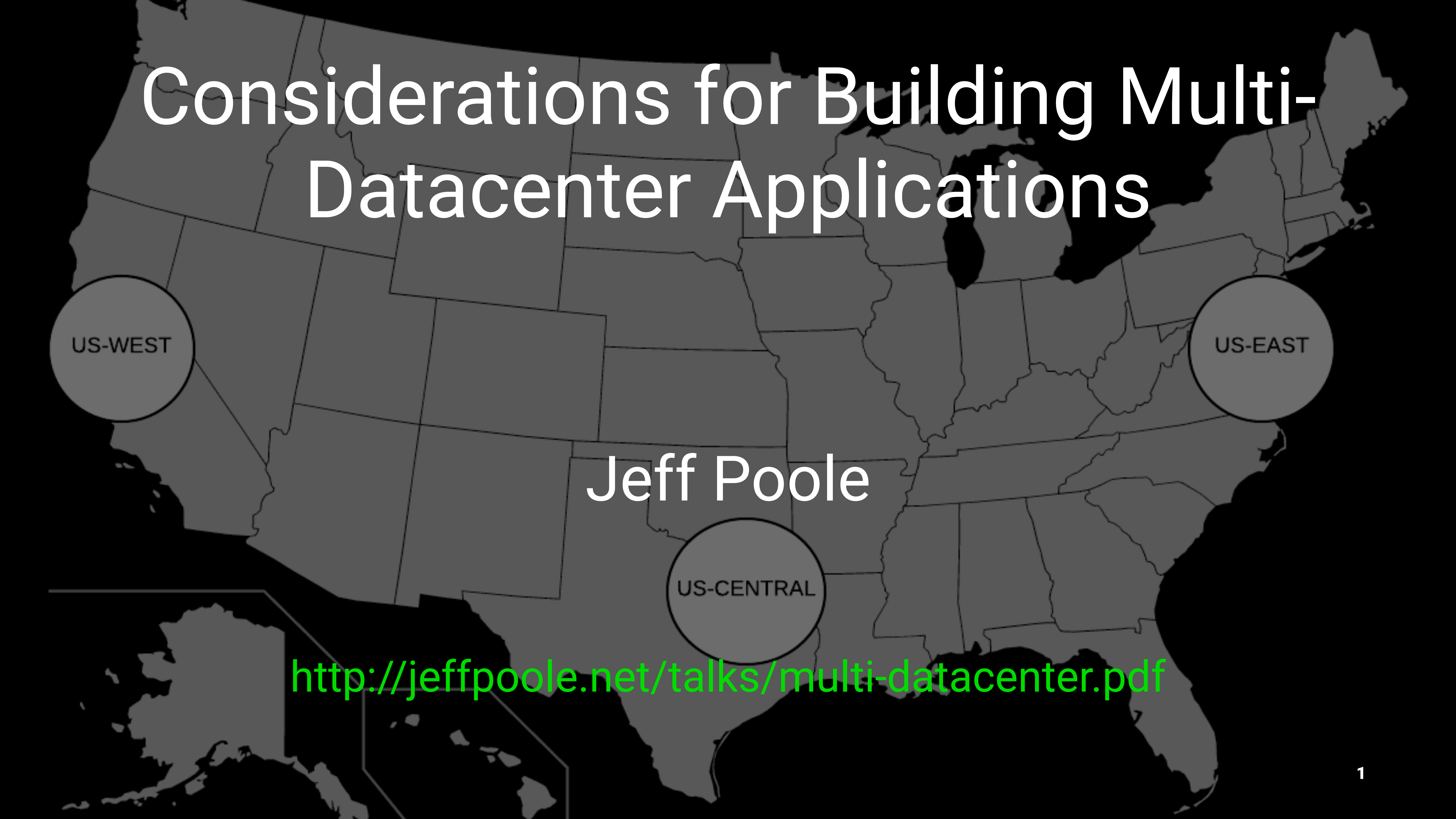


Considerations for Building Multi-Datacenter Applications

Jeff Poole

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

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

Jeff Poole

~~Principal Software Engineer~~ DevOps Manager

vivint.SmartHome™

@_JeffPoole / jeff@jeffpoole.net

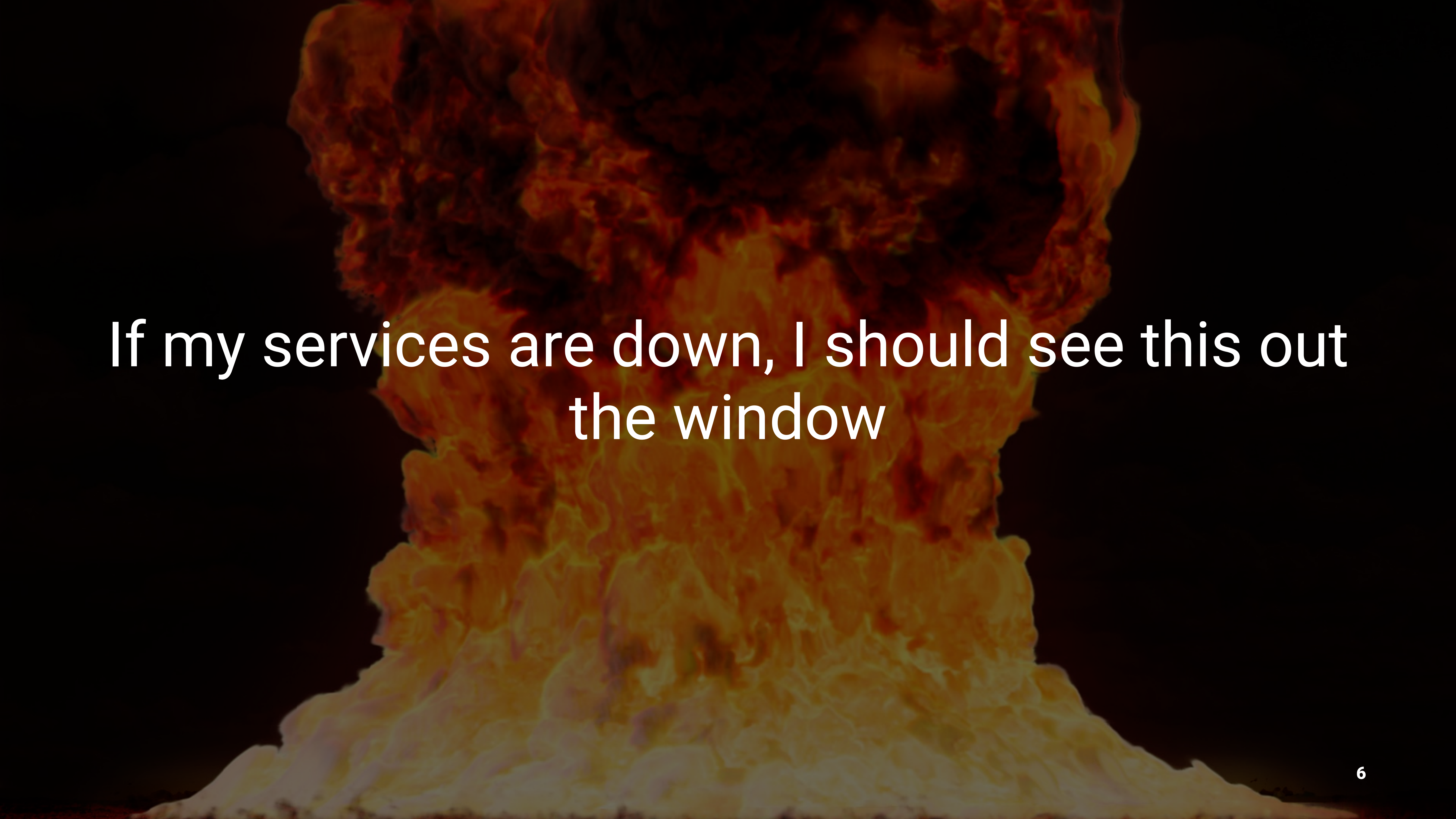
What will I cover?

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

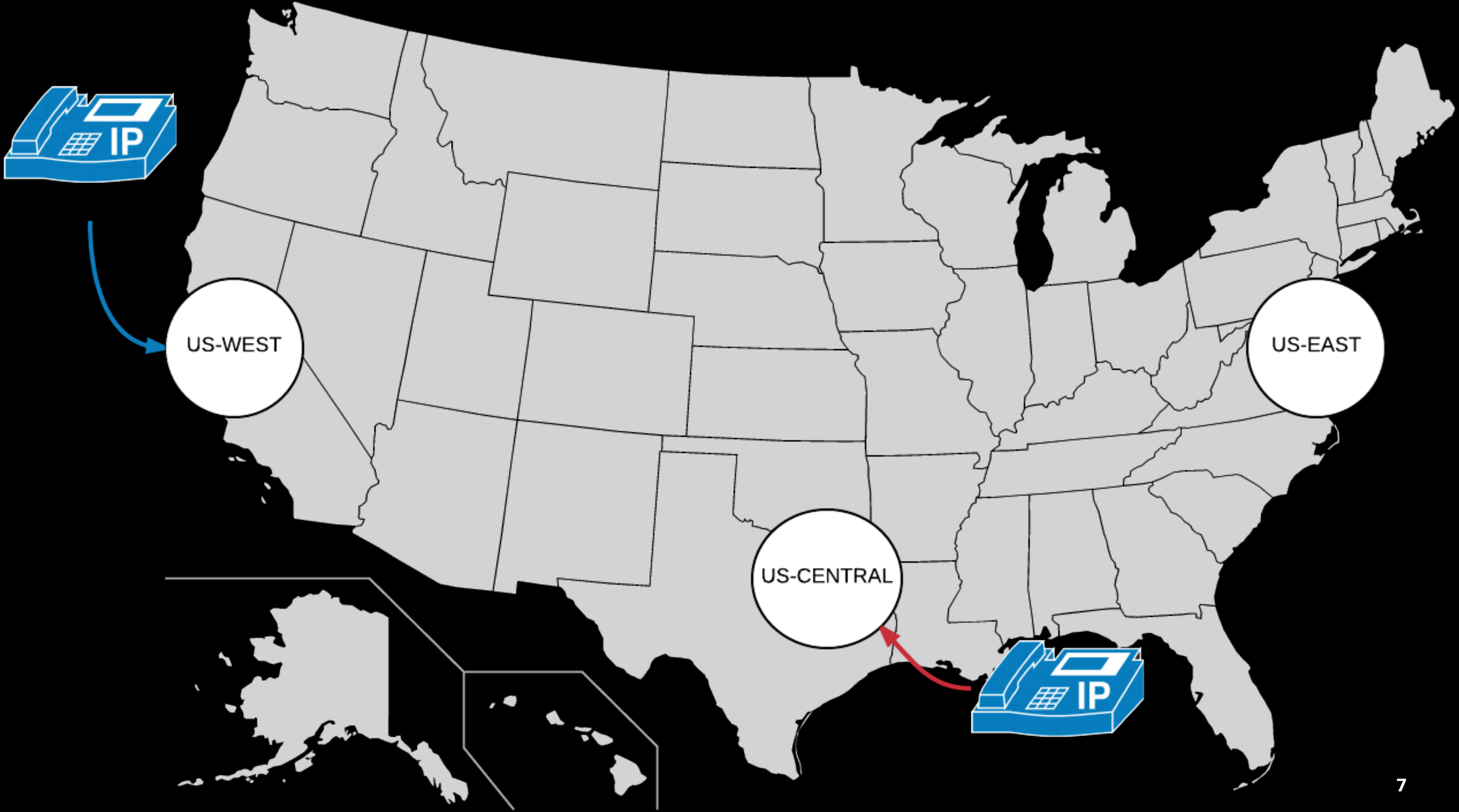
Why go through the pain?

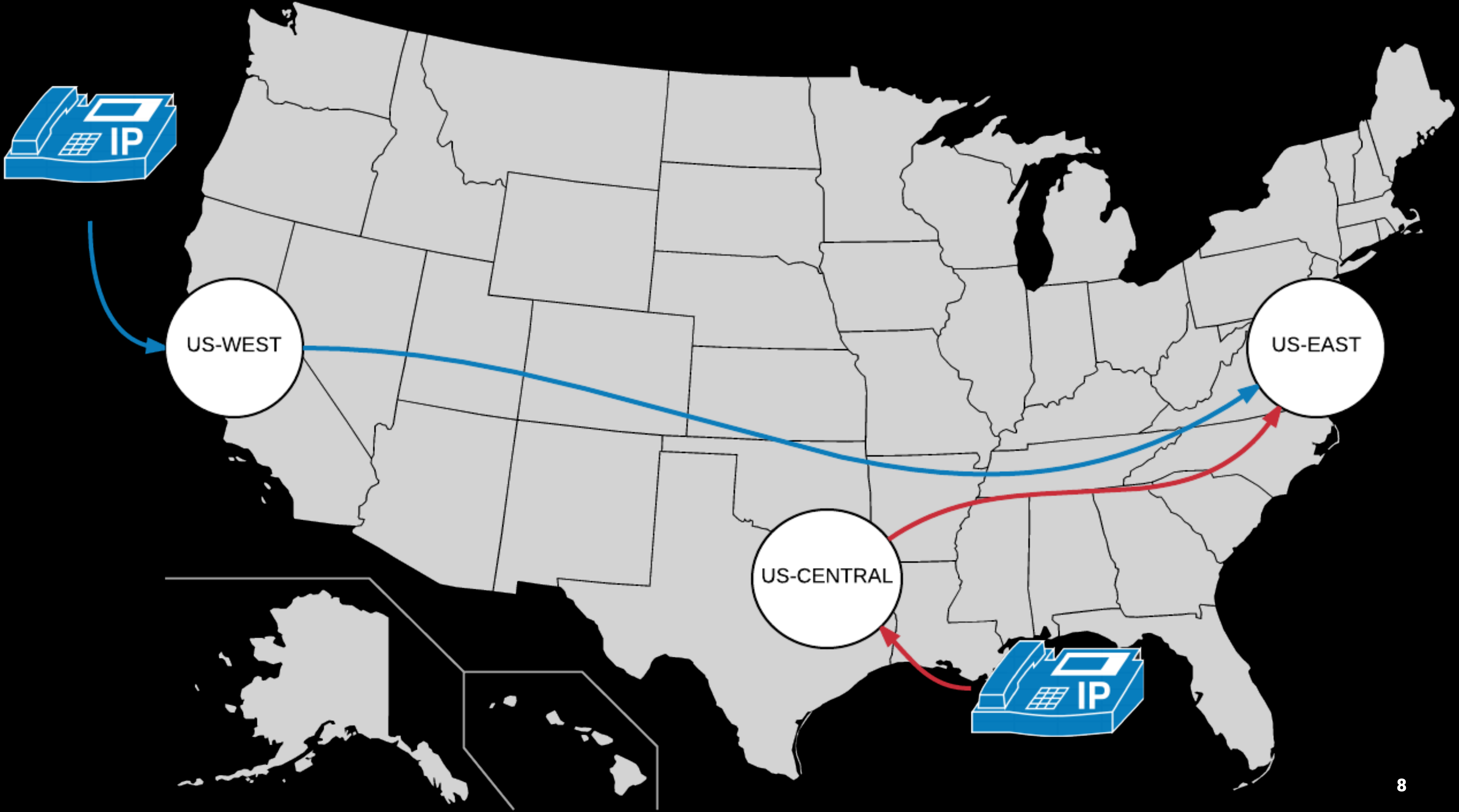
Reason: Resilience

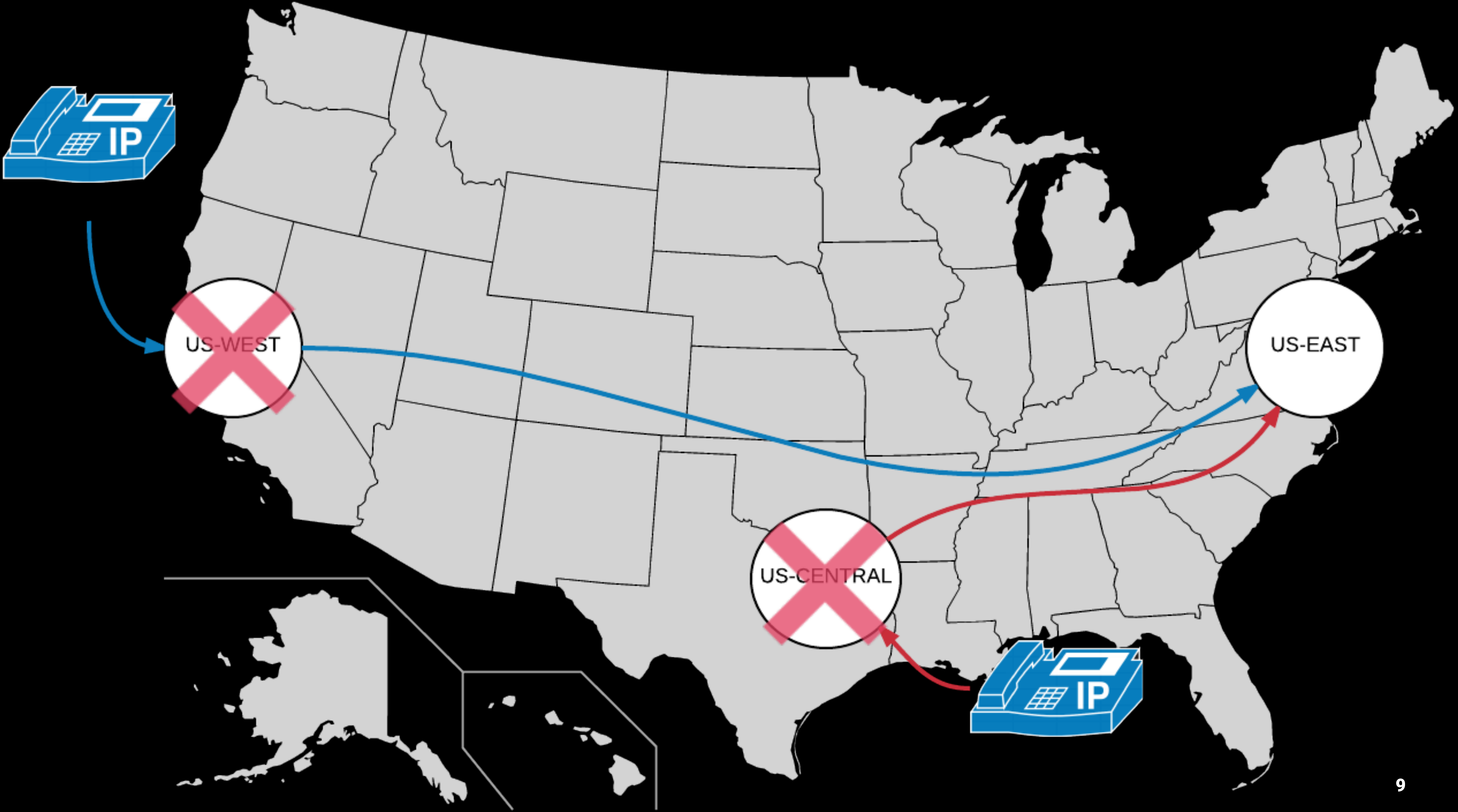
- 24/7 uptime expectations
- No cloud provider or datacenter has 100% uptime
- "It's not *my* fault!" doesn't cut it



If my services are down, I should see this out the window


IP
US-WEST
US-EAST
US-CENTRAL
IP


US-WEST
US-EAST
US-CENTRAL
IP
IP


IP
US-WEST
US-EAST
US-CENTRAL
IP

Why not use a "warm" backup site?

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

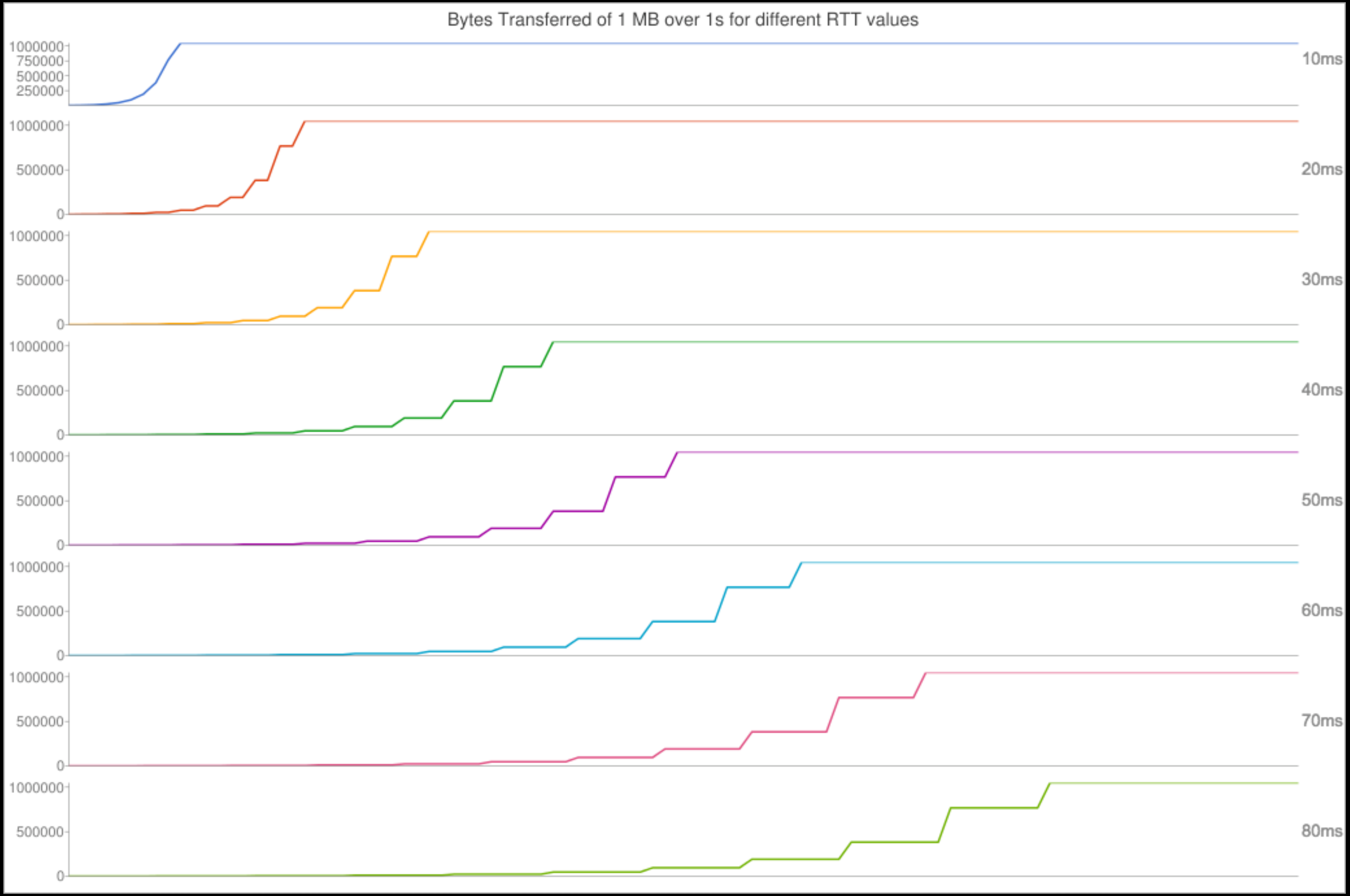
Think Schrödinger’s Backup:

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

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

client
server


Bytes Transferred of 1 MB over 1s for different RTT values
1000000
750000
500000
250000
0
10ms
20ms
30ms
40ms
50ms
60ms
70ms
80ms

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

Planning

Planning

Latency constraints

How quickly do you need to service user requests?

Are there asynchronous requests with different requirements?

Planning

Change propagation

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

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...

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

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?

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?

Routing from outside the system

Routing from outside the system

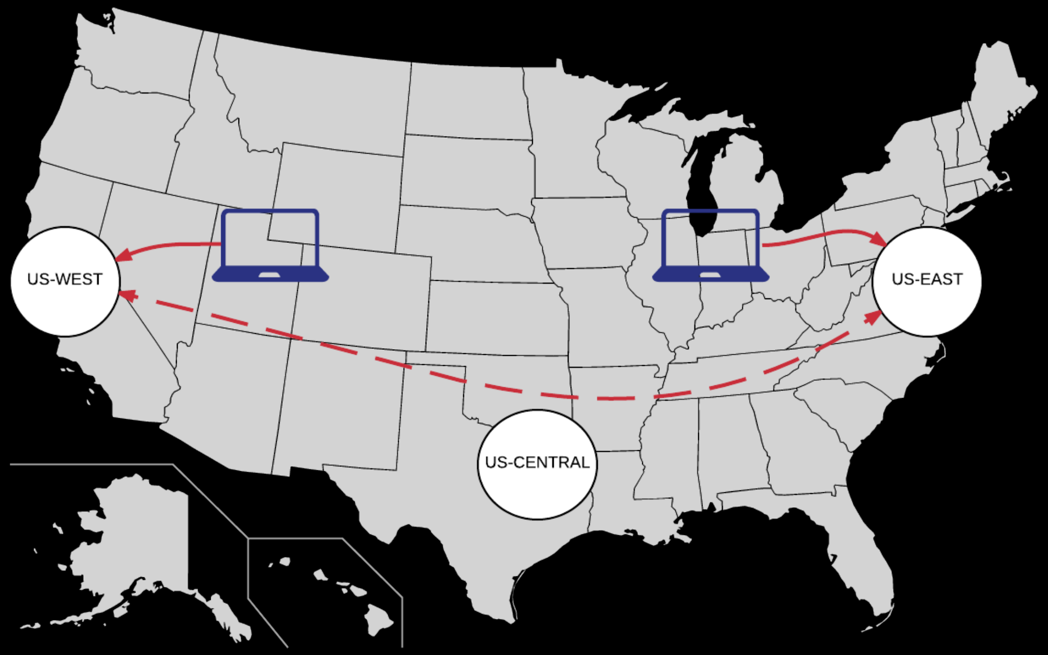
When you have multiple datacenters, what requests do you route where?

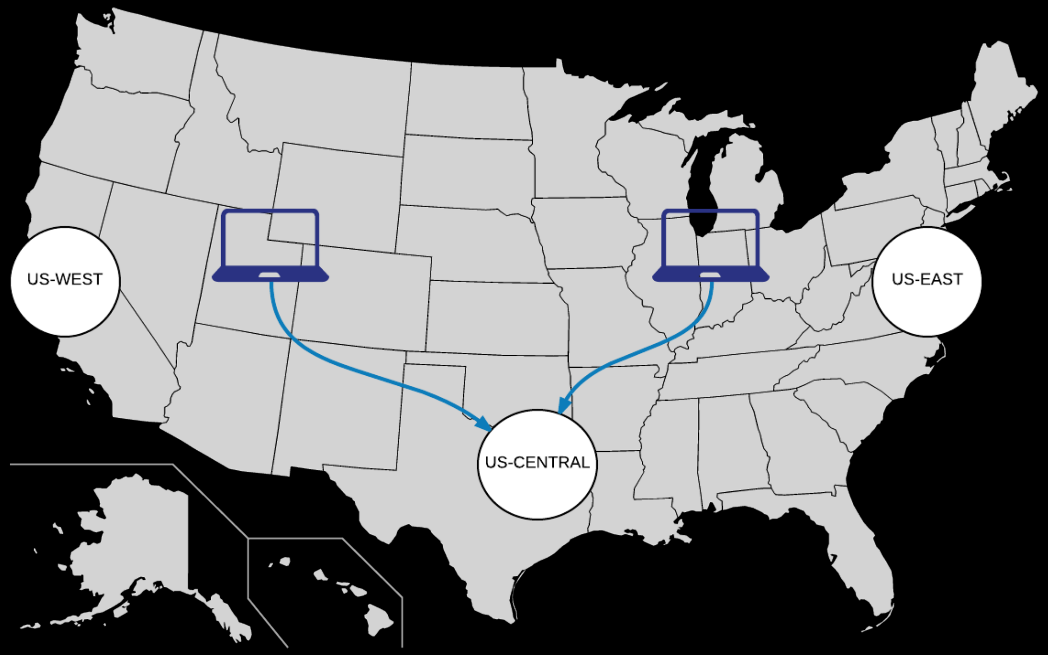
- User interaction
- Fixed hardware, or users with a known, fixed location

Routing from outside the system

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

- Matching two peers for real-time communication





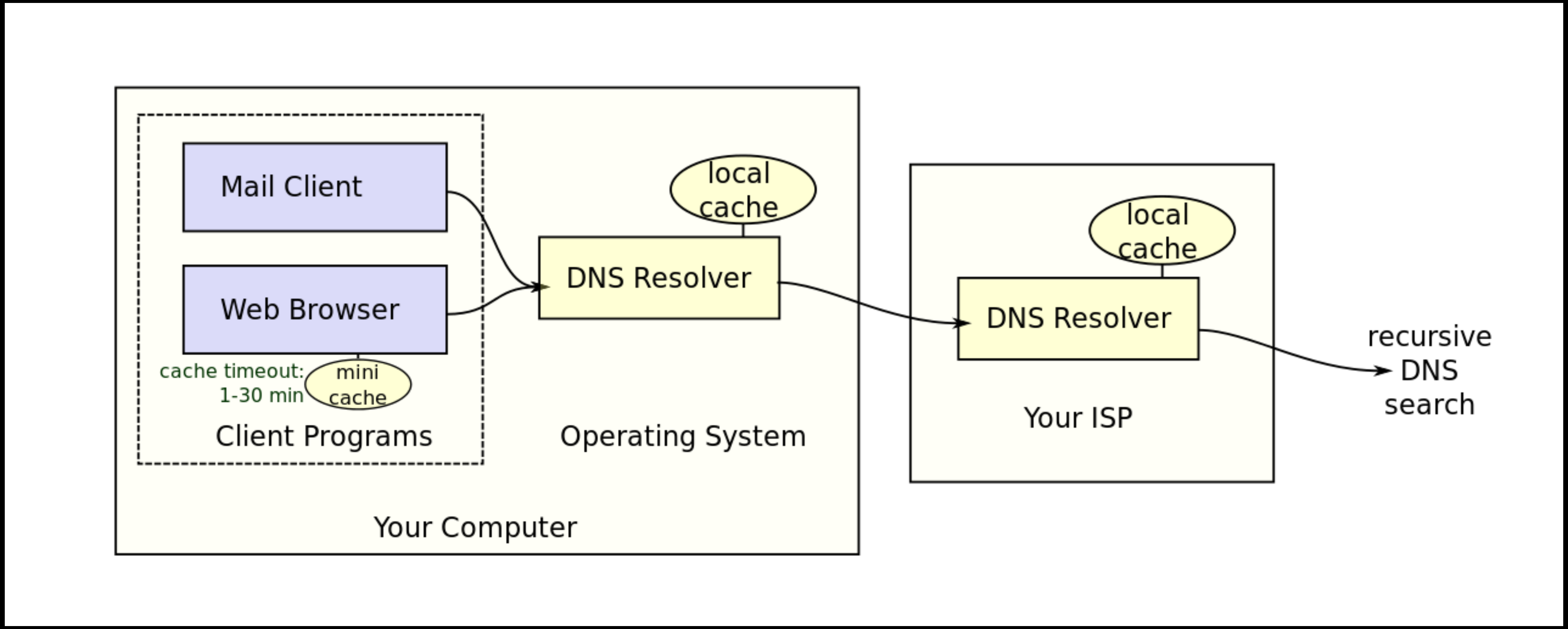
DNS

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.

DNS - Caching considerations

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



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).

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.

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

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

Routing within the system

Routing within the system

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

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

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

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

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

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

Service discovery

Service discovery

Two parts:
- Service registration
- Service discovery

Service registration

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"
```

Service registration

Generic key-value datastore with TTL/expiration

Example: Zookeeper, etcd, Redis

Service registration

Purpose-built service discovery system

Example: Consul

Service registration

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

Examples: Kubernetes, Docker Swarm

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

Service discovery

DNS (multiple A/AAAA records)

Example response:

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

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]
```

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

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")

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

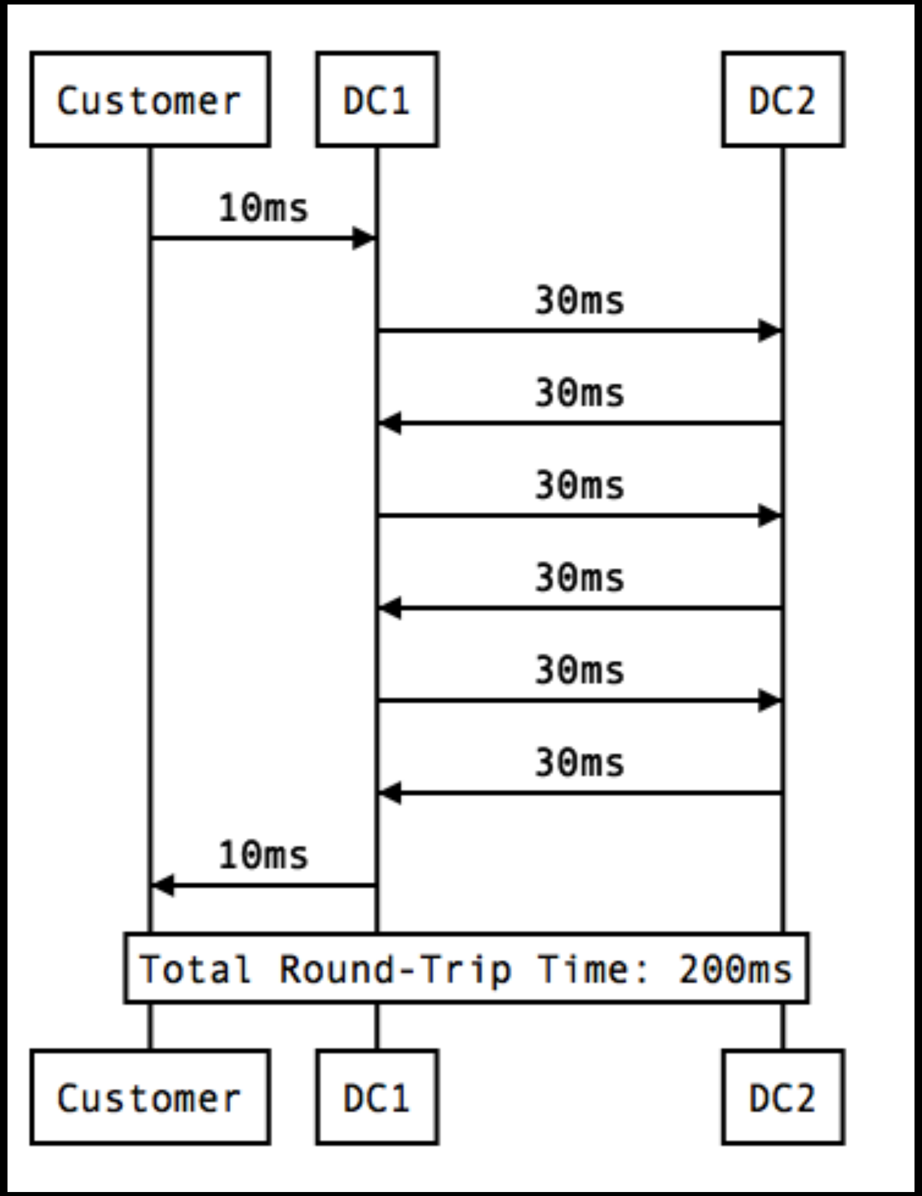
Enhances client retry logic

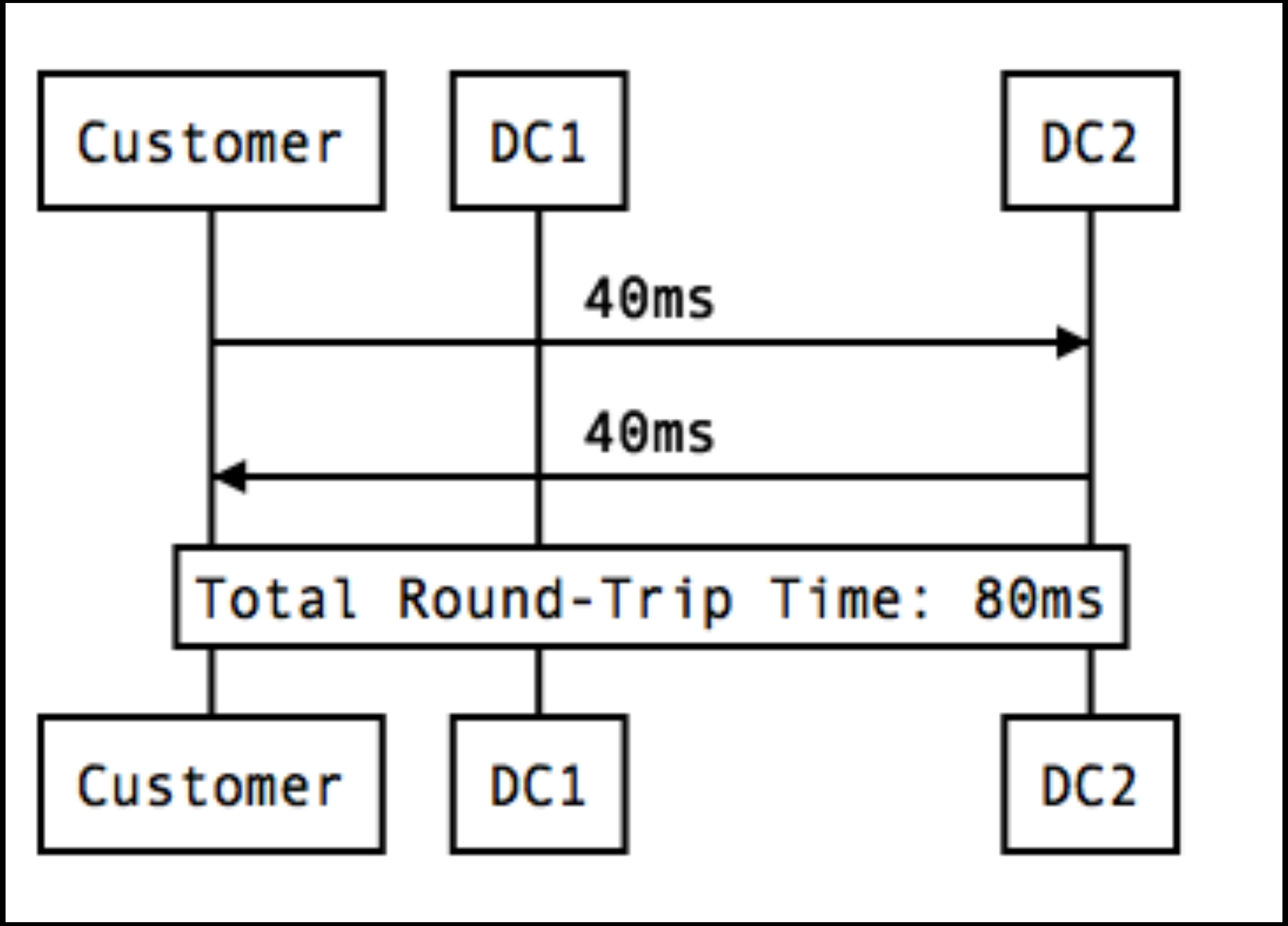
Data management

Data management

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

Data management





Data planning

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

Size

Total size of your data set

Affects storage requirements and initial replication process

Rate of change, and size of changes

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

Determines the necessary bandwidth

Latency sensitivity

How stale can your data be?

"Is this something I would consider caching?"

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

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.

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?

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)