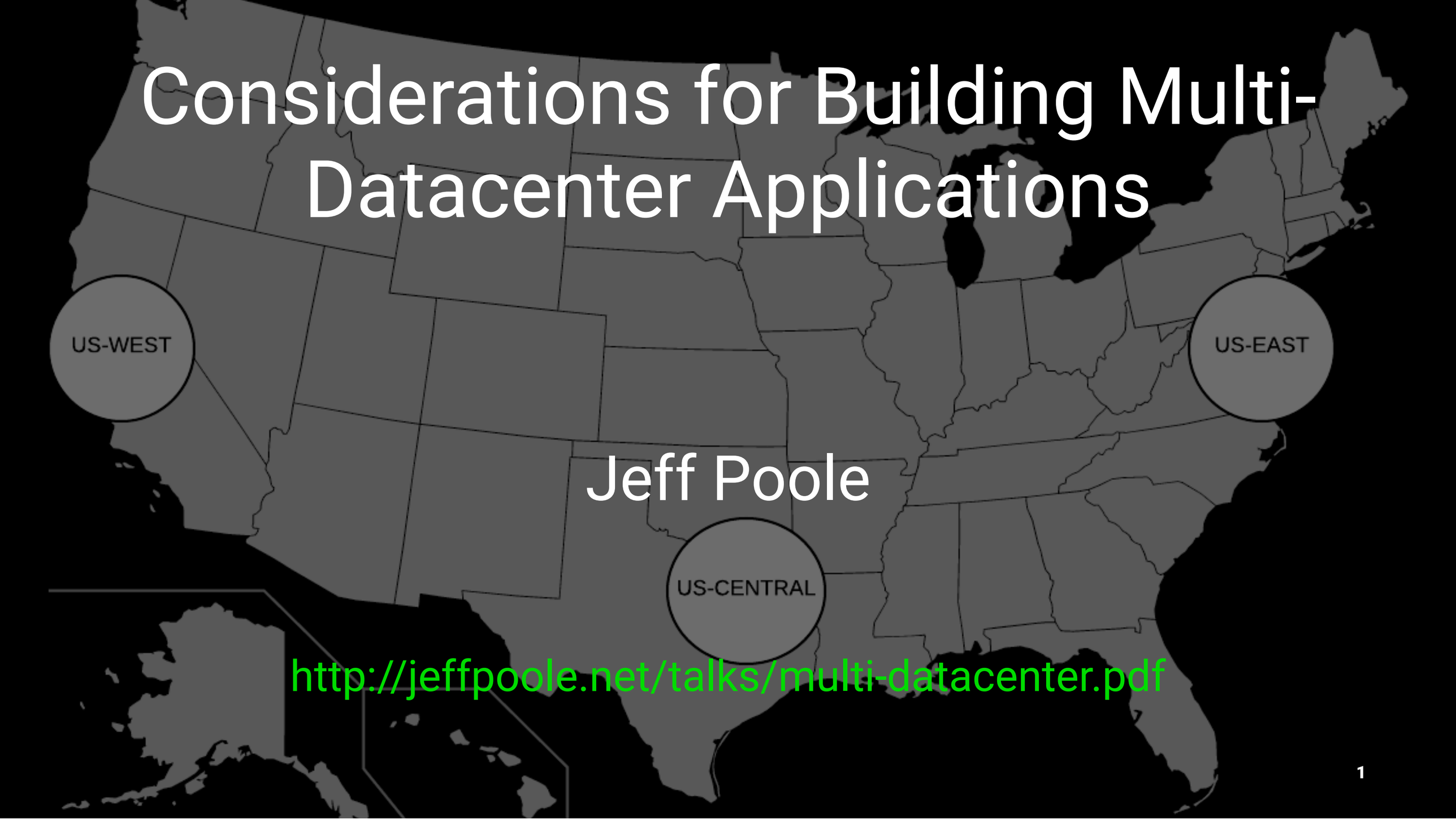
Considerations for Building Multi-Datacenter Applications



Jeff Poole

http://jeffpoole.net/talks/multi-datacenter.pdf

Running a geographically distributed system can be hard.

But, while it can be challenging, it can also provide benefits that are hard to get any other way.

I believe it is like giving your application superpowers.

Hopefully, I will help you bring those superpowers to your own organizations.

Who am I, and why do I care about this?

Jeff Poole

~~**Principal Software Engineer**~~ **DevOps Manager**

vivint.SmartHome™

@_JeffPoole / jeff@jeffpoole.net

My name is Jeff Poole, and I work as a DevOps Manager at Vivint SmartHome. We are a provider of smart home and alarm systems across the US and Canada.

Your alarm system has to be reliable. Downtime is not ok.

If you have a smart home, but it's buggy or slow to respond, you get frustrated.

In a former job, I worked in telephony, which also has high expectations for uptime and latency.

What will I cover?

- Not a how-to
- General concepts
- Fill your toolbox, so you can design your own
- I bring more questions than answers

First, let me explain what this talk will and will not cover. This **isn’t a talk about the right way** to create an application that spans several sites. In fact, **this talk isn’t about the right way to do anything**. Every application is different.

What I will discuss are **the problems that need to be solved** in a multi-datacenter environment, and **some possible ways to solve them**. I will give examples as best I can to show how these approaches could apply to different applications.

I hope that this will **give you the right questions to ask** when architecting your own distributed applications, or solutions to explore when trying to solve problems you’ve run into.

Why go through the pain?

Let’s discuss the reasons why someone might want to have code in different datacenters.

I see two big, obvious reasons to have multiple physical locations for your application.

Reason: Resilience

- 24/7 uptime expectations
- No cloud provider or datacenter has 100% uptime
- "It's not *my* fault!" doesn't cut it

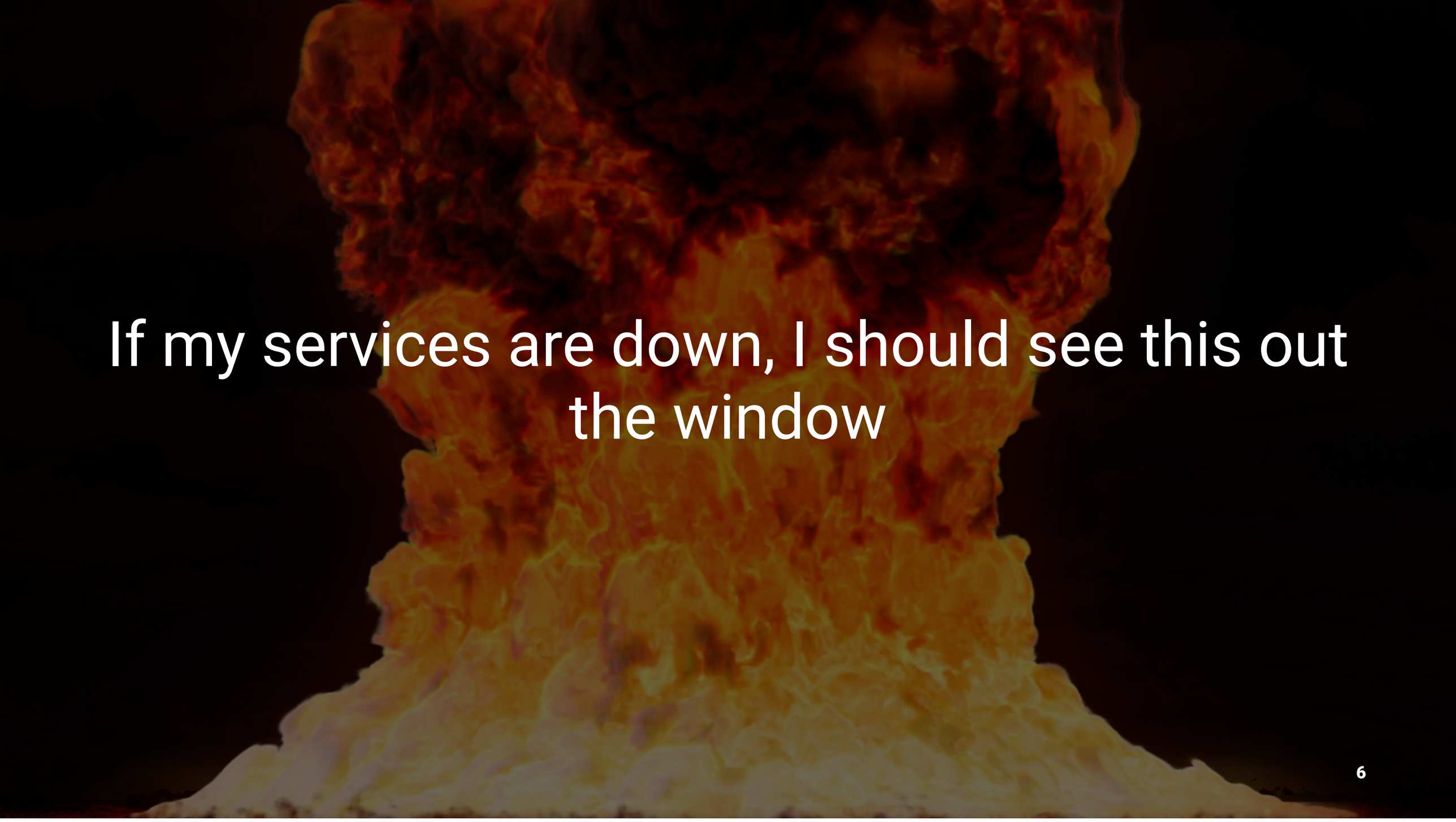
We live in a world where our applications are expected to be always up. Many, if not most, of us are accountable 24/7.

No cloud provider or datacenter has 100% uptime. AWS has outages. Datacenters experience **fiber cuts**. I had a **core router misconfiguration** take down two datacenters for 20 minutes. I've seen generators for backup power where someone forgot to fuel them up or perform proper maintenance, and of course that only gets discovered when they are needed.

Natural disasters can take down whole regions -- for example, **Hurricane Sandy** affected several datacenters here in NYC, many because their generators or fuel tanks were flooded (a scenario I suspect most of them had not expected!).

If you've ever had to explain that **your system is down because a provider is down**, you've probably had the experience of realizing that isn't a very satisfying excuse. **Uptime is our responsibility.**

So we want to be always up.



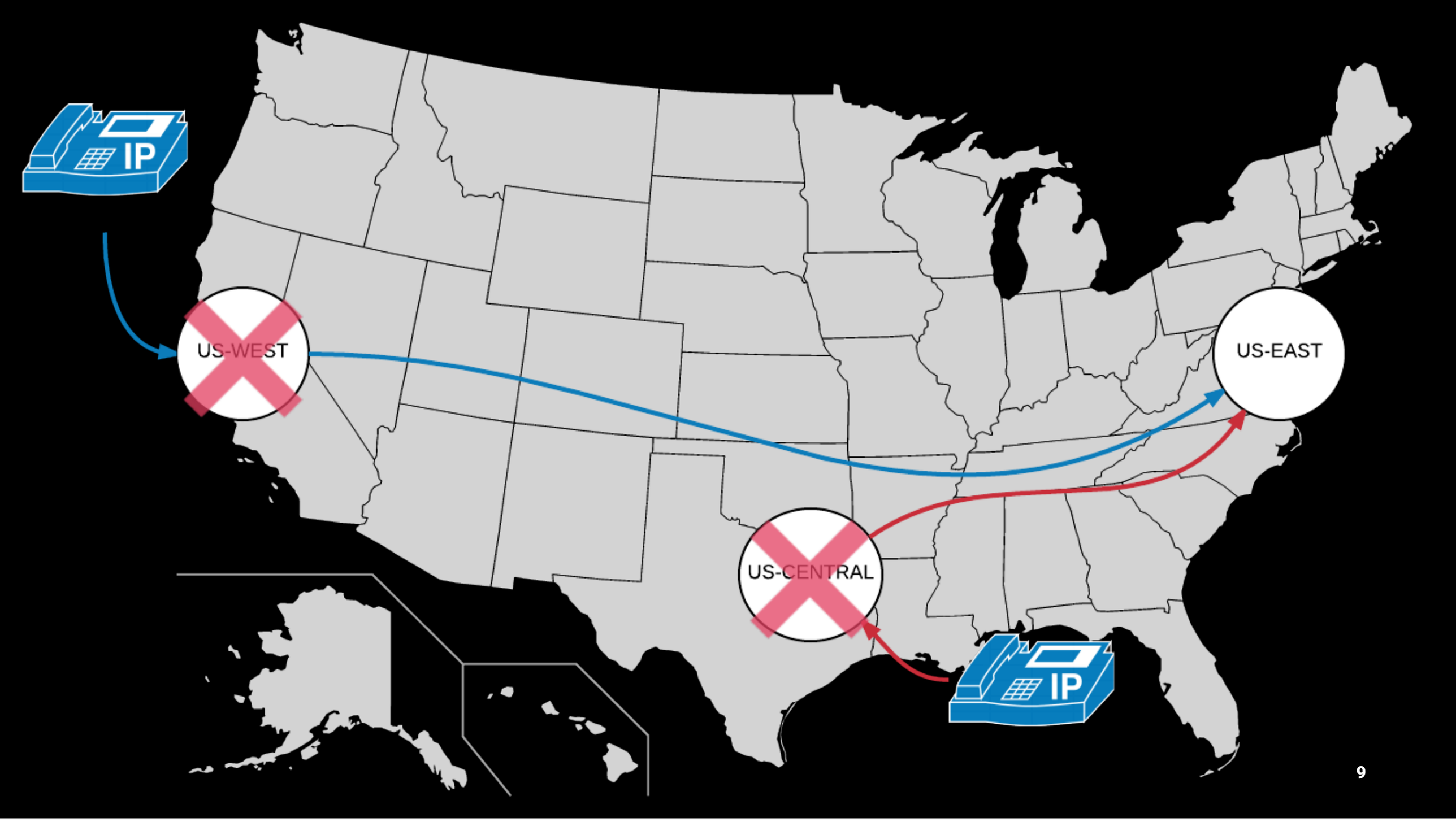
Short of an apocalyptic event, someone else’s outage shouldn’t make us go down.

I worked in telephony at a previous job, and one day we got reports that calls from our new system (which I was working on at the time) weren’t able to complete to our old system. My system was in three datacenters at the time.

After tracing a few call attempts, I noticed that all of the calls were ending up in our New York datacenter at some point.

Turns out an IP from that datacenter hadn’t been whitelisted to talk to our old system. A simple misconfiguration problem.

Why were all the calls going through New York?



Turns out a critical service was down in BOTH of our other two datacenters.

I learned two things that day.

First, that the right architecture can keep your system running just fine in the face of significant system failures.

Second, that our monitoring systems were terrible, and I had a lot to learn.

Why not use a "warm" backup site?

Because if you have ***never*** actually served clients through it, you ***probably can't***

Think Schrödinger’s Backup:

"The condition of any backup is unknown until a restore is attempted"

Some people make their first step by having a second datacenter that is a "warm" backup. One that gets a copy of the data and theoretically could run services

If you've never served clients out of that datacenter, how do you know it will work?

Like Schrödinger’s Backup, where you have no idea if a backup is any good until you've tried to restore it. You have no idea if you can run out of a backup datacenter until you try.

Best way to try is to **always serve production traffic** out of it!

Reason: Speed

- Speed is limited by latency between customer and datacenter
- Can't exceed the speed of light
- Typically 60-80ms US coast-to-coast

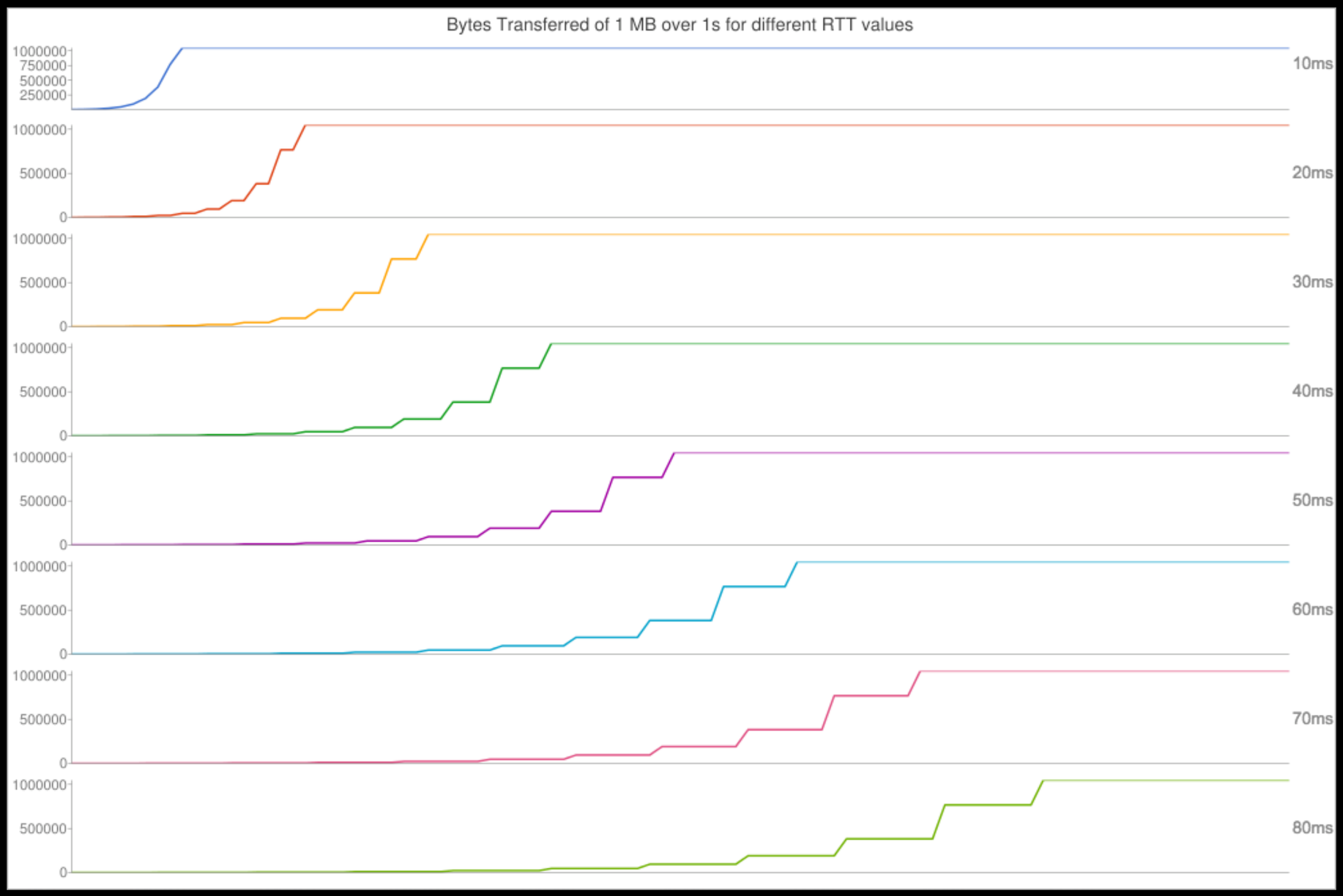
We want to be fast. You can only be so fast if your customer is on the opposite side of the globe.

Across the US, you generally see **60-80 ms of round-trip latency**. That kind of latency has a direct impact on things like real-time media (audio/video), but even for something as simple as downloading files over HTTP, latency can have a pretty dramatic impact.

Remember that **TCP slow start** means that the rate your bandwidth ramps up to its maximum value is **a function of latency**.

This is a visualization of a HTTP transfer slowed down 40x

Notice how it speeds up after every round trip



I graphed the expected bytes transferred for a **1 MB file** as a function of latency.

The x-axis is time, with the full length equal to one second.

You can see how dramatically latency affects a 1 megabyte download, and that would be a small webpage this day and age.

As download times start to be **measured in seconds**, customers get frustrated

Reason: Scale

- Limited ability to add capacity in one datacenter
- May be easier to reliably get 1/Nth the bandwidth in N datacenters than all in one place

Reason: Regulatory

- May be hard to find one location that meets all regulations

There are two other reasons that could come up, but aren't as compelling.

You might not be able to continue growing in your current location due to space or bandwidth. So you may need a new datacenter to scale. You may be so big there is no one datacenter that can handle your needs.

Once you are in two locations, it's much easier to move to three or four...

There may be regulations you can't meet in one datacenter. Maybe you have governments as clients who care where you store their data. Or consumer data laws.

Planning

Before going into the details, it makes sense to figure out what you need to consider.

For each piece of functionality in your application, you want to ask yourself...

Planning

Latency constraints

How quickly do you need to service user requests?

Are there asynchronous requests with different requirements?

How quickly do requests need to be serviced?

If you handle user requests **asynchronously**, how long until they need to take effect?

Planning

Change propagation

How long it takes for a change to become visible EVERYWHERE, not just to the user who initiated it.

Subtly different than latency.

This is about how long it takes for changes to data to become visible EVERYWHERE.

Users may see their own change sooner.

Planning

Support for full datacenter outages

Do you have to design for a full datacenter outage?

Don't forget to plan for that datacenter coming back...

Do you have to handle a full datacenter outage?

Nothing critical can be in only one datacenter. Every service needs to be able to fail over if a datacenter disappears.

Importantly, **work through the process of the datacenter coming back up** -- you don’t want a database node to come back up and assume it is a primary / write master when a node in another datacenter has already taken over that functionality.

Any database that doesn’t require **a majority to ack writes** before returning **can lose data**.

If the datacenter is down, you can't re-configure it.

Make sure you are happy with how things will come back online in scenarios such as network **connectivity loss** (servers keep running) and a **power loss** (servers boot back up and come online in a potentially unpredictable order).

Planning

Ability to overprovision

To handle a single datacenter failure, you need (N+1)/N times the resources you need to handle your load

More datacenters may require less hardware

If you need to handle a full datacenter outage, one thing to keep in mind is, you actually need less hardware the more datacenters you have.

If you have two datacenters, you need two times the minimum level of hardware to keep up with your traffic, because the loss of a datacenter takes out half of your capacity.

If you have three datacenters, you only need one-and-a-half times the minimum level.

If you had 11 datacenters, you'd only need 10% more than you would otherwise need if you assume no outages.

More datacenters -> less over-provisioning needed

Planning

Support for partial datacenter outages

What if one service is down in the current datacenter?

Is it worth the latency penalty to go somewhere else?

How do you decide where to go?

Do you want to handle a partial datacenter outage? If you never care about partial datacenter outages and you can overprovision all your services, you may be best off keeping as much of a request as possible in one datacenter. Deciding if you should route to a different datacenter can be challenging.

If a service is completely down in the current datacenter, that makes the decision easy, but then where to route that request? To the geographically closest option? To the one with the most spare capacity? Once you know what approach you want to take, how do you decide which datacenter is the closest or least loaded? The closest could be determined by a static list for each datacenter, but if you want to route based on capacity you need a dynamic system.

Planning

Support for partial datacenter outages

What if one service is ~~down~~ **overloaded** in the current datacenter?

Is it worth the latency penalty to go somewhere else?

How do you decide where to go?

If a service is just overloaded in the current datacenter, when do you decide to take the latency penalty to send the request elsewhere? Do you just want to route the initial request to a less loaded datacenter, or make the decision on a hop-by-hop basis? Is it possible that you might end up bouncing between datacenters and, if so, is the latency penalty worth it for

Routing from outside the system

When you have multiple datacenters, where should requests from users go?

Routing from outside the system

When you have multiple datacenters, what requests do you route where?

- User interaction
- Fixed hardware, or users with a known, fixed location

(:right-arrow:) **User interaction** -> all datacenters. If it all goes through one datacenter, you still have **a single point of failure** from a user perspective, and **no latency advantage**.

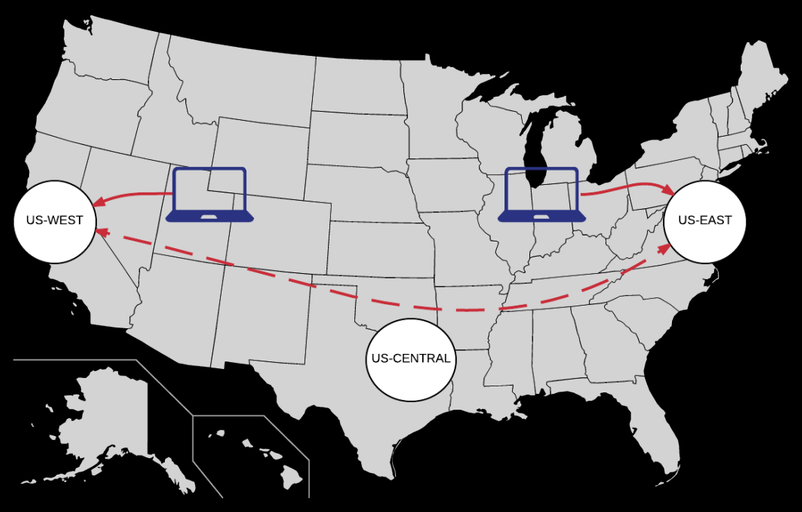
(ADVANCE) If there is **fixed hardware** you communicate with, or if a user can be expected to stay in one location enough, you can pin that interaction to the closest datacenter.

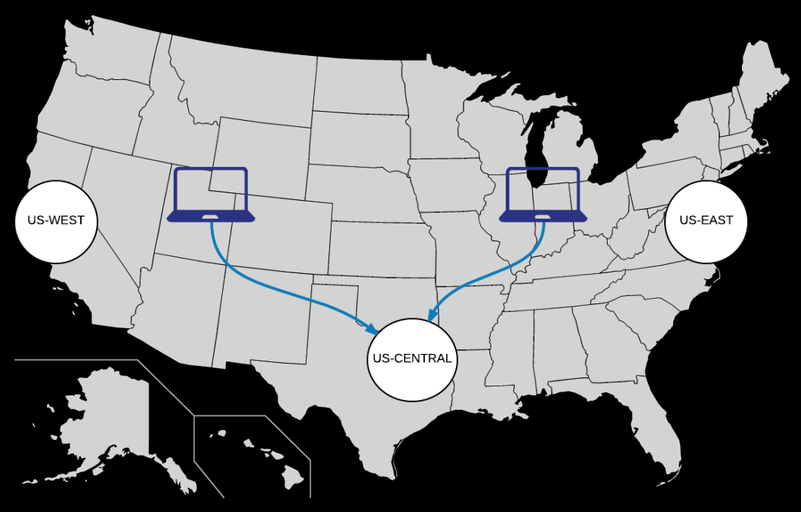
You should still be able to send them elsewhere in case of failure.

Routing from outside the system

When you have multiple datacenters, what requests do you route where?

- Matching two peers for real-time communication





When trying to **match two peers** (communication, gaming) you may want to direct **both sides to the same datacenter** since that may minimize end-to-end latency

Customers in Utah and Chicago maybe should both connect to a Dallas, TX datacenter instead of locally close decisions of Los Angeles and New York, with a cross-country hop between them

Once you know where to route requests, **how do you make that happen?**

DNS

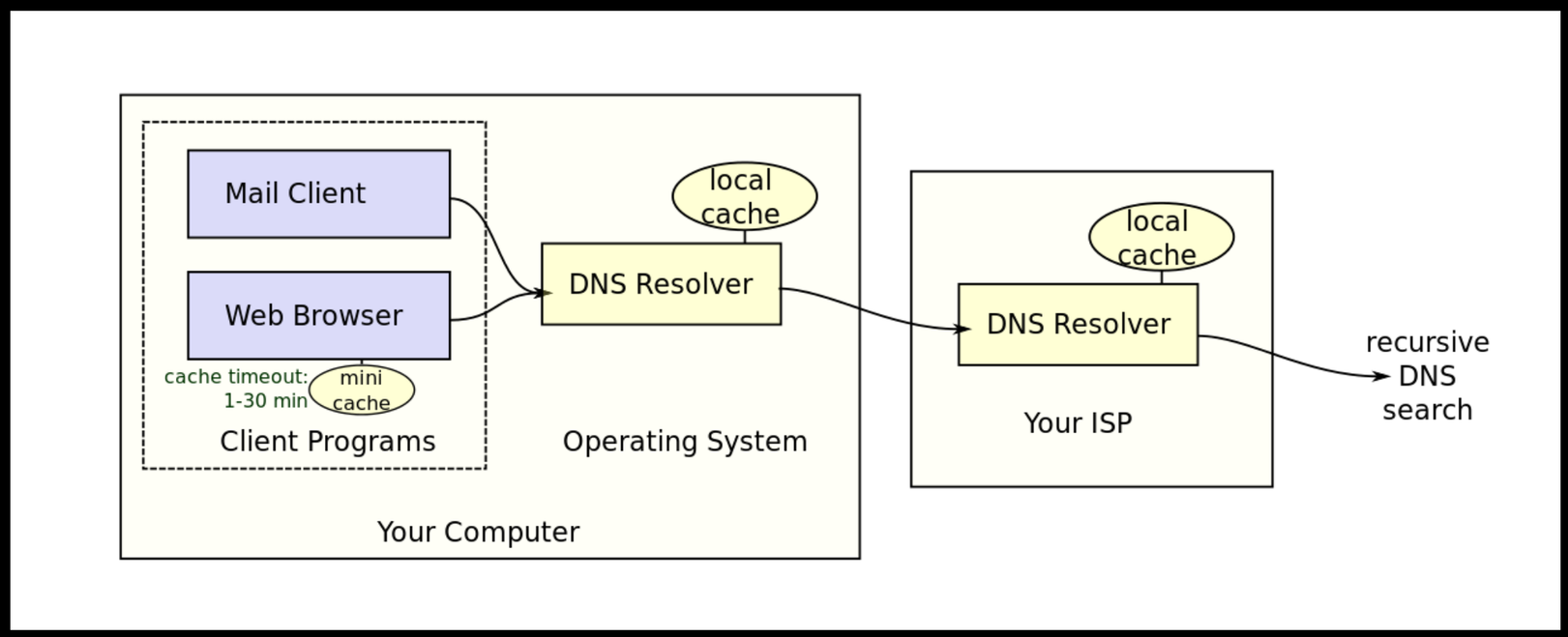
Users will normally start with a hostname, whether it is in their browser or in some application.

This is the first opportunity to control where they go.

Generally, your users will start with a DNS name or URL. So the first step is DNS lookup.

DNS - Caching considerations

One thing to be aware of -- most clients will go through a caching DNS server to reach yours.



One thing to keep in mind during this discussion is the effect of intermediate servers.

Typically, users will ask a server at their ISP or in their corporate network to look up any DNS request, which will eventually perform the DNS lookup on their behalf or return a cached value.

If all the users relying on that server are local to that server, then the results will be good.

If a large, nationwide ISP decides to run DNS out of centralized servers, all their customers will look like they are coming from the same physical location from a DNS perspective.

So you need to be aware that a DNS request may not come from the physical location of the user trying to connect.

DNS - GeoDNS

With GeoDNS, you get the source IP of the request, look it up in a geolocation database, and return appropriate responses.

EDNS support is critical for dealing with intermediate servers.

Pretty easy to implement (<1 kLOC) or use a provider that manages it for you (Amazon Route53).

A better option is to use GeoDNS.

This involves getting the user's IP address, running it through a geolocation database, and then deciding what IPs to send them for your hostname.

You can use the IP of the requesting server, but intermediate servers could throw you off.

However, if your users are going through a DNS server that supports EDNS (Google's public DNS and OpenDNS), you can get subnet information (generally a /24) about the requester passed all the way through.

Amazon's **Route53** DNS also supports a **latency-based routing*** policy if you run in AWS, because they track latency between their datacenters and various client networks.

DNS - Multiple records

Returning multiple addresses allows browsers and other apps to try different IPs if the first doesn't work.

Balance needs to be made between returning multiple IPs for high availability and targeting yours users precisely for low latency.

Browsers and other apps will often fail over to a different IP if more than one is returned in a DNS query, so if you want a truly reliable service, you'll want to return multiple IPs in different datacenters.

Of course, that limits your ability to send them to the closest site. It's a trade-off that needs to be considered.

IP Routing

Once your client has an IP address, you could to use Anycast to route traffic to a "close" datacenter

Hard to do right

No guarantee packets follow same path -- can break TCP

Only really worth considering for short-lived exchanges, good request/response, especially with UDP

Once DNS resolution is done, the user will have an IP address. At this point, you can use AnyCast as well to route requests for a given IP address to the closest datacenter, but this comes with the caveats about it being hard to do right PLUS you generally cannot guarantee that one user will send all packets to the same destination, so it is only really usable for short connections or when you can actually handle packets showing up in different datacenters.

Some content delivery networks use AnyCast since any one HTTP connection is likely to go to a single datacenter and be relatively short-lived. UDP request/response traffic (such as DNS lookups) are much more suited for AnyCast.

Application Layer

You might be able to redirect them at the application layer.

In HTTP, you could look up their source IP in a geolocaiton database, and redirect them to, say, us-east.myservice.com

If your clients connect via devices or apps that you can configure, consider setting the configuration there to go to the right place.

Probably the most reliable if you can do it

Finally, you can use application-specific tricks. When a user connects to your website, you can look at their source IP and redirect them to a server geographically close to them for the rest of their session. If the connections from outside are from devices you have control over, you can use a custom system to make sure they preferentially connect to close servers. This is likely to provide the best results if it applies in your case.

If you have the ability to configure clients ahead of time or redirect them on the fly, absolutely consider doing something at the application layer if latency is a prime concern.

Routing within the system

Routing within the system

Once a request enters our system, how do we decide where to route it?

So, now that some outside client has connected to a reasonable datacenter, how do you route traffic in and between datacenters?

When talking about this, I'll refer to two ways of communicating in a distributed system: push and pull.

Examples of push communication would be REST requests or RPC calls.

Examples of pull communication would be message queues or publish-subscribe systems.

Routing - Stay in one DC

Advantages:
- Simple to implement
- Minimizes latency (assuming no capacity problems)

Disadvantages:
- Doesn't handle partial datacenter outages
- Can't load balance across datacenters by service
- Can have increased latency if the data for that user "lives" in another DC

One option is once a request arrives in a datacenter, you keep it in that datacenter. This is a simple method, though it **limits your resiliency** and your **ability to load balance across multiple datacenters** if one gets overwhelmed.

This can be done by just keeping any service discovery, queues, and/or pub-sub systems local to a datacenter.

Aside from its simplicity, it avoids cross-datacenter connections and so can minimize latency of each request as long as each datacenter has sufficient capacity.

Routing - Route to a "home" DC

Advantages:
- Fairly simple to implement
- Works well if a user's data "lives" in one DC
- Better latency to only have one hop to "home" DC than to keep making requests there

Disadvantages:
- Doesn't handle partial datacenter outages
- Need to be able to find new "home" if the home DC fails
- Can only spread load by spreading out where data lives

Another option is to **give each user a “home” datacenter** and route their traffic there immediately, then keep it in that datacenter. Ideally, most of a user's traffic will enter at their home datacenter.

This approach is most useful when there is a lot of data associated with a user and keeping it in sync between multiple datacenters is difficult.

If you only need one round trip with the data (one get or set operation) you can keep the request in the original datacenter, but if you are going to be making multiple requests to a different datacenter the latency penalty can be pretty high. This also makes it easier to keep the view of the data that one user sees consistent, even in the face of some replication delays between datacenters.

Routing - Route to closest available service

Advantages:
- Provides greatest resilience to partial datacenter outages
- Can be enhanced to shunt load around heavily-loaded service instances

Disadvantages:
- Challenging to implement well
- Lots of knobs to tweak (do we include load? which DC do we try next?)
- Can increase per-request latency if the request bounces around

If you want to maximize availability in the face of instance failures and uneven load between datacenters, one option is to route each request to a random server in the nearest latency band.

That is, if there are local servers available, balance across those. Otherwise, if there are some in the next closest datacenter, load balance across those.

To do this you need to have a datacenter-aware service discovery system, so that can be taken into account, or the clients need a preferential list of service discovery systems to use, starting with one in the local datacenter.

An additional enhancement can be to request from further servers when “close” servers are overloaded by some metric.

Choosing the “best” destination can involve lots of factors and tradeoffs, so keep in mind that sometimes imperfect simple algorithms are better than overly complicated ones.

Routing - Route to closest available service

When doing this with pull-based systems (queues or pub-sub), you can:
- Make the decision on the producer side ("My normal queue is overwhelmed, so I'm putting this message in a different datacenter")
- Make the decision on the consumer side ("My normal queue is empty and a remote one seems to be overloaded, so I'll grab a message from there")
- Some unholy combination of both

The pull equivalent of this would be where either producers write to remote queues when the local ones are overloaded or consumers read from remote queues if they have spare capacity and there are remote queues backing up.

In this case, you need to be careful to balance a desire to spread the workload evenly with trying to limit unnecessary hops between datacenters. Usually you wouldn’t want to pull from a remote queue unless it was truly falling behind and had a significant backlog, otherwise you risk just adding latency with no real benefit.

Routing - Route to least loaded service

Advantages:
- Spreads load over all resources evenly

Disadvantages:
- Latency can be much worse than staying in one location
- Extra cross-datacenter bandwidth

You could also route to the least loaded service. But, if you make a lot of calls to different datacenters, you can increase the latency dramatically. This would only make sense for batch operations where latency doesn't matter and operational efficiency does.

Service discovery

Since I touched upon this, it makes sense for me to talk about service discovery. There are a ton of ways to implement service discovery in a system, including writing your own system from scratch. But I wanted to cover a quick survey of some options since it is so intimately tied with the idea of internal routing.

Service discovery

Two parts:
- Service registration
- Service discovery

There are two parts to service discovery that are impossible to separate -- service registration and discovery itself.

Service registration

Registration is how services get into your service discovery system in the first place.

Registration is how services get into your service discovery system in the first place.

Service registration

Manual, static list

Example:

```
export SERVICE_ADDRESSES="10.1.1.1:2012,10.1.1.5:2079"
```


The simplest approach, but by far the least flexible, is if you manually keep a list of what services run where. This might work when you are starting out, but it is easy to outgrow.

You can’t dynamically add instances of services, nor can you remove them, which means the client is going to have to do some work if you want to handle service failure.

A slight improvement is if you are using a configuration management system like Chef, Puppet, Ansible, or SaltStack, you can use those to generate your static lists or they can use the same static list to actually deploy those services.

Service registration

Generic key-value datastore with TTL/expiration

Example: Zookeeper, etcd, Redis

Another option is to use a generic key-value datastore and have services register themselves into it when they start.

If you use this approach, you may want to have some form of **heartbeating** by having services **keep re-registering themselves with an expiration or TTL** on the record so you can detect if a service goes away.

Something like Zookeeper, etcd, or Redis can work well for a case like this, and all can effectively push changes or change notifications to client applications if they want to subscribe.

Zookeeper has the concept of ephemeral nodes that disappear when a client disconnects, so this can be an alternative to re-publishing an expiring value.

Service registration

Purpose-built service discovery system

Example: Consul

You can also use a tool purpose built for service discovery like Hashicorp's Consul.

Consul has support for built-in **healthchecks, run locally** on each node for scalability, and a **gossip protocol for detecting nodes that are completely down**, which scales well as the number of nodes increase. It also natively supports the concept of **multiple datacenters**, which obviously appeals to me, though it leaves the decisions on routing up to the user.

Consul makes its results available **via HTTP or DNS**, so it is relatively easy to use from any language. If you are considering building a service discovery system on a generic datastore, Consul is worth a serious look.

Service registration

Service orchestration (i.e., you already know where they are)

Examples: Kubernetes, Docker Swarm

Another option is when you have a system orchestrating your services. This could be Kubernetes, Docker Swarm or more traditional orchestration tools.

When you have a system that orchestrates your services, it knows what services are created where, and as such the system can act as a source of truth for where services are running.

Kubernetes provides explicit support for this with it’s Service abstraction, and Docker Swarm provides both Service hostnames using DNS within the cluster.

Of course, you can build a system yourself, but it will likely follow the patterns here or be built with the components already mentioned.

Service discovery

Manual, static list

Example:

```
export SERVICE_ADDRESSES="10.1.1.1:2012,10.1.1.5:2079"
```

Probably what you are doing if your service registry is also a static list

Could be generated from a more advanced registry

The simplest approach is to use some kind of static list, perhaps as configuration for your code or provided as environment variables on startup.

If you use this approach, you have a limited ability to add new instances of services and any support for services becoming unavailable needs to be implemented in each client.

Service discovery

DNS (multiple A/AAAA records)

Example response:

```
> dig a servicex.example.com +short
10.20.1.5
10.20.1.13
```


Next, you can use DNS. The simplest approach uses multiple A records. If you use DNS in a service, you want to make sure you control the DNS lookup process so that the program doesn’t look up the name once then cache the results forever. A big advantage here is that you can use this with any arbitrary program and it will at least get to a valid server (though it may cache the lookup).

Service discovery

DNS (SRV records)

Example SRV response:

```
> dig srv _http._tcp.servicex.example.com +short
1 10 8080 node2.us-east.example.com.
1 10 8080 node4.us-east.example.com.
2 10 8080 node4.us-central.example.com.
3 10 8080 node8.us-west.example.com.
```

```
[priority] [weight] [port] [host]
```


You can also use DNS SRV records.

An advantage of SRV records is that they support a port number (great if you are dynamically allocating ports for services) and both priority and weight, such that a centralized system can adjust priorities and weights to control routing. For example, when looking up service X, the DNS server in the “US East” datacenter could have the lowest priority for local servers, followed by a higher priority value for servers in “US Central”, and the highest priority value for servers in “US West”. Within a datacenter, the weights could be adjusted based on load, response latency, or error rates.

As long as you control the servers (and therefore the caching policies) DNS isn’t a bad approach, since it was inherently designed for service discovery (finding the hosts that service a DNS name).

Service discovery

Load balancer / reverse proxy

Examples: HAProxy, Nginx, Traefik

Allow clients to stay dumb by putting intelligence in the proxy (or what configures the proxy)

Possible single point of failure if the proxy fails

Another approach is you can use a load balancer or reverse proxy. Something like HAproxy or Nginx work well here.

Traefik can subscribe to a variety of backends (Kubernetes, etcd, Consul) and update on the fly, so you don't need any glue code between your registration system and the load balancer.

Many of these tools are limited to HTTP traffic (HAproxy being a notable exception), so they work best with REST interfaces or other protocols that run on HTTP.

This allows intelligent service routing with fairly simple clients.

Service discovery

Local proxy

Examples: Linkerd, Envoy

Like a load balancer / proxy, but runs on each node (connect via localhost)

Supports distributed tracing and per-host metrics

Both have some concept of datacenter-aware routing ("zones")

A special case of load balancers would be a local proxy. In this case, you effectively run a load balancer on every host, and clients connect to the local load balancer. Linkerd and Envoy are good examples of this approach, and they can provide additional benefits on top of a centralized load balancer including limited blast radius if an instance fails, distributed tracing of calls, and load balancing based on datacenters ("zones") or the latency from the client machine (at least for Linkerd).

Service discovery

Thick client

Examples: Netflix Ribbon, Twitter Finagle

In this case, the client does all the work to decide which instances to route to and deal with slow or unhealthy instances

Latency and connectivity checks are more accurate than centralized systems

Enhances client retry logic

The last approach I'll mention is having a simple registry of services, but then using a "thick" client to access them. An example here would be Netflix's Ribbon or Twitter's Finagle.

These clients have some kind of internal load balancing functionality. Looking at Ribbon, one of the load balancing policies it can use is response-time weighted, such that it sends more requests to services that respond faster, which can be good for load balancing or a crude way of preferring local services.

Ribbon also has functionality called "zone affinity", which filters out the servers that are not in the same zone as the client, unless there are no servers available in the client zone

Finagle is used under the hood in Linkerd

Data management

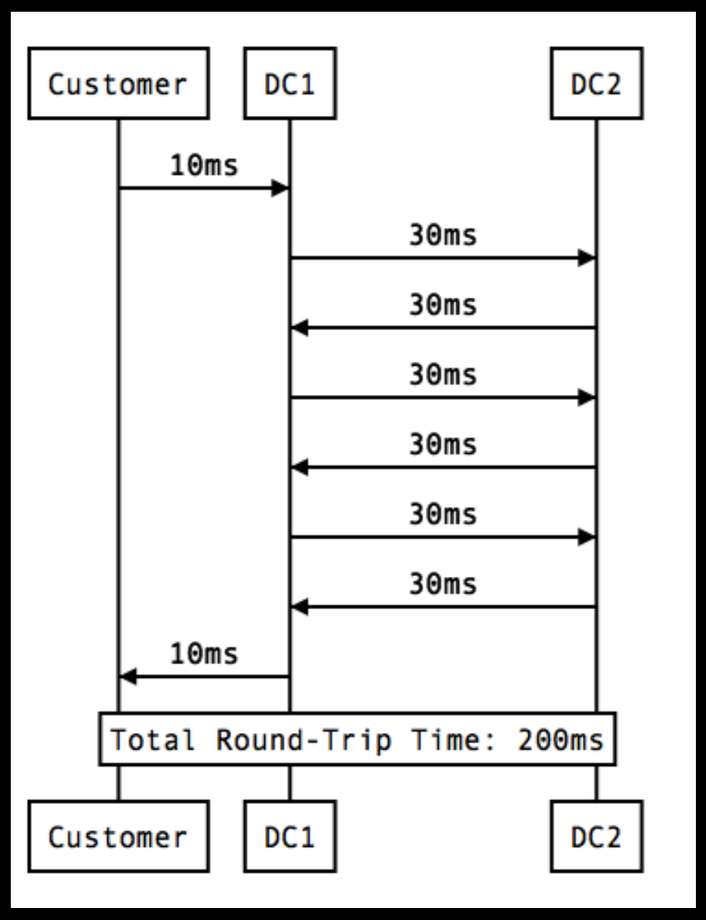
Data management is another concern. In a way, data is at the core of the most significant architectural decisions one needs to make when physically distributing parts of an application. It can dominate the decisions about how to route data if different datacenters have different access to the data.

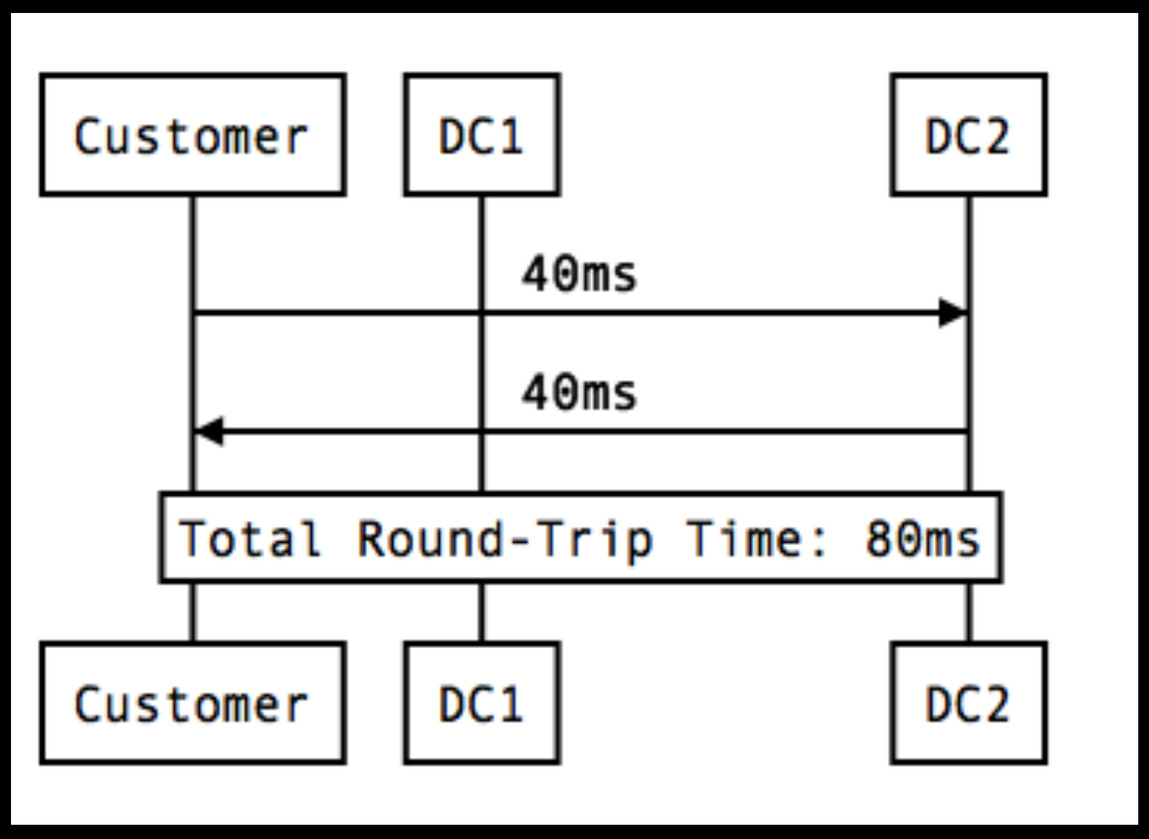
Data management

	Changes infrequently	**Changes frequently**
Small data	Easy	OK
Large data	OK	**Danger Zone!**

I find it helpful to think of data as if it has mass and inertia. A small amount of data can be updated frequently and replicated everywhere. A large amount of data can be replicated to all datacenters in a timely manner if it doesn’t change much. For example, a database of user information -- you may have a lot of users, but it probably grows at a reasonable rate and updates are relatively infrequent. If you have a ton of data, changing all the time, fully replicating it may not be practical.

Data management





On the other hand, latency constraints may require some data replication. If you need to make more than one request for data in a remote datacenter, it may have been faster to only have one datacenter.

Here is an example with two datacenters where a request going to the first datacenter would need to make three calls into the second datacenter. If the request had gone straight to the second datacenter, the round-trip time would only have to be paid once instead of partially three times.

Data planning

Start by looking at your data and segment it based on its characteristics and replication requirements.

For planning purposes, I think it is useful to segment your data based on its characteristics and replication requirements. For each segment of data, you want to look at these qualities:

Size

Total size of your data set

Affects storage requirements and initial replication process

Size -- The size of your total data set. This affects your initial replication process, and your storage requirements. Remember that in some failure scenarios, you might need to repeat your initial replication, so don’t think of it as a one-time task -- it may need to happen again.

Rate of change, and size of changes

How often does your data change, and how big are those changes?

Determines the necessary bandwidth

Rate of change, and size of those changes -- You need to plan how much bandwidth it will require to replicate changes, or if it is even possible. Make sure you have significant headroom -- I'd suggest at least 2 times your expected needs -- in case you fall behind or the data rate has a transient increase.

You need to have the spare bandwidth to catch back up or you risk falling further and further behind.

Latency sensitivity

How stale can your data be?

"Is this something I would consider caching?"

Latency sensitivity -- How stale can your data be? For some data, minutes are ok. A user profile picture, for example -- you can cache those for a few minutes, especially if you invalidate the cache that user is accessing immediately.

Most data can change in terms of seconds (call routing in a phone system and other user settings).

If you have frequent sensor data of some kind, you may need low latency depending on how it comes in. If you know a sensor opened, that’s great, but if all you get is that the sensor IS open and have to determine that it was closed before, or that the sensor toggled and you only know it’s current state if you knew the previous state, you need current data.

How often the data is needed

Frequent reads mean you want it close to where it will be needed

Infrequent reads may mean the latency hit to go to another datacenter may not matter

How often the data is needed -- If data is needed frequently, you want it close to where requests might be processed -- either by replicating it or making sure requests are routed to the data. If it is looked up once a day, a trip to another datacenter probably isn’t an issue.

Read / write ratio

Data that is frequently read but rarely written is a good candidate for a single write master with replicas or caches in other datacenters.

Data that is frequently written may indicate having a "home" datacenter for a user is a good idea.

Data that is frequently written and can be stale may be a candidate for queuing writes and batching them to the datastore.

Read / write ratio -- Data that is frequently read but rarely written is a great candidate for strategies where there is a single write master at any given time that sends copies of data to read-only replicas.

If writes are common, you may want to have a “home” location for each user that houses the write master for their data, batch writes if possible, or if you can sacrifice either consistency or availability, use a datastore that supports writes from multiple locations.

Consistency requirements

Do two writes to the same data need to be seen in order?

Is it ok if two reads at the same time can get different data for some time?

Consistency requirements -- Do two writes to the same data need to be seen in order?

If a user changes a setting from two computers simultaneously through two different datacenters, they probably won’t be surprised if either change takes effect.

If you are tracking sensor data from a door, it may matter if it opened before going back to closed, and if tracking two doors it may matter which one opened first if you are trying to determine if someone is leaving or entering a building.

Wrap up

We want reliable systems...

...systems more reliable than any one provider or datacenter

Wrap up

We want to be fast.

Wrap up

Figure out your requirements for your data and user interaction

Wrap up

Plan how to get users to the right datacenter

Wrap up

Decide how to route requests one inside your system...

...and how that works with service discovery

Wrap up

Make sure you have a plan to handle your data:

- replication
- caching
- consistency requirements
- ...

Wrap up

Make something awesome...

Wrap up

Make something awesome...

...then tell everyone how you did it so we can all make more awesome things

References

- Envoy - https://lyft.github.io/envoy
- Linkerd - https://linkerd.io/
- Twitter Finagle - https://twitter.github.io/finagle/
- Netflix Ribbon - https://github.com/Netflix/ribbon
- MongoDB - https://www.mongodb.com/
- Cassandra - http://cassandra.apache.org/
- Project Voldemort - http://www.project-voldemort.com/

Attributions

- US map background from Wikimedia Commons by user Theshibboleth (CC BY-SA 3.0)
- Packet visualization used from here with permission from Carlos Bueno
- Explosion image from Pixabay (CC0 Public Domain)
- DNS image from Wikipedia (Public Domain)
- Superhero image from Wikimedia Commons by user FRacco (CC BY-SA 4.0)