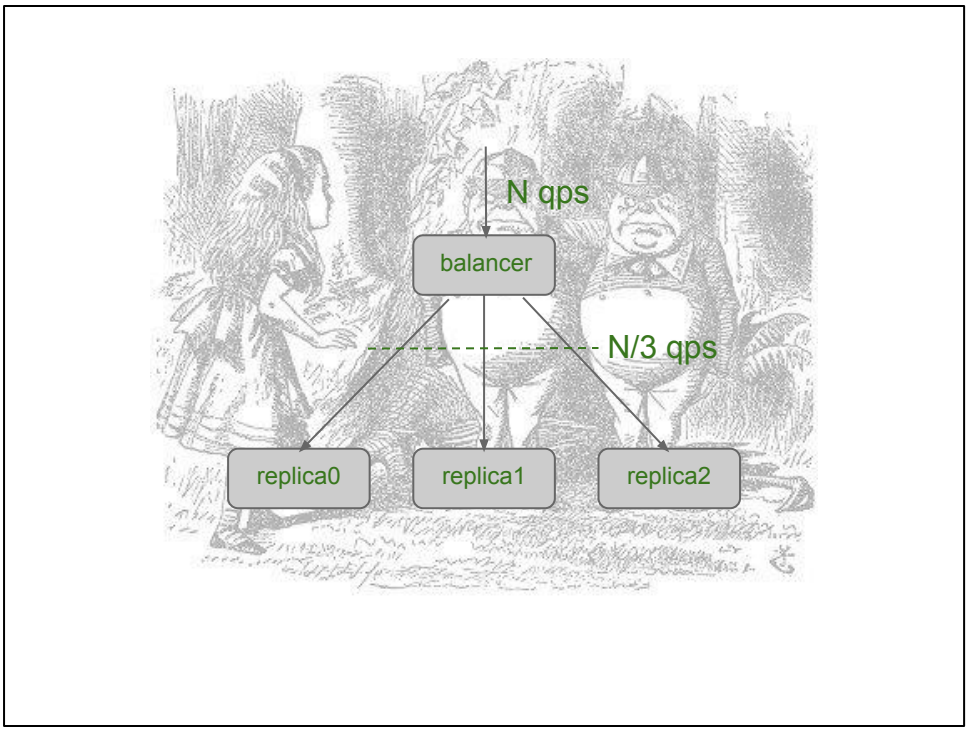
We can view this in terms of failure domains. A failure domain is the set of things (for example site availability, network capacity, etc.) lost when a particular system component fails.

Say we plan to serve N qps at peak.

If all N queries are hitting one server, and the machine it's running on falls off the network, or gets kicked over for an automated kernel upgrade, we drop everything on the floor.

We have one big failure domain.

Image from illustration of Tweedledum (centre) and Tweedledee (right) and Alice (left); by John Tenniel. http://en.wikipedia.org/wiki/File:Tennieldumdee.jpg. Public domain.



Instead, if any one of 3 load-balanced replicas can safely handle all N queries, we can lose one to accident, one to maintenance and we're still smiling.

Notwithstanding whatever small number of queries get dropped as the load-balancer catches up with the state of its backends.

You might have heard this called "N+2 redundancy".

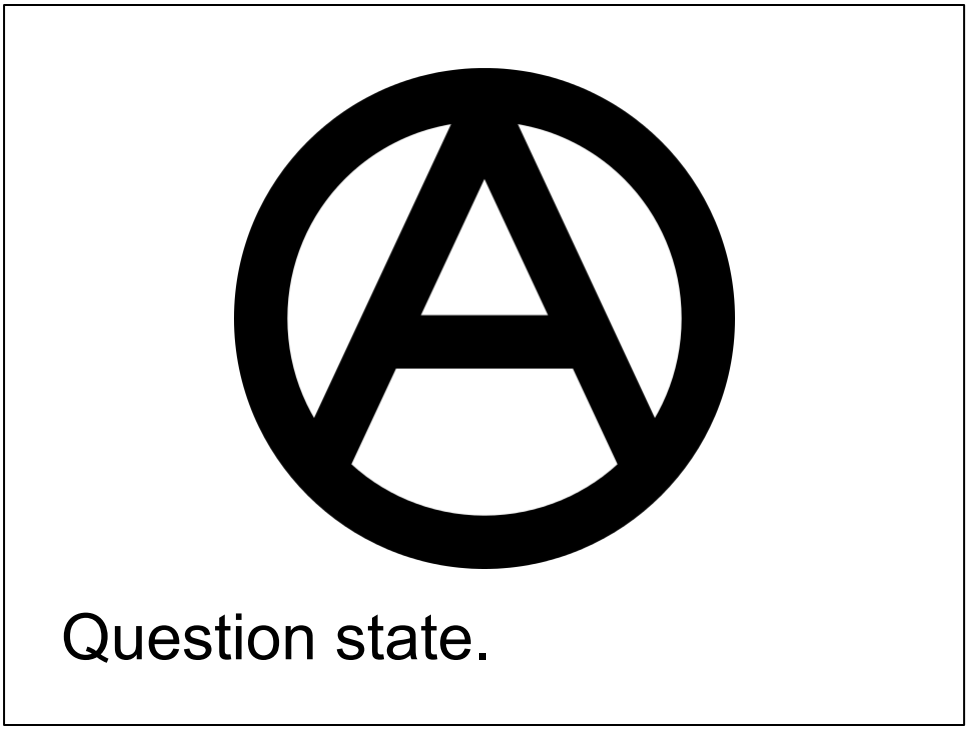
To expand further, if we're running 15 replicas, each of which can handle N/10 queries, we can lose a third of them and still serve peak traffic.

We can reason like this at several levels: for example, we could plan for N+1 redundancy within a single datacenter, and N+2 globally - that is, we can lose 2 independent datacenters and keep serving.

Note there are tradeoffs in running many replicas; for example, we may need to duplicate large datasets across machines (in memory or otherwise) to avoid contention at a central resource.

We also need to take failure at the balancing layer into account.

Image from illustration of Tweedledum (centre) and Tweedledee (right) and Alice (left); by John Tenniel. http://en.wikipedia.org/wiki/File:Tennieldumdee.jpg. Public domain.



Our web application doesn't need to be very complex before we want to persist some kind of mutable state across requests, for example:

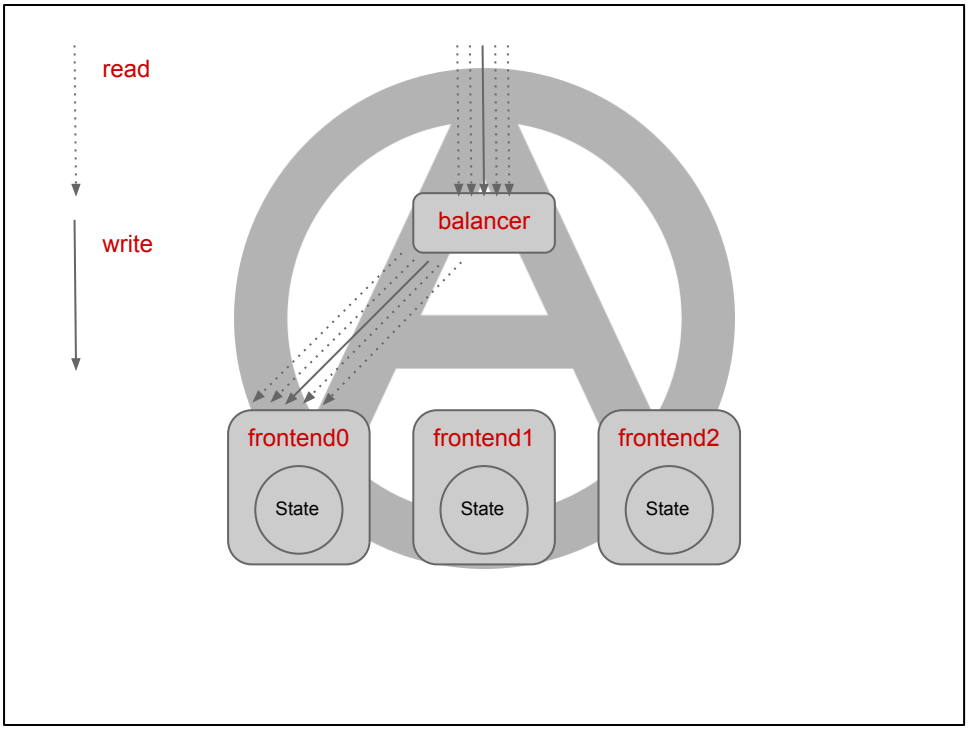
- objects in a database;
- a shared document in memory;
- or simple key=value pairs associated with a user.

We have to be careful with where we put such state in our serving system.

Since an arbitrary request can change a particular object's state, successive requests for it need to hit the same place.

This presents a challenge to our "horizontal" model of scaling, but before we discuss that, let's treat the "stateful" part of our system as relatively opaque and talk about how we can handle it in terms of structure.

Image from Basic traditional circumscribed "A" anarchy symbol. http://en.wikipedia.org/wiki/File:Anarchy-symbol.svg. Public domain.



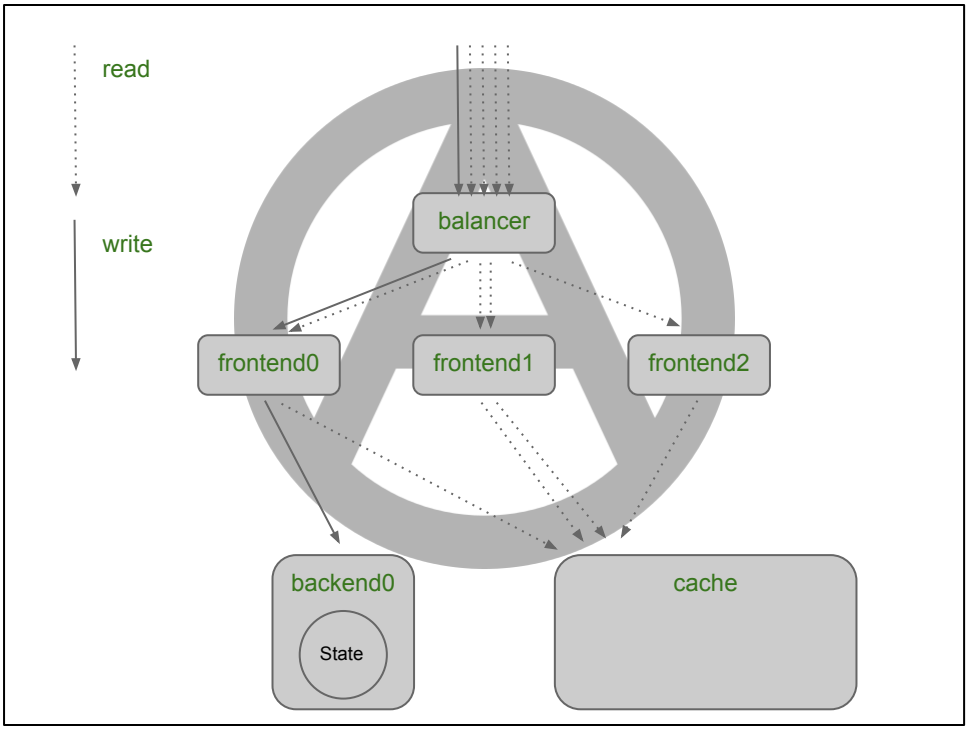
In the worst case, our frontend serving layer maintains some cross-request state, and all read or write requests that need it must hit the same server.

The balancing layer needs to maintain “client → frontend” affinity across all requests.

Note that this is something of a disaster for scalability - we have a balancing scheme which is now doing the opposite of balancing.

This is frequently a small site’s first stop on the “scaling trail”.

Image from Basic traditional circumscribed "A" anarchy symbol. http://en.wikipedia.org/wiki/File:Anarchy-symbol.svg. Public domain.



Instead, it's often useful to separate the read and write paths for servers that maintain state, particularly given that for many applications writes are rarer than reads.

This enables more even distribution of load on the most frequent request path.

So depending on the application and the kinds of transforms we're doing with requests and responses, it may be possible to push the state down towards the leaves of the request tree.

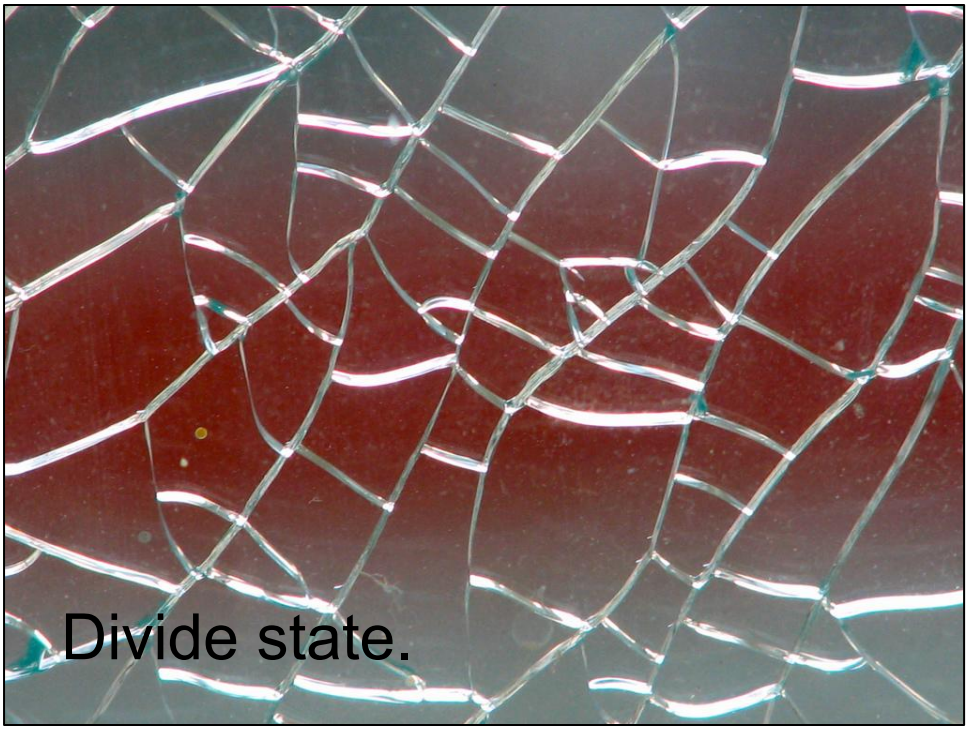
In general, if it's not essential for a particular serving layer or server type to maintain state across requests, it's best that it doesn't: we'll be more easily able to scale and reason about the system.

In this example, we have an independently-scaled frontend serving layer that shares a read cache.

Writes go directly to the state server, which is responsible for updating the cache.

Note that there are consistency problems with this approach, but we've found that variants on this scheme work for many applications, both at the frontend and deep within the serving stack.

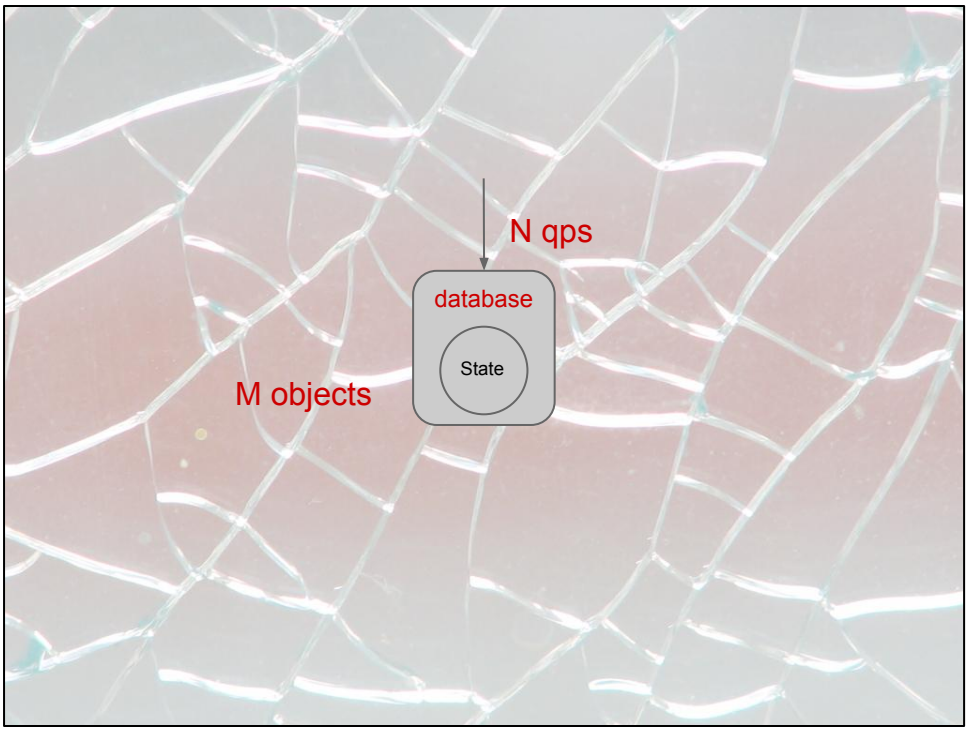
Image from Basic traditional circumscribed "A" anarchy symbol. http://en.wikipedia.org/wiki/File:Anarchy-symbol.svg. Public domain.



So how do we handle horizontal scaling of components that maintain state?

This problem, and the pattern of its solution, is analogous to that of replication discussed earlier; again we're dealing with the capacity of a resource, its failure domains, and how to distribute it.

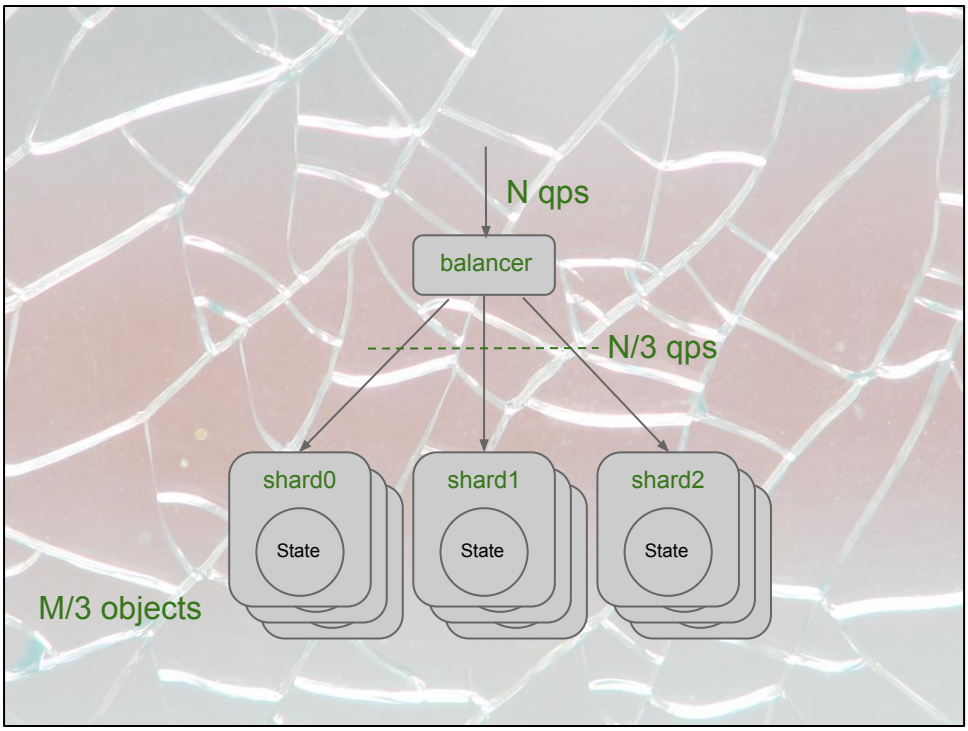
This time the resource is some set of state rather than just request-handling capacity. As we'll see, we may need to scale the two independently.



Here we have a single database, possibly replicated somehow, but as long as it keeps growing, we need to scale vertically.

Failure means the whole state space is unavailable.

Luckily, this is (or can usually be made) an “embarrassingly parallel problem” - where it takes little effort to separate it into a number of parallel tasks.



So, we “shard” the data.

As an example, we can assign each database object a unique ID, in such a way that the IDs are evenly spread over a large space. Then we assign ranges within that space to specific servers.

Each of those ranges is what we call a shard, or a slice of the data.

We teach our load balancer to look at the ID of each incoming request and send it to the correct server. It might be better to call it a “sharder” in this configuration.

We use this technique a lot at Google. There are various ways of implementing a system like this; for example, it’s not necessarily the case that the request routing must be done by a load balancer.

Note the difference to simple replication: if we want to maintain availability of shard0’s data when its server falls over, then we need a second availability dimension, indicated here as shard replicas.

Note also that this isn’t so different to the “client → frontend” affinity diagram we saw earlier, except here it’s our explicit intention to shard the state and concomitant load amongst multiple servers.

As our site grows, we can add more servers, and increase the number of shards, so

we can keep scaling horizontally.

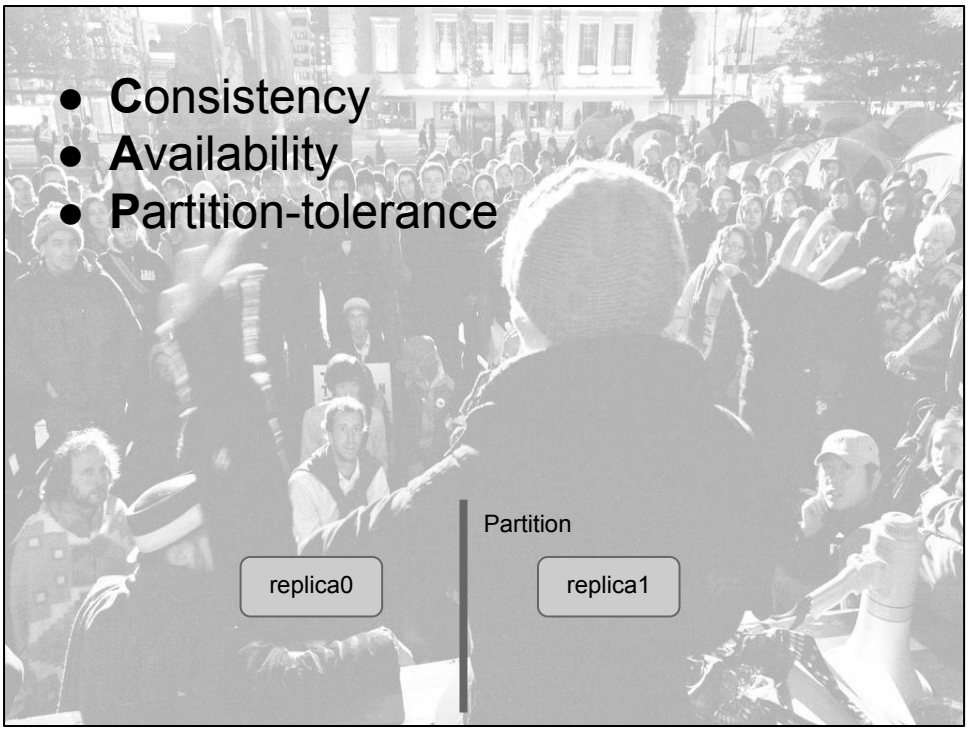
*Re*sharding can be a problem, depending on how it's done. For example, if we have a user database and we shard by some modulus of a hash of the username, our sharding stays stable and roughly equal as we add new customers, but when we mod over a new number we have a headache.

Among the many things I've been glossing over in this talk is how the components of our distributed system can come to any agreement.

In the face of ubiquitous failure, how can a set of processes communicate with one another and reach consensus on some piece of state?

As an example, how can a set of "master-slave" replicas of a data shard decide which is currently the write master?

This is a classic problem in distributed systems, and a set of "consensus algorithms" has been developed in answer to it.



A brief (but related) digression. As its author Eric Brewer puts it, the CAP theorem states that any networked shared-data system can have at most two of three desirable properties:

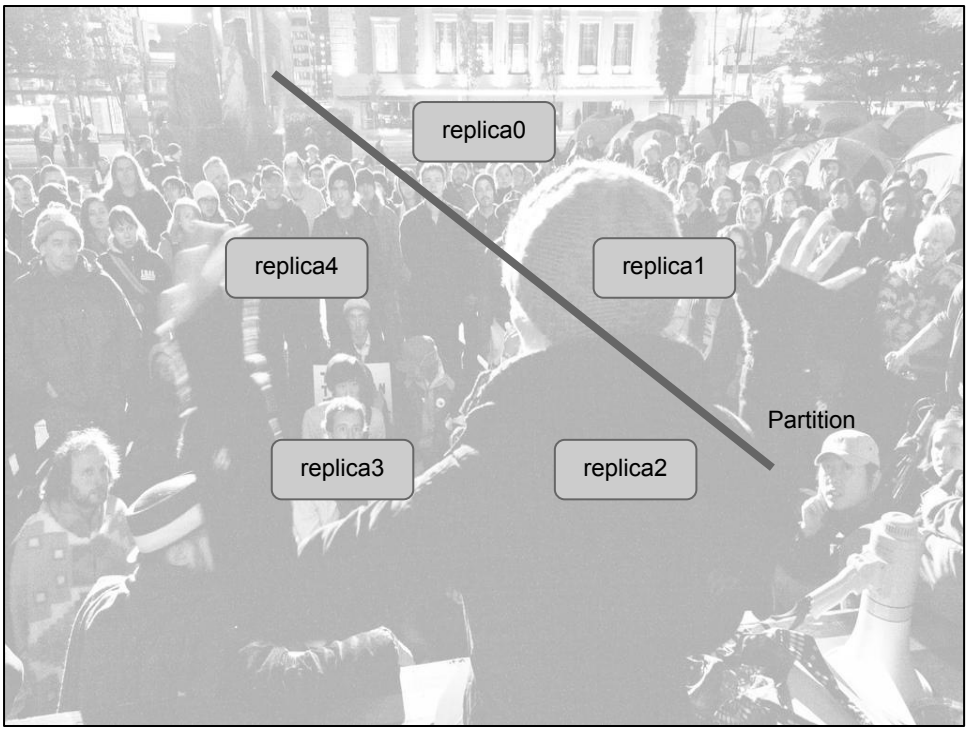
- **C**onsistency equivalent to having a single up-to-date copy of the data;
- high **a**vailability of that data (for updates); and
- tolerance to network **p**artitions.

If we consider two data replicas located either side of a network partition, they can only be

- consistent if updates are not allowed at both nodes - sacrificing availability;
- available if updates are allowed at both nodes - sacrificing consistency;
- consistent and available if the partition is not there - sacrificing partition tolerance.

This is a really useful tool for reasoning about the problems that face us when coordinating servers in a distributed system; and this is the environment that consensus algorithms have to work in.

Note that you need to consider your clients as part of the system when reasoning about it in this way.



The consensus algorithm at the base of many large-scale distributed systems is Paxos.

For example, at Google it forms the foundation of the Chubby lock service, which in turn is used for coordination of servers and to store small amounts of metadata for e. g. Bigtable and Colossus.

Each datacenter has a Chubby “cell” consisting of five replicas; each replica maintains a local log which is synchronized with the others via Paxos, and used as the basis of a replicated database.

One of these replicas is master; if it fails, another takes over and serves from its local log. Clients contacting a replica are redirected to the current master.

As long as three replicas remain on one side of a partition, they can agree on a master and proceed.

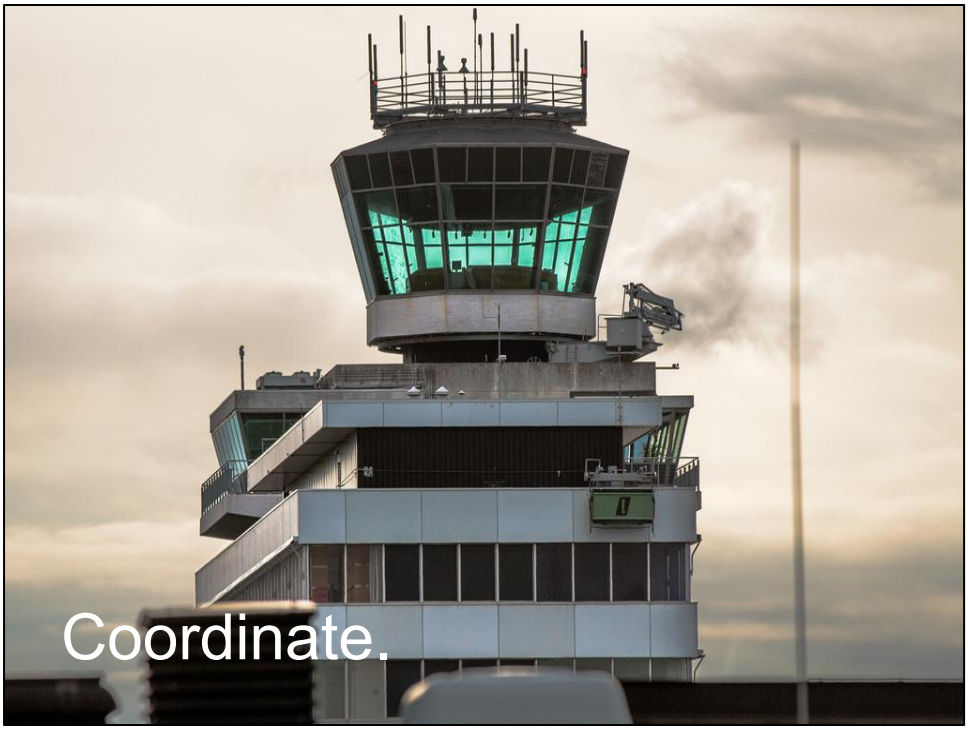
“Stranded” replicas will not accept writes: so Chubby sacrifices **a**vailability for **c**onsistency and **p**artition-tolerance.

The upshot is that clients can access a simple, reliable (but limited) filesystem in which each file or directory can also act as a (single) writer or (multiple) reader lock.

This facility makes many kinds of distributed coordination problems much easier.

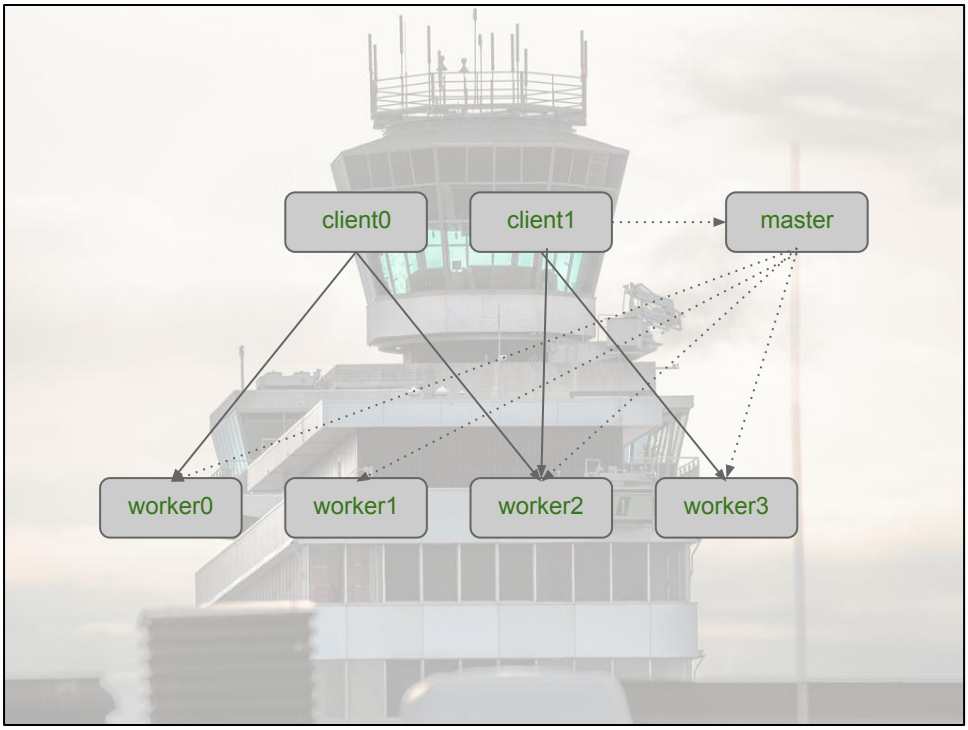
For example, a set of “master-slave” replicas of a data shard can decide which is currently the write master by each attempting to acquire a write lock on a ‘master’ file.

Niall will discuss some subtleties of consensus and failure in his talk later.



Once we have some kind of coordination primitives built on consensus, we still need patterns of coordination at scale.

In particular, if we have thousands of similar jobs that need to coordinate to achieve some goal, having them talk amongst themselves can rapidly become expensive and unwieldy.



A pattern that has worked well, repeatedly, and at surprisingly large scale at Google is to have a coordinating “master” process.

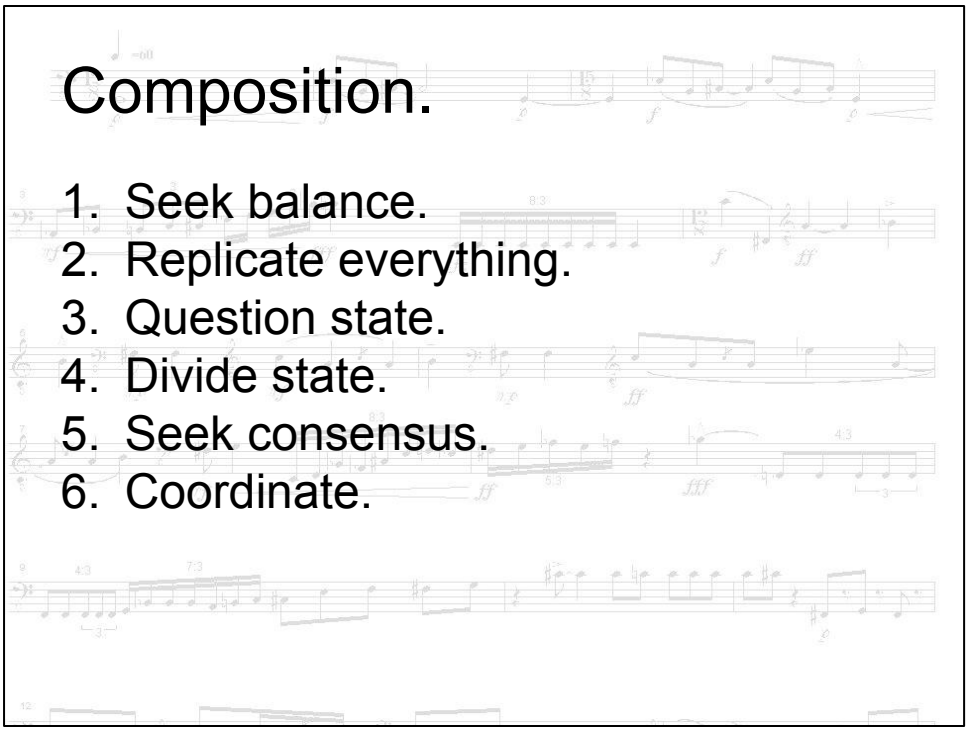
It’s important that this be, for the most part, outside the serving path: so clients don’t necessarily need to hit the master to get service.

The occasional lookup or query is fine, but if frequent operations have to go through a single master, that won’t scale.

The master can gather and maintain “global” state across its fleet of workers, and assign them different roles as necessary.

A good example is the Bigtable “master → tabletserver” communication, via which the master can assign each tablets to serve - for example when it detects that some other tabletserver has fallen over.

This makes reasoning about the whole system’s behaviour much easier, and is also a good place to make the whole system’s status transparent per our discussion above.



Recap: means of composition.

Control fan-in with request/response trees.

Replication for availability and balancing.

Be careful where to put state, and shard it to scale horizontally.

Use consensus to coordinate jobs.

A master outside the serving path can be a great coordination point.

Questions?

Abstraction.

We've talked about how we compose the parts of a large-scale system.

What are we building? At what point do we step back and say "OK, we have a new basic block we can build more upon"?

Image from Composition VII (The Three Graces) by Theo van Doesburg. http://en.wikipedia.org/wiki/File:Theo_van_Doesburg_Composition_VII_(the_three_graces).jpg. Public domain.

There's a lot of talk of clouds and service-oriented architectures.

The latter, at least, with good reason: the service is a useful and successful unit of abstraction in large-scale systems.

A service is a clearly specified API with a discovery or addressing method and (hopefully) well-defined dependencies and characteristics.

The client doesn't need to understand precisely what is happening underneath, and that is part of the goal:
to bundle complexity of state and execution in much the same way that our programming languages allow us to.

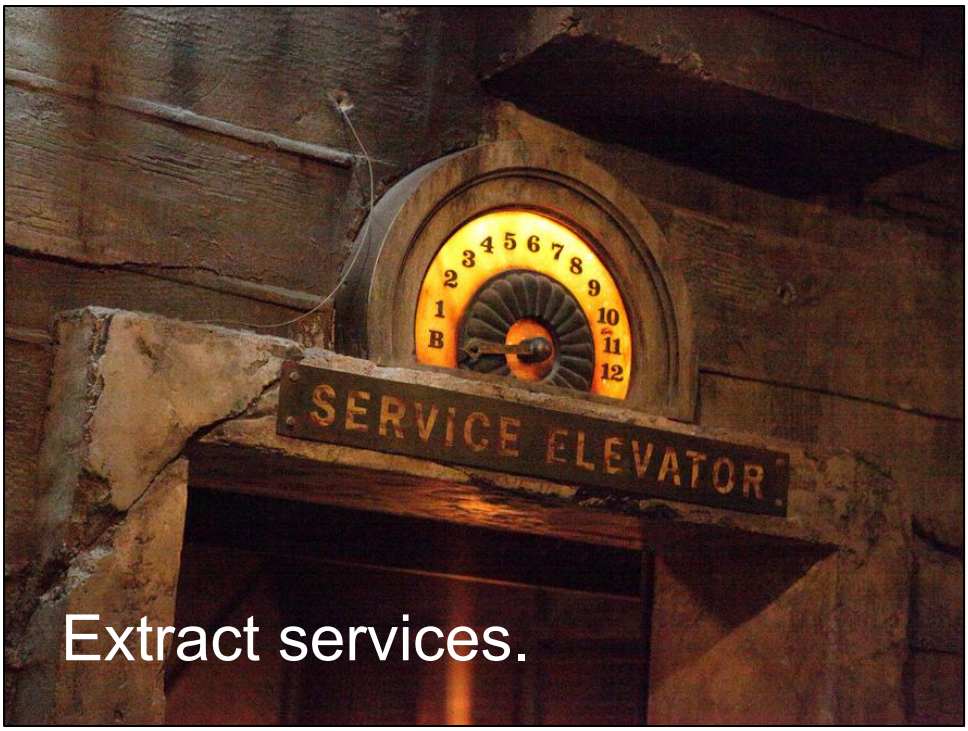
We can re-implement a service with minimal effect on clients; small teams can work independently, for example on infrastructure or application services.

So when building a large system, consider how to break it down as relatively independent services, and how it will interact with lower-level services.

This makes the design considerably easier to reason about.

A mature service should provide the numbers we need to do back-of-the-envelope design work against it, in the form of service-level objectives - but that's a topic for another day.

Image from Crews Build Up the Foundation for the New Belmont Road by NCDOTcommunications. http://www.flickr.com/photos/ncdot/6276129967/. Creative Commons Attribution 2.0 Generic.



We say to build services, but ultimately we're trying to disassemble them.

If we're proposing an upgrade, we have to somehow completely replace the functionality that is already in use: consider Olivier's talk about Moonshot earlier this morning.

If we're proposing new functionality from scratch, we know we'll have designed it wrong (because we have no existing usage for accurate modeling), so we know we'll have to replace it really soon.

If we build a service without planning to disassemble it later, we've accidentally volunteered for the most boring kind of maintenance job out there.

Right up until the service has too many users for the scale it was designed for, at which point we have an unpleasant job.

So as we design and build our services, it's healthy to take an experimental or "refactoring" approach to it.

Analogous to "Extract Class", then, we might "Extract Service". If we're building everything with disassembling it in mind, this should not be too difficult.

This is a good way to experiment with and discover the right boundaries between "application" and "infrastructure" services.

If we are trying to extract an infrastructure service, we try to find another customer for it as soon as possible.

Quick feedback from multiple customers is vital to understanding what should properly be pushed down into infrastructure and what should stay in the application.

Abstraction.

1. Build services.
2. Extract services.

Recap: service-level abstraction.

The main unit of abstraction is a service.

Design services, but be ready to take them apart, and to extract infrastructure or sub-services from them.

Questions?

Image from Composition VII (The Three Graces) by Theo van Doesburg. http://en.wikipedia.org/wiki/File:Theo_van_Doesburg_Composition_VII_(the_three_graces).jpg. Public domain.

No matter what system we build, no matter what we design, the one thing we can be sure of is change.

Poor Heraclitus knew that. So how do we cope with constant change in large web systems?

Image from Heraclitus by Johannes Moreelse. http://en.wikipedia.org/wiki/File:Utrecht_Moreelse_Heraclite.JPG. Public domain.

We are one of the biggest risks to our own systems' availability.

In a fast-moving business environment, we need to introduce change to add features, fix bugs, optimize, and scale. How do we do it safely? How do we protect us from ourselves?

The key here is flexibility amid the changes.

First, try to change one thing at a time. There are a lot of variables in large web systems, and if we tweak ten at once it'll be hard or impossible to untangle which caused a 5% user-visible latency increase across our fleet.

One good tactic is to decouple server upgrades from new features by guarding them each with a flag or other configuration. Then we can roll out and switch them on successively while watching for unwanted effects.

Next, never change that one thing everywhere at once. Run it in a test environment first, with synthetic or recorded traffic; then run it on one server or some small percentage of our fleet. This is our "canary": think of a coal mine.

When we've gained confidence that it's OK, roll it out exponentially: 1 server, 1 rack, 1 datacenter, 2 datacenters, 5, 10, the whole fleet. At each point, leave enough time

Never, ever push anything late on a Friday afternoon, before you go on vacation for 3 weeks. Pick a release and update schedule that makes sense in terms of developer and operator availability, and slack times for the site.

For big, user-affecting changes, consider using percentage rollouts and A/B testing. This gives us a chance to gather data from a proportion of our users and verify changes before rolling it out everywhere.

I mentioned touching on more topics related to monitoring earlier; this is one of them.

Capacity monitoring tends to be neglected in small systems; when we have enough machines for basic redundancy, actually reaching capacity can seem a distant threat at best.

Large-scale systems typically punish this kind of thinking: modest growth can amount to a great deal of extra data at rest or in flight.

If we are caught out here, it can be disastrous: given the lead time for building new datacenter capacity, we can end up with a significant gap between what the business requires and what we can deliver.

Track the capacity metrics of our service, and project them forward, including estimates of seasonal changes and the business' plans.

Be aware that environmental changes can impact our capacity plans: for example, the shiny, wildly popular new camera that takes larger photos by default and starts squeezing our upload bandwidth.

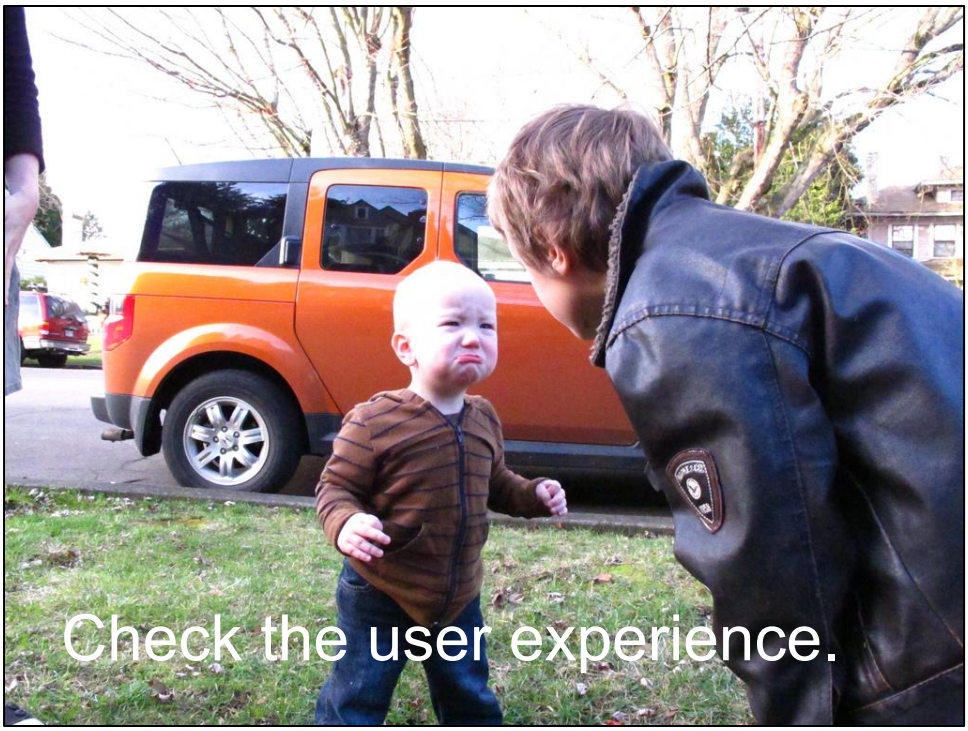
No capacity plan survives contact with the real world, so be ready to adapt it.

If we start eating into our N+2 reliability buffer because we don't have enough capacity, we'll make it harder to actually do the upgrades, power work, etc. that we need.

So we should either have explicit spare capacity or a solid plan to deliver it in line with expected growth.

Finally, Murphy's law: the most inconvenient part of the system will fail at the most inconvenient time.

Ask yourself what and when this might be and think about how to head Murphy off at the pass.



Finally, none of the above will make any difference if some unknown issue is hurting our users.

An antipattern in monitoring large systems is to focus on causes we're aware of rather than user-facing symptoms it might be harder to monitor: for example, paging on "5% of application servers flapping in Dublin" rather than "99th %ile user-observed latency at Dublin is through the roof."

Lots of things could cause the latter.

If we're only monitoring the former, our users call us to tell us our stuff is broken. Then we figure it out and add new monitoring that'll catch the new cause and page us next time.

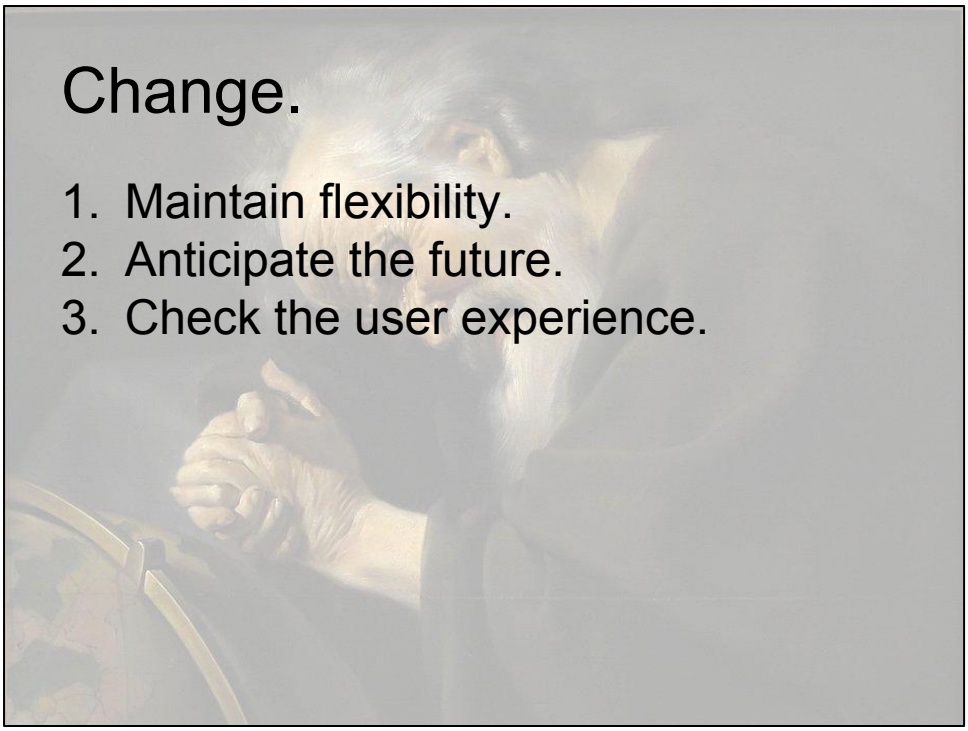
We make our users part of monitoring system, and that's a recipe for a bad experience.

Users care about surprisingly few things: availability, latency, durability, features, more or less in that order. As we mentioned earlier, we've seen time and again that users really care about speed.

We need to check what the experience is like for real users under real conditions - for example, with the same bandwidth and latency as real users will experience.

That giant image we just accidentally put on the front page loads fine locally, but users on phones or tablets with a 3G connection won't be happy.

So even if it's hard, check the user experience as closely as you can.

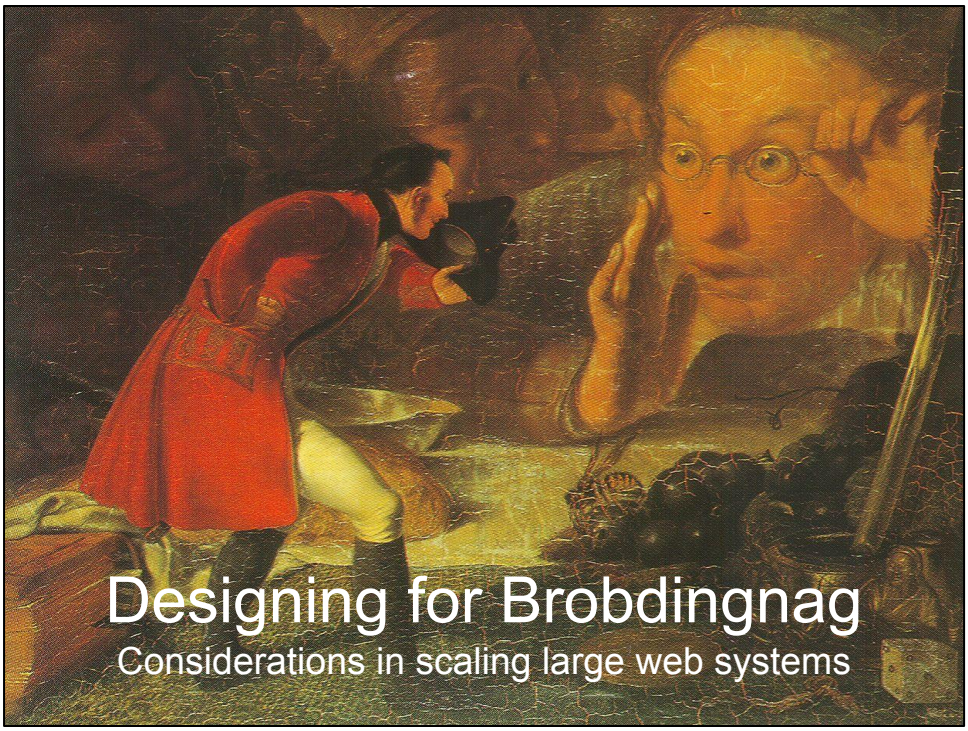


Recap: handling change in large systems.

Don’t change too many things at once. Roll out carefully.
Be careful about capacity planning.
Always check the user experience.

Questions?

Image from Heraclitus by Johannes Moreelse. http://en.wikipedia.org/wiki/File:Utrecht_Moreelse_Heraclite.JPG. Public domain.



That brings us to the end of this talk.

We hope you'll find the ideas we presented here useful, both in the workshops today and as you reflect on them in future.

Let's briefly review the topics we covered.

<Recap slides from Environment, Fundamentals, Composition, Abstraction and Change>

Questions?

Image from "Gulliver in Brobdingnag" by Richard Redgrave. http://en.wikipedia.org/wiki/File:Szene_aus_Gulliver%27s_Reisen_-_Gulliver_in_Brobdingnag.jpg. Public domain.

Some resources we like & think you might find useful.

Notes on Distributed Systems for Young Bloods by Jeff Hodges.

Hints for Computer System Design by Butler W. Lampson.

End-to-end Arguments in System Design by J.H. Saltzer et al.

Fallacies of Distributed Computing Explained by Arnon Rotem-Gal-Oz.

7 Ways to Handle Concurrency in Distributed Systems by Colin Scott.

Latency Trends by Colin Scott (an update to “numbers every programmer should know”).

Crash-Only Software by George Candea and Armando Fox.

Building Software Systems At Google and Lessons Learned by Jeff Dean.

CAP Twelve Years Later: How the "Rules" Have Changed by Eric Brewer.

The Consensus Protocols series by Henry Robinson.

- Two-phase commit;
- Three-phase commit;
- Paxos.

Distributed Systems and Parallel Computing at research.google.com.