CS130: Software Engineering

Lecture 13

Web Server &
Distributed System Architecture
Anti-patterns

https://forms.gle/zbBhkVhdA5xZNEHDA

- A tweet: What's the best functionality to add to a webserver?... Wrong answers only.
- A tweet: Have you had merge problems? How did you solve them?

Building Supercomputers: Then and Now

Web servers in 2000

Netra ft 1800

- Redundant hardware
  - Tolerates hardware failure
- Built-in monitoring
  - Detects hardware failures
- Hot-swappable
  - All components can be swapped without shutdown
  - Including motherboard (!)

Drawbacks of big iron

- Hardware was very reliable, but very expensive
  - See also: World's Most Expensive Hard Drive Teardown (2GB = $250k in 1989)
- Physically constrained
  - The amount of power dissipated in a fixed volume is limited

Drawbacks of big iron

- And if you go bigger, they turn into mini distributed systems
  - See also: Cray Internal Network Topology
  - This is also why everything was “done” in the 1970s

Fast Forward to Today

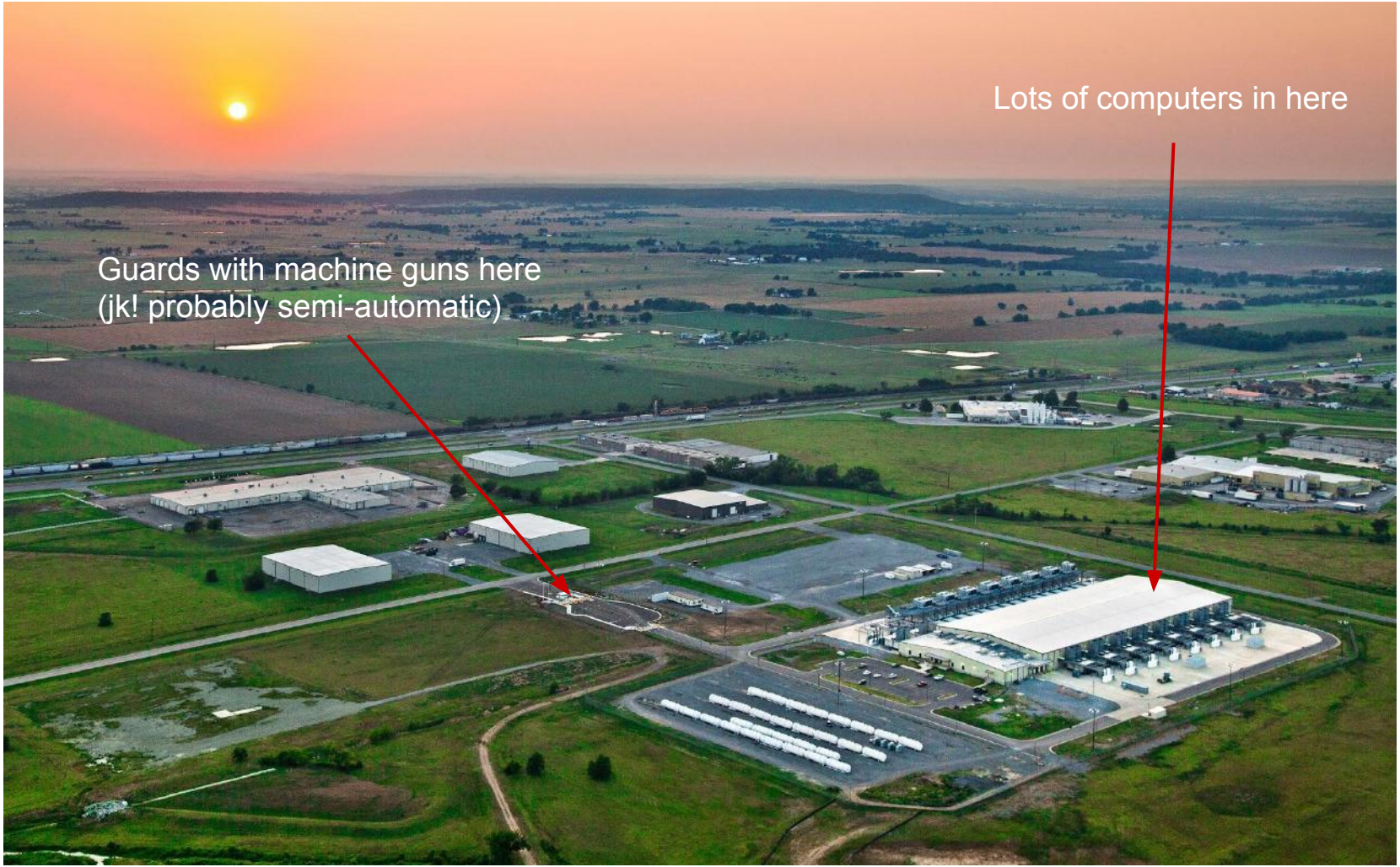
Distributed systems have largely eclipsed big iron:

- Networks are faster
- Commodity hardware is much cheaper and less crappy

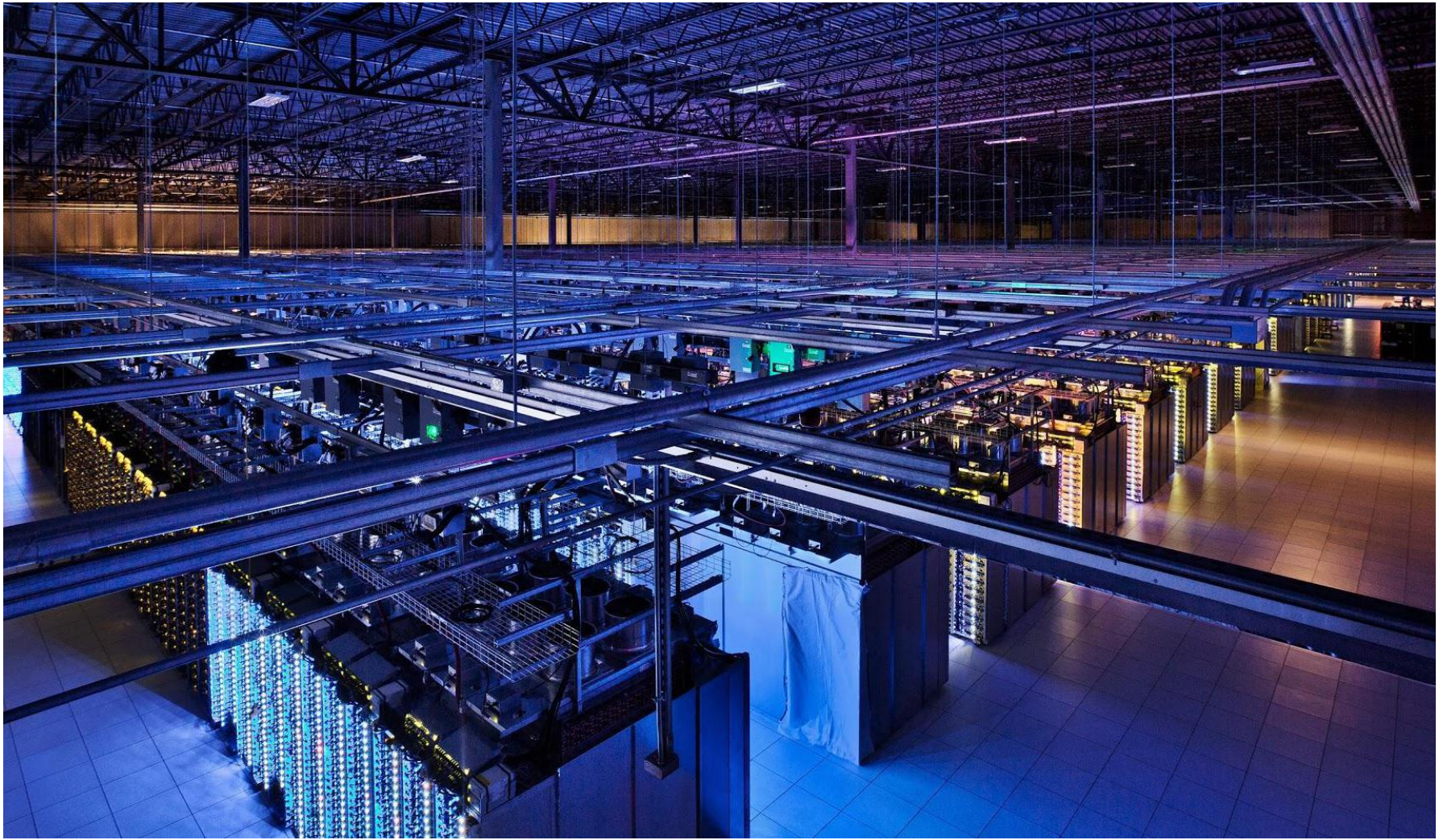
We can apply many of the concepts from mainframe computing and supercomputing to distributed computing

In some sense, we are building a warehouse sized supercomputer

Google Datacenter in Georgia



Supercomputer as Warehouse (Google Oklahoma)



Inside the warehouse (Google Iowa)

Cooling all those computers (Google Georgia)

Datacenter Efficiency

Power use is up (absolute scale), but increased workload is more efficiently executed

BigTech has commitments to go carbon neutral

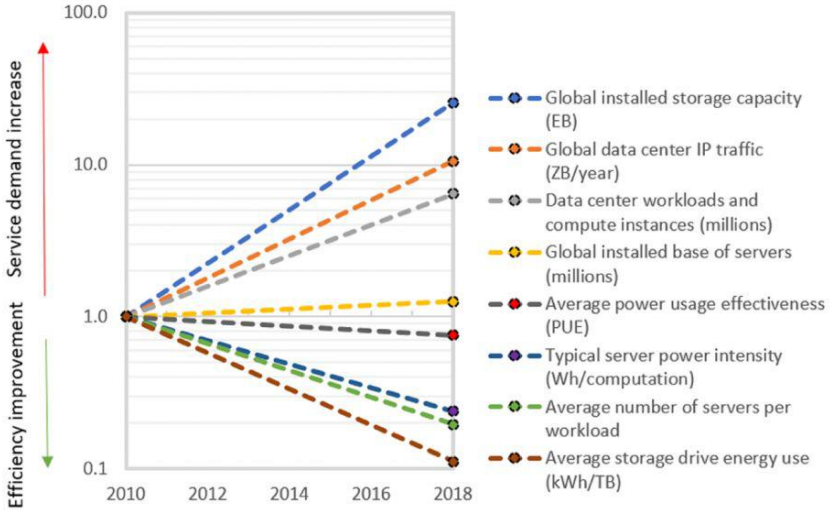
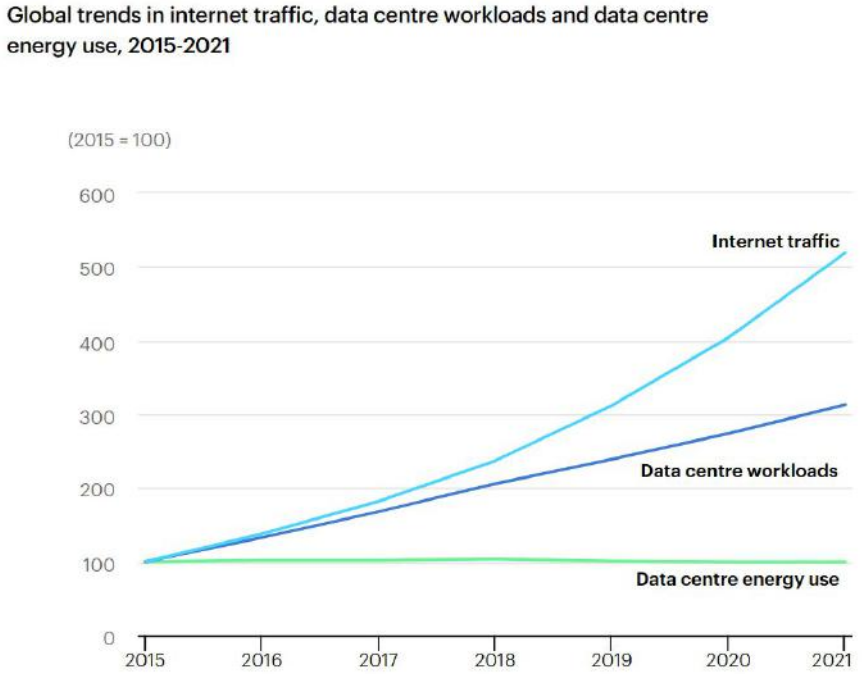
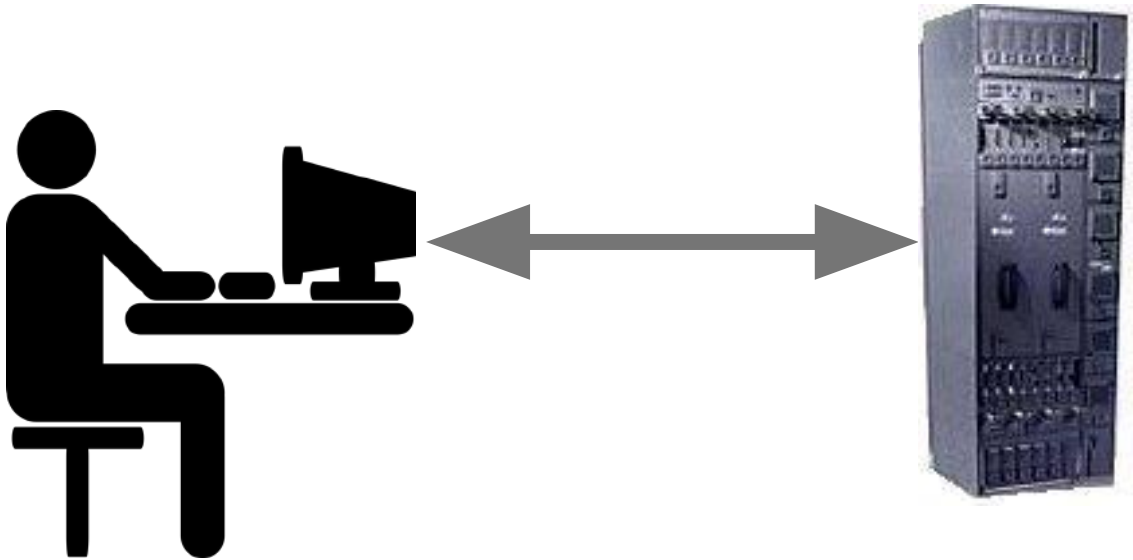


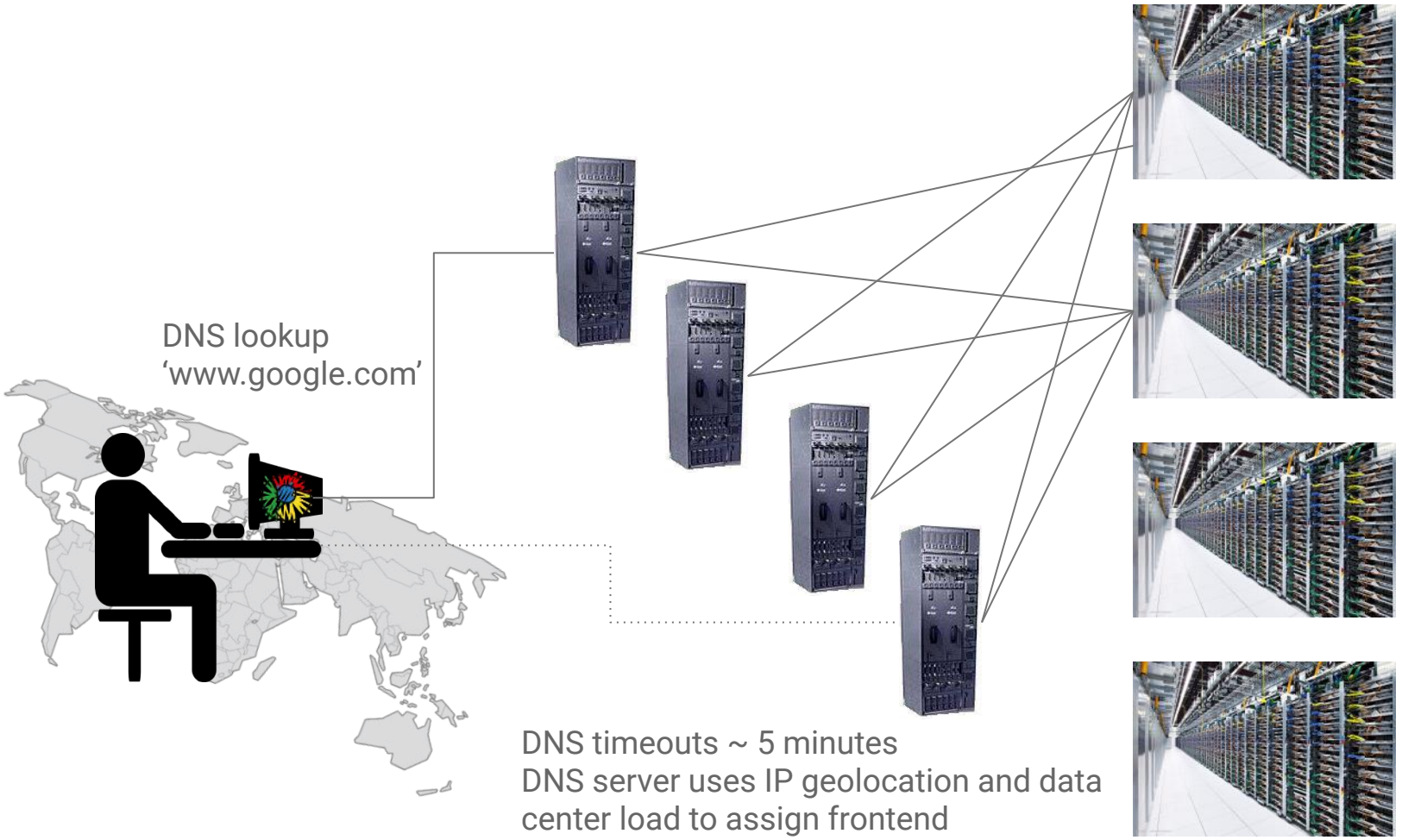
Figure 3. Relative change in global data center energy use drivers (2010=1). Source: Masanet et al. 2020.



Distributed applications

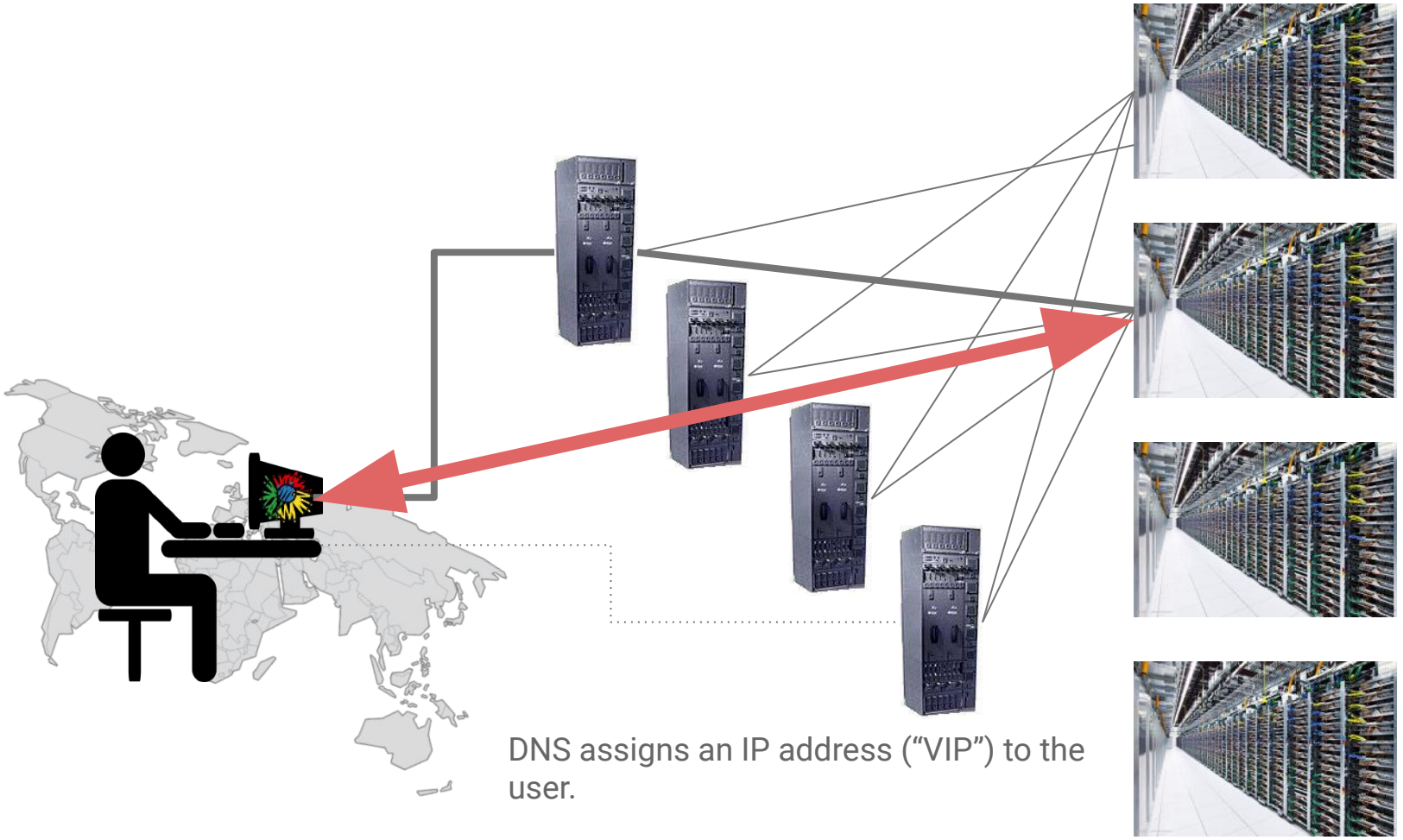
Informal picture of web browsing, before the reverse proxy...





DNS timeouts ~ 5 minutes
DNS server uses IP geolocation and data center load to assign frontend

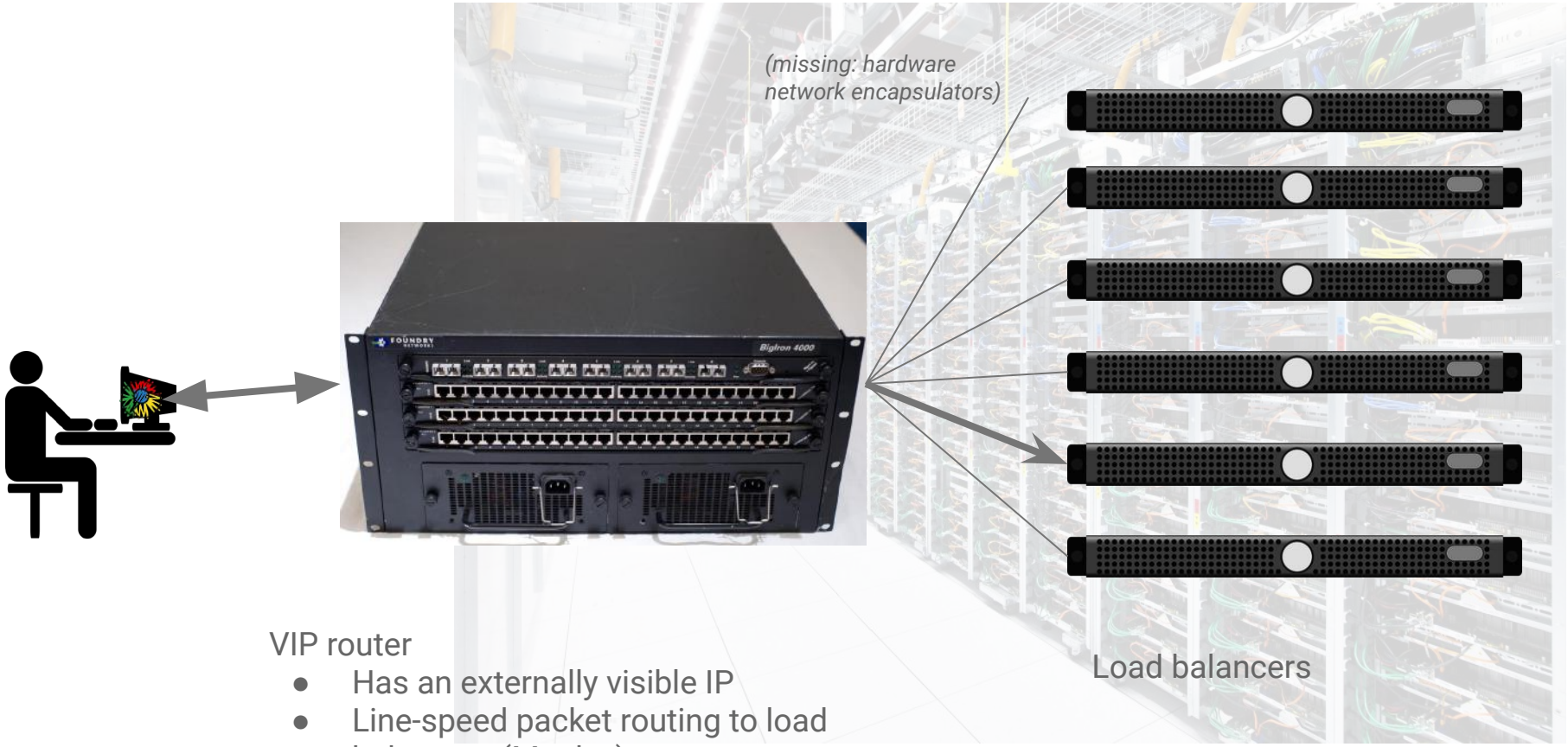
Example: Actually Serving the Internet



DNS assigns an IP address (“VIP”) to the user.

Now is it simple?

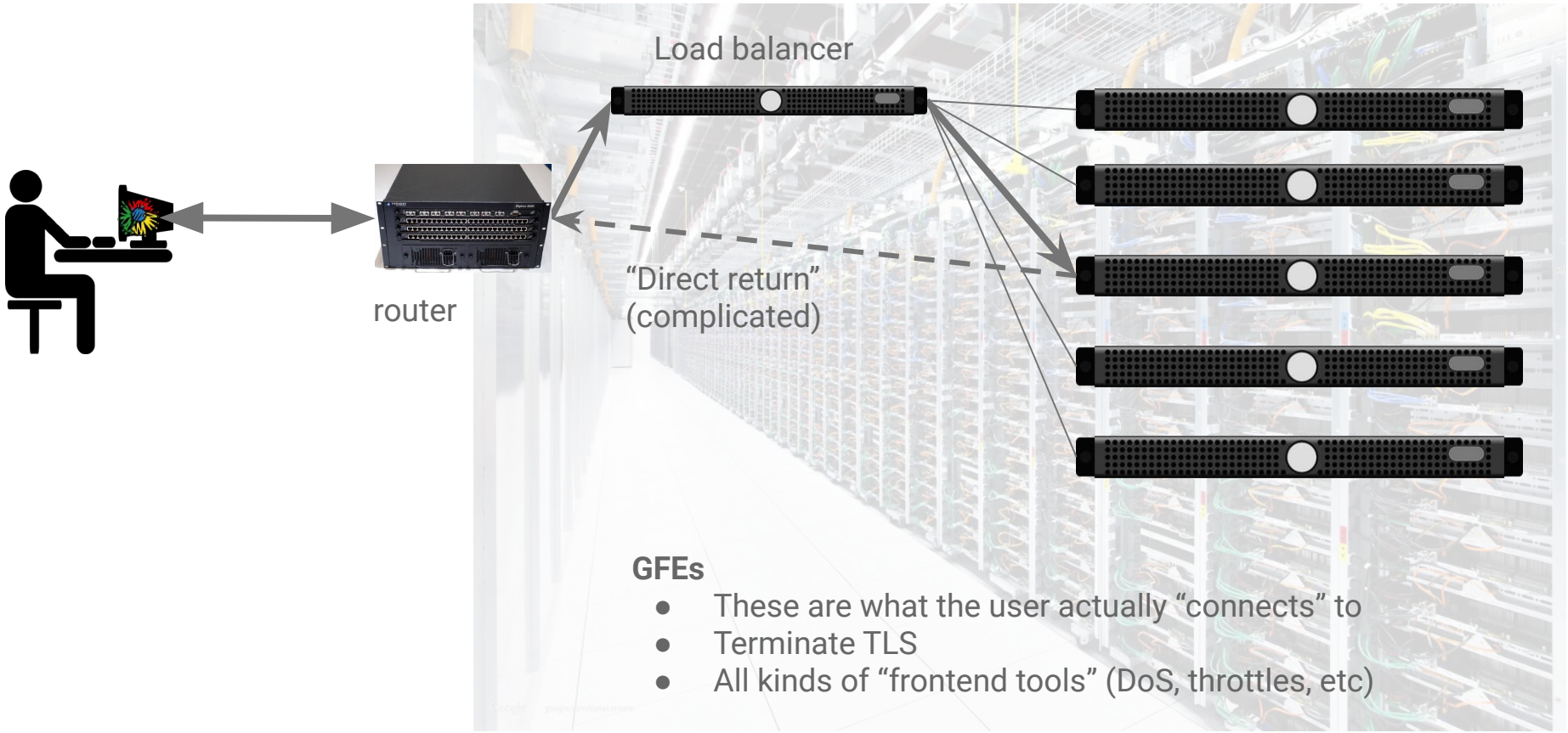
Example: Actually Serving the Internet



VIP router

- Has an externally visible IP
- Line-speed packet routing to load balancers (Maglev)
- IP affinity

Example: Actually Serving the Internet

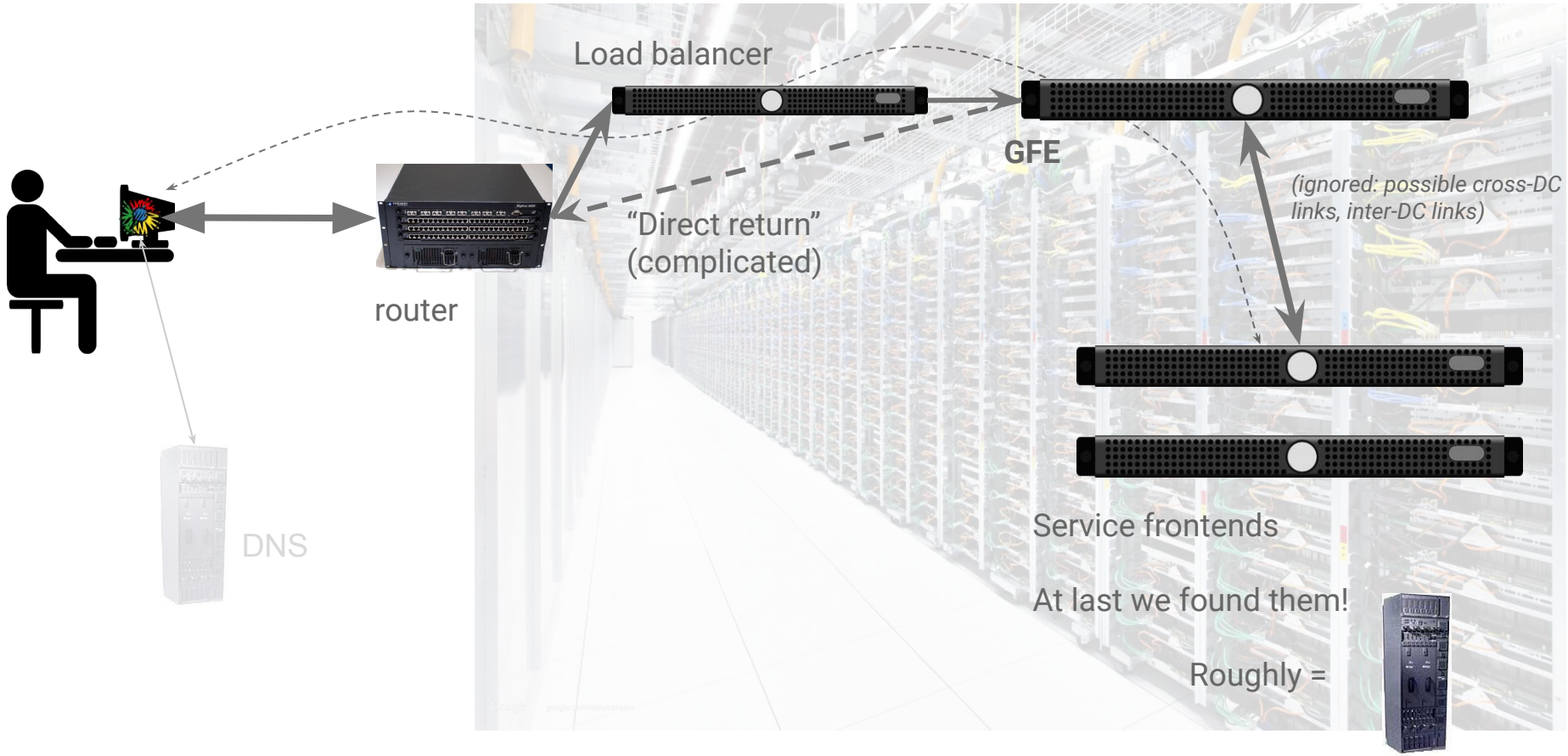


GFEs

- These are what the user actually “connects” to
- Terminate TLS
- All kinds of “frontend tools” (DoS, throttles, etc)

Still not at the specific service ….

Example: Actually Serving the Internet



Example: Actually Serving the Internet

Performance critical for load balancing

Think about the routing tables and stream accounting the router needs:

- To run at ~line speed 10Gbps packet stream
- Small input packets translates to 10M packets/sec
- 100ns/packet. Single-threaded, that would be about 350 machine code instructions

Need:

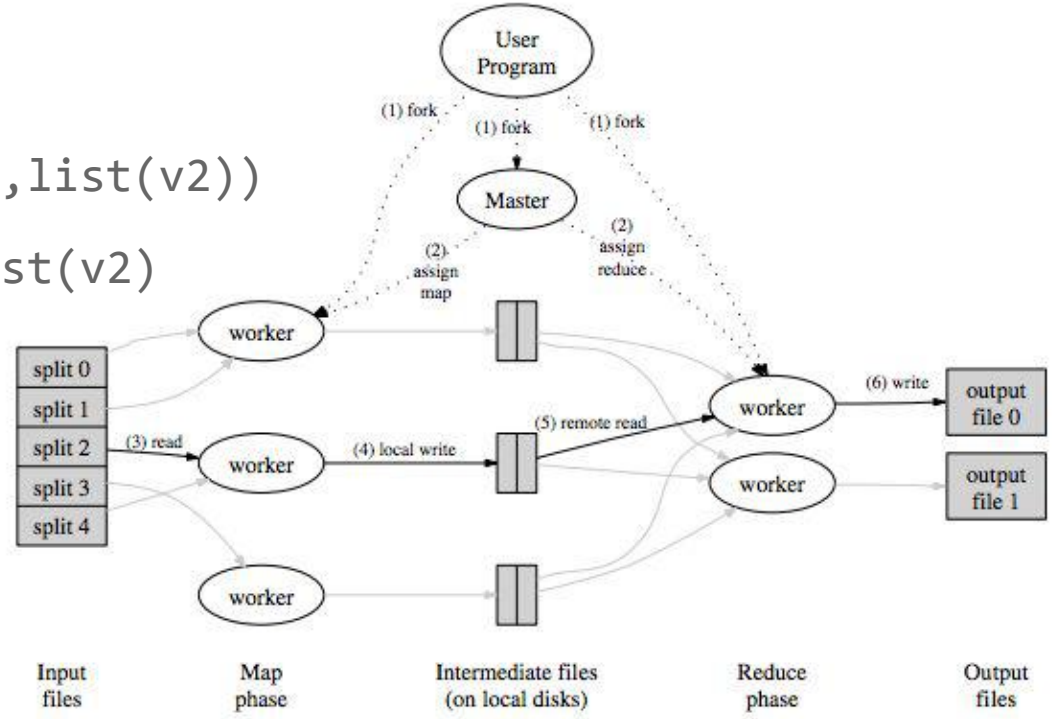
- Extremely fast hashing and memory structures
- Total control of NICs
- High resistance to CPU cache misses
- Load shedding

Example: MapReduce

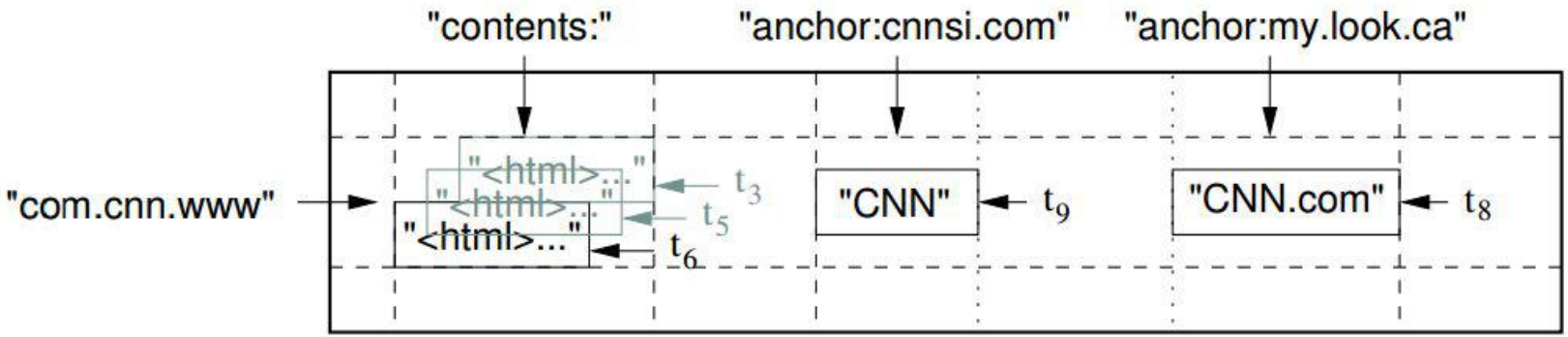
```
map: (k1,v1) → list(k2,v2)

shuffle: list(k2,v2) → (k2,list(v2))

reduce: (k2,list(v2)) → list(v2)
```

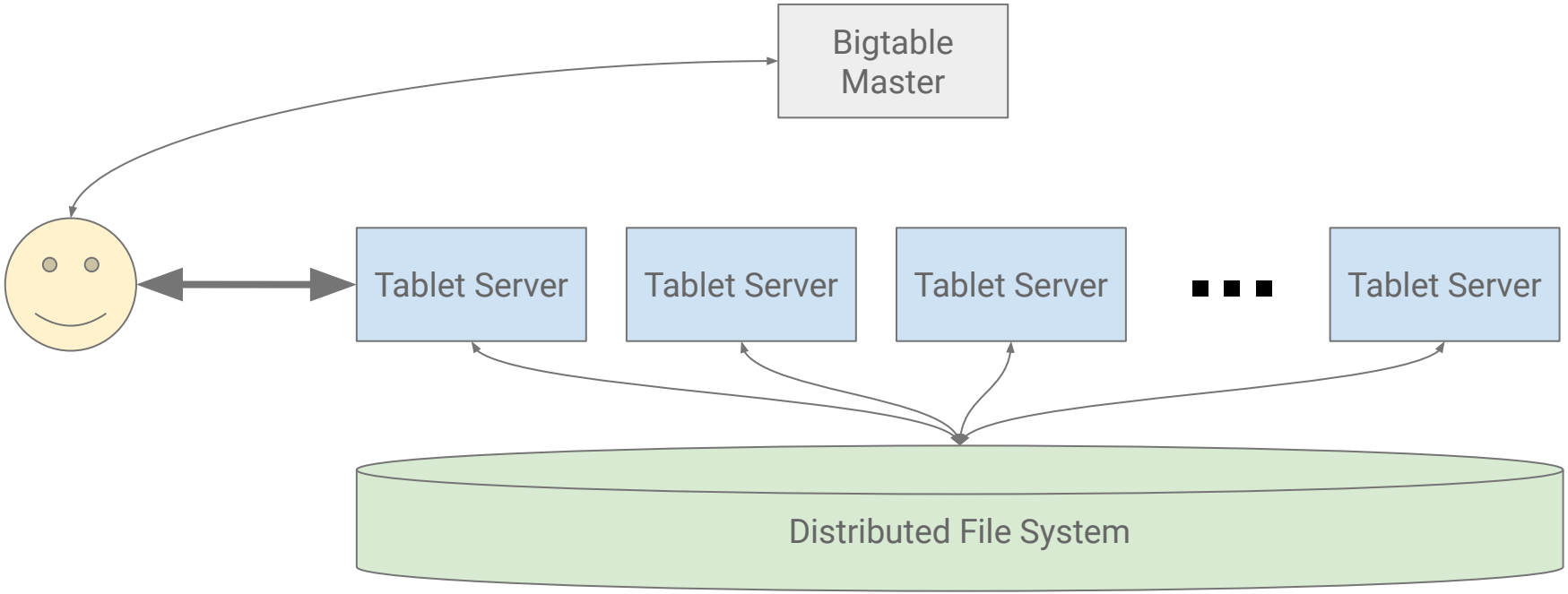


Example: Bigtable (data model)



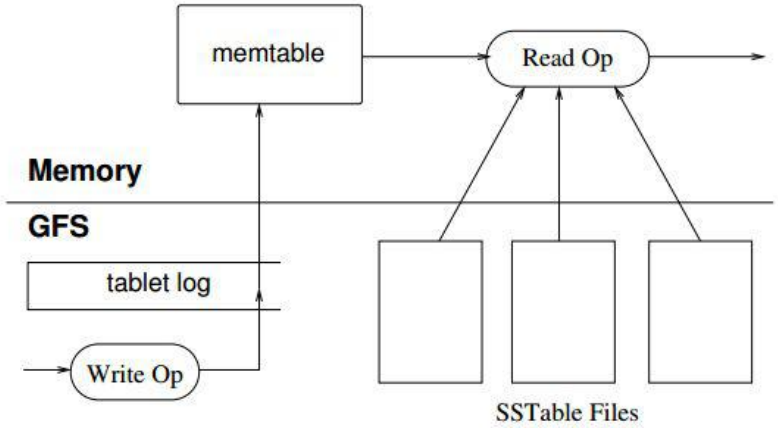
Data Model

Example: Bigtable (system layout)

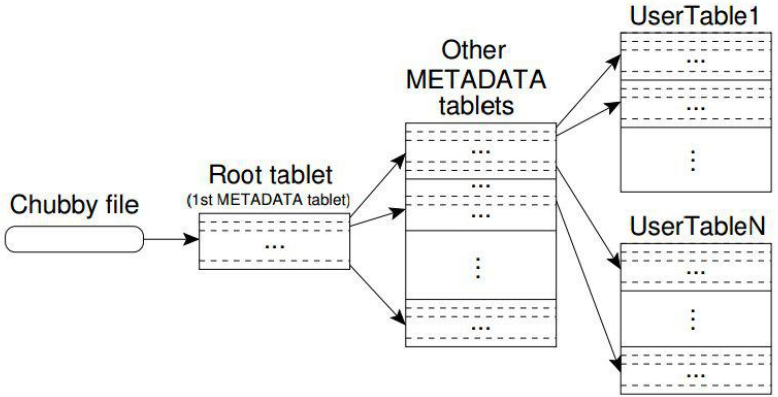


System Architecture

Example: Bigtable (application flow)



Read/Write Data Flow



Tablet Assignment

Example: Spanner (logical data model)

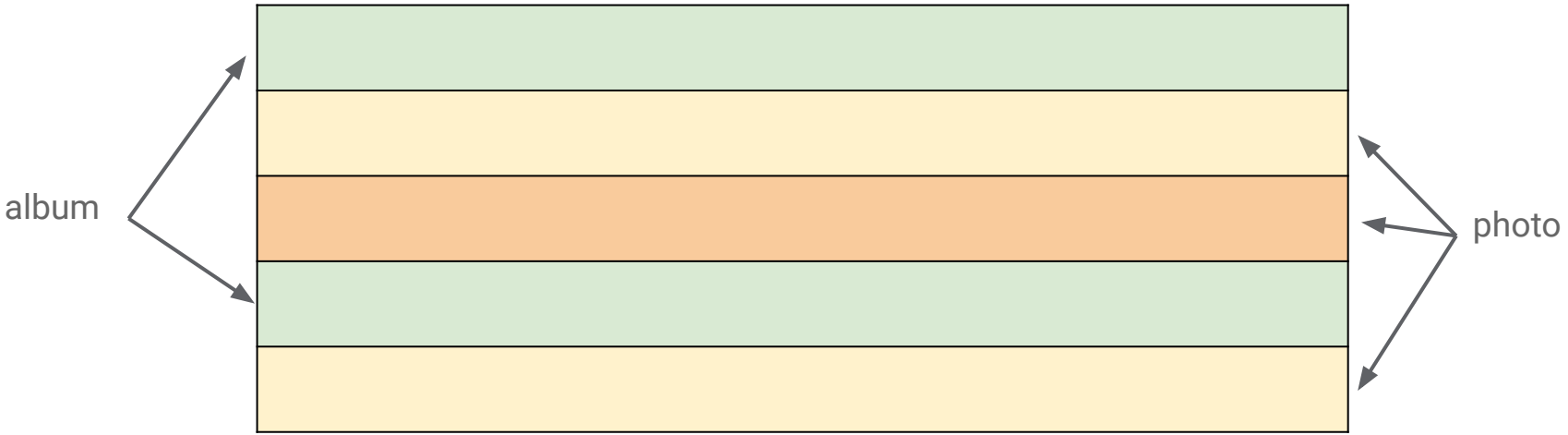
Albums

user_id	album_id	name
1	1	Maui
1	2	St. Louis

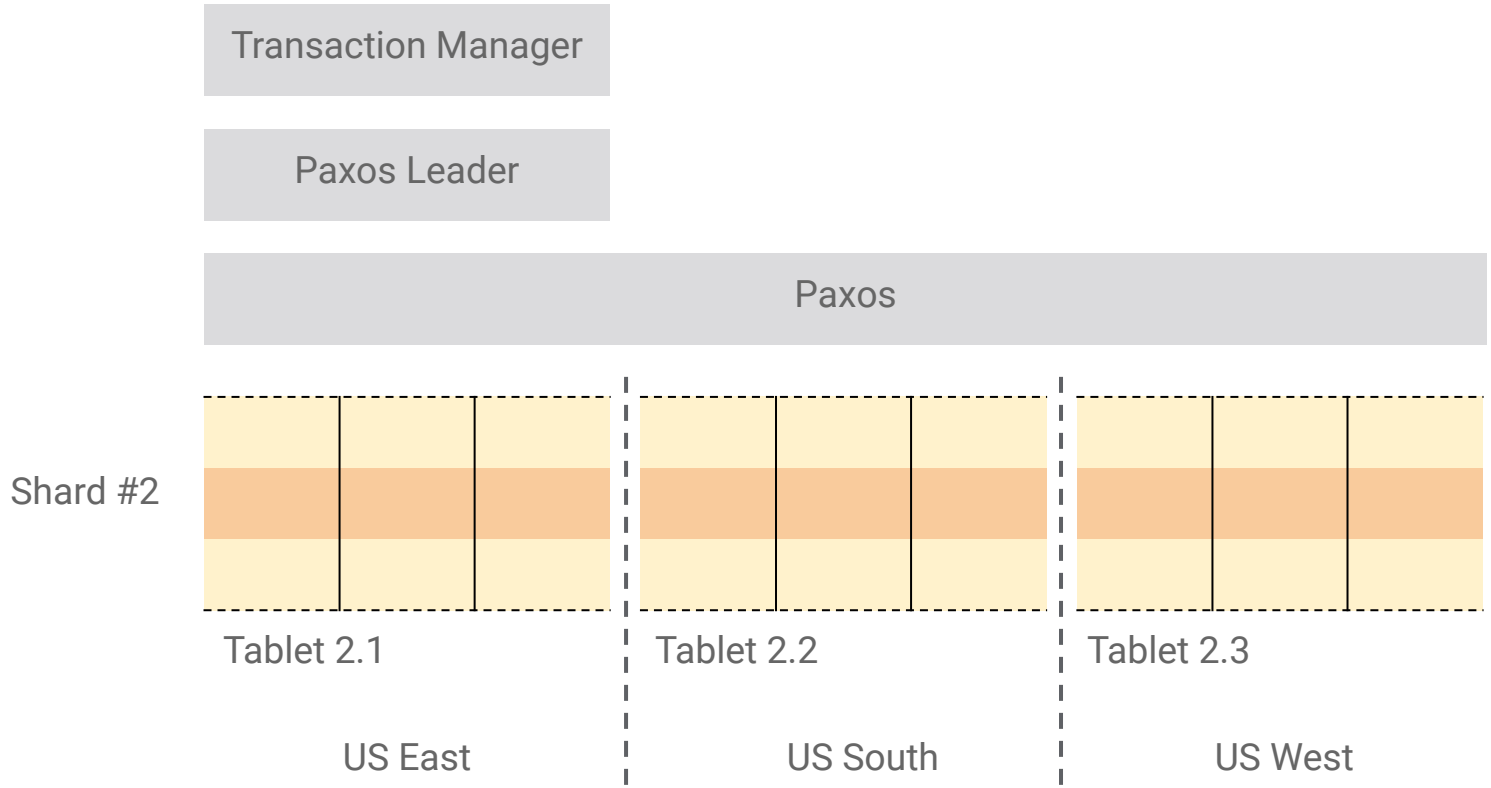
Photos

user_id	album_id	photo_id	title
1	1	2	Beach
1	1	5	Snorkeling
1	2	3	Gateway Arch

Example: Spanner (physical data model)



Example: Spanner (replication)

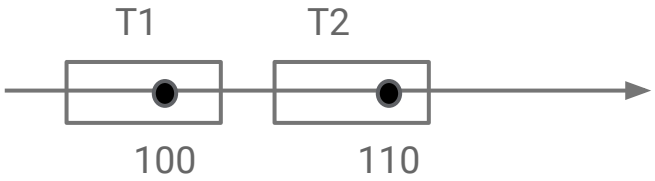


Example: Spanner (external consistency)

Definition:

> If T1 commits before T2 starts, T1 should be serialized before T2. In other words, T2's commit timestamp should be greater than T1's commit timestamp.

Note: Applies even if T1 and T2 don't conflict.

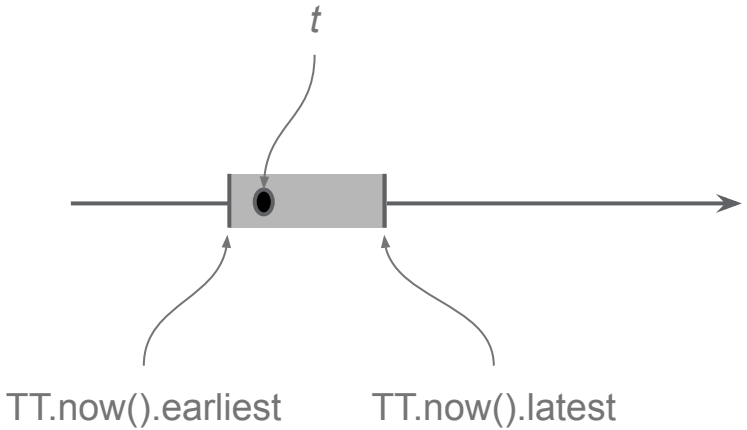


Example: Spanner (transaction protocol)

Idea: There is a global “true” time *t*

TT.now() = [earliest, latest] ∋ *t*

- TT.now().earliest definitely in the past
- TT.now().latest definitely in the future



Using TrueTime to serialize transactions:

1. Acquire locks
2. Execute reads
3. **Pick commit timestamp T = TT.now().latest**
4. Replicate writes (through paxos)
5. **Wait until TT.now().earliest > T**
6. Ack transaction commit
7. Apply write
8. Release locks

Distributed System Design Patterns

Distributed system design patterns

- There are design patterns for code
  - Things like Factory, Visitor, etc.
- There are also common system design patterns
- This (not exhaustive) list contains many of the patterns we commonly see in systems we work with

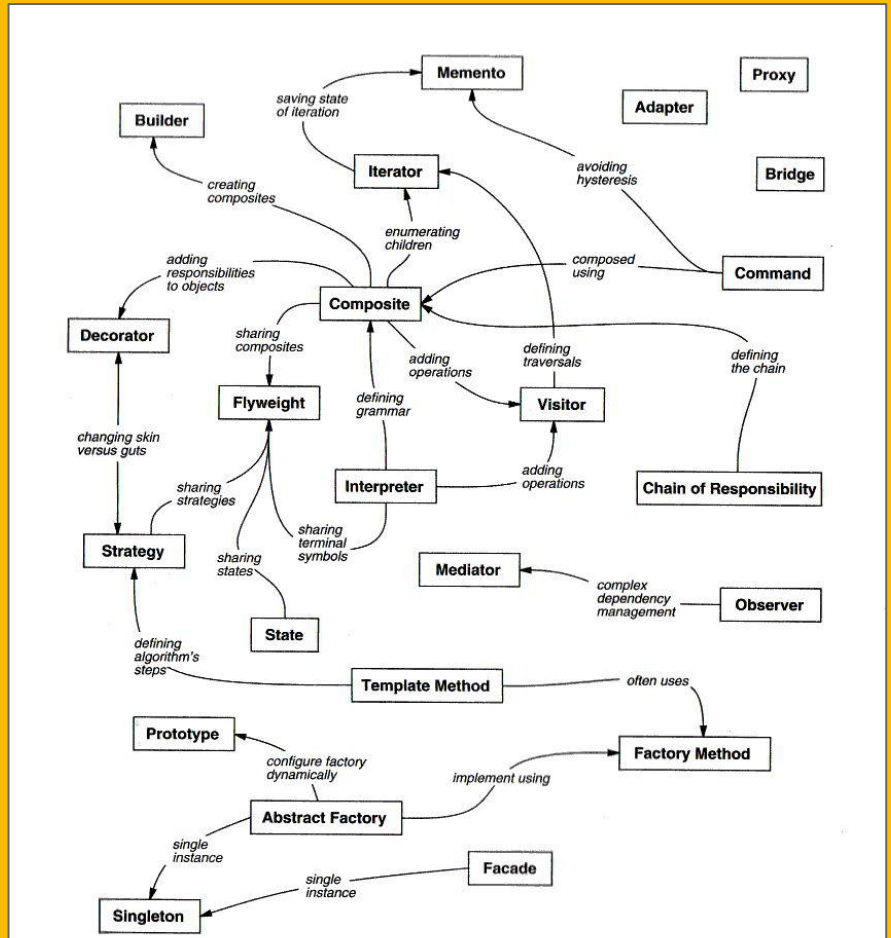


Figure 1.1: Design pattern relationships

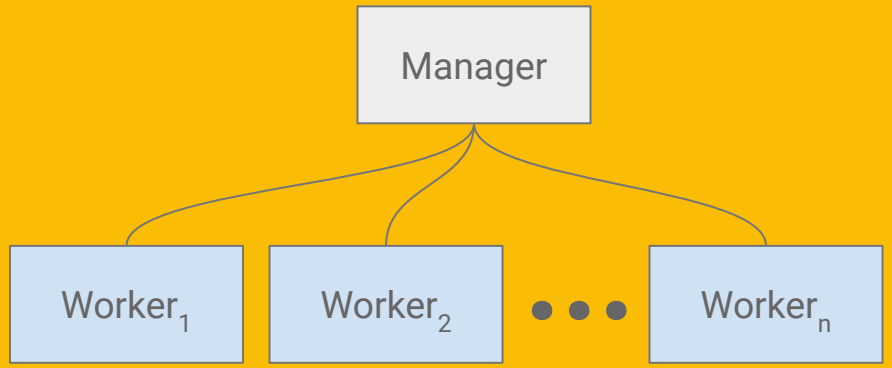
Manager / worker

Key features:

- Useful when problem is data parallel
- Divides load across several identical jobs
- Coordination done by manager

Examples:

- Load balancing
- MapReduce master
- Bigtable master



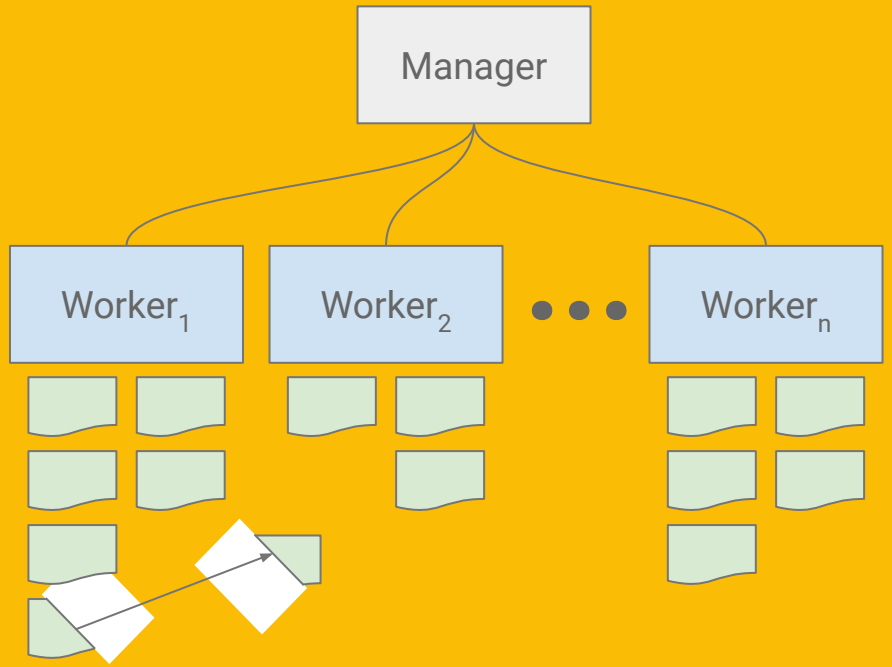
Dynamic (load-based) sharding

Key features:

- Useful when single worker maxed out
- Divides that worker's problem across several other workers
- Can easily add more workers to increase total assigned resources

Examples:

- Reallocation of map shards (MR)
- Load-based tablet splitting (BT)



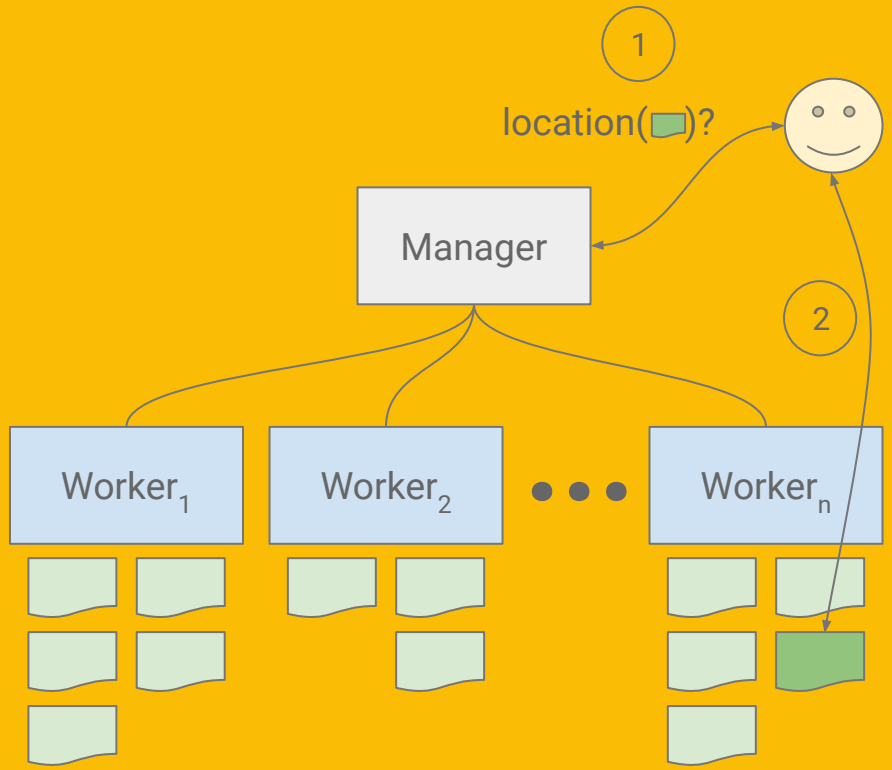
Separate routing path from data path

Key features:

- Route maintained by node that client queries (cached with TTL [1])
- Clients communicate directly with backend that has data [2]
- Increases total ingress/egress bandwidth

Examples:

- Manager routes, TabletServer serves (BT)
- Load balancers



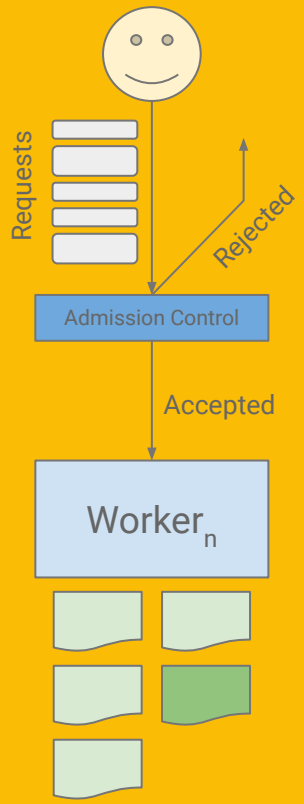
Admission control

Key features:

- Make sure request can be serviced before accepting request
- Prevents memory exhaustion on overload caused by queuing requests
- Often handled directly in the select() loop

Examples:

- Push back in Bigtable
- Load balancer



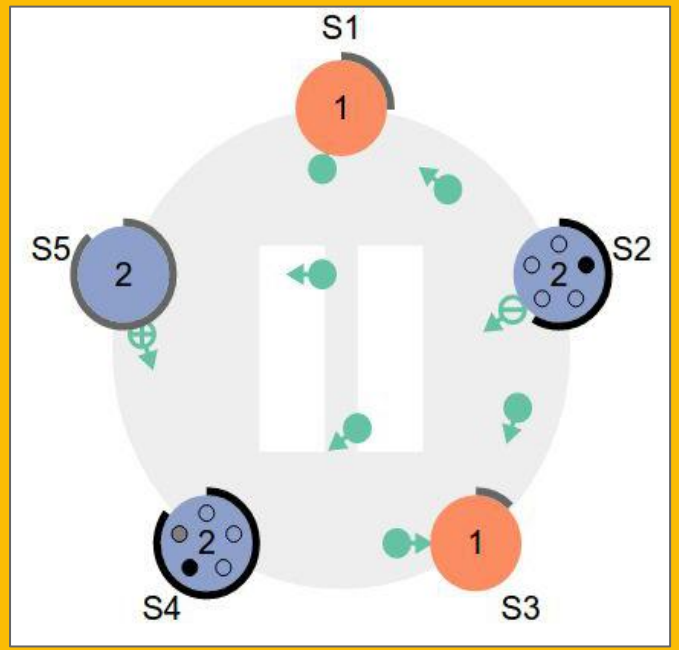
Distributed consensus

Key features:

- Allow group of distributed processes to agree on some state change
- Often implemented using Paxos or RAFT
- Helps deal with network partitions when some nodes may be unreachable (more on this in a bit)

Examples:

- Bigtable master election
- Spanner transactions



Decoupling through queuing

Key features:

- Allows systems to process work at different rates
- When consumers are slow, work is queued in some durable store
- Can slow down producers if consumers get too far behind

Examples:

- Commit log & compactions (Bigtable)
- Pubsub

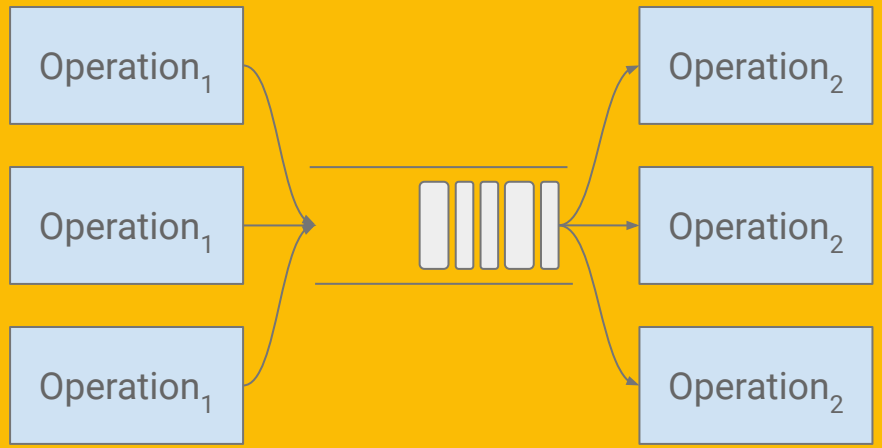
Decoupling through queuing

Key features:

- Allows systems to process work at different rates
- When consumers are slow, work is queued in some durable store
- Can slow down producers if consumers get too far behind

Examples:

- Commit log & compactions (Bigtable)
- Pubsub



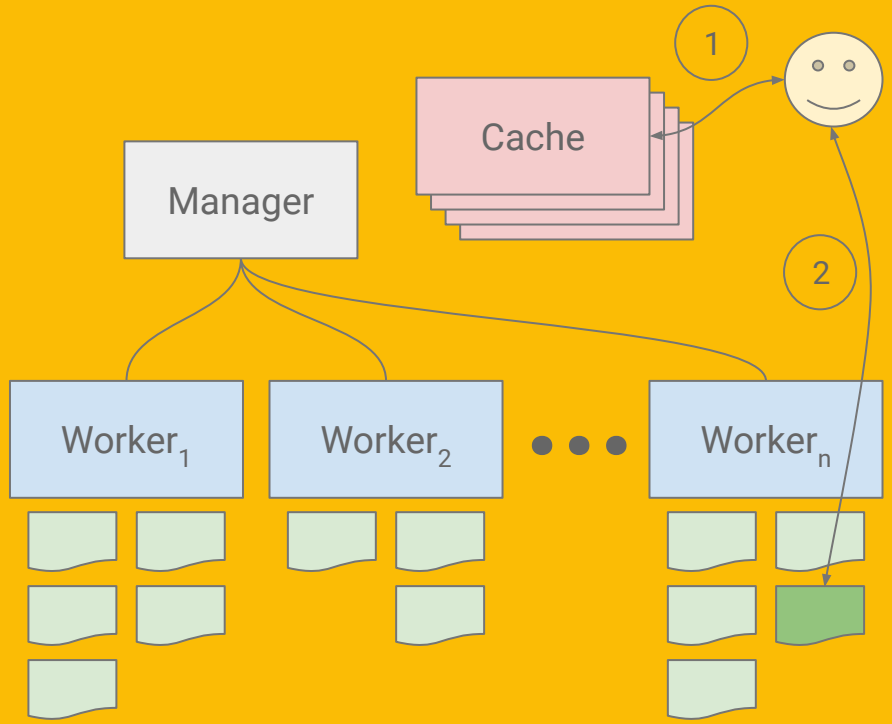
Memcache

Key features:

- Avoid recomputing same result many times within some time window
- Works well with temporal locality of requests or a small working set
- Can save CPU or Spindles
- Doesn't need to be memory only, can be flash to expand working set size

Examples:

- Bigtable (block cache, flash cache)
- Basically everyone...



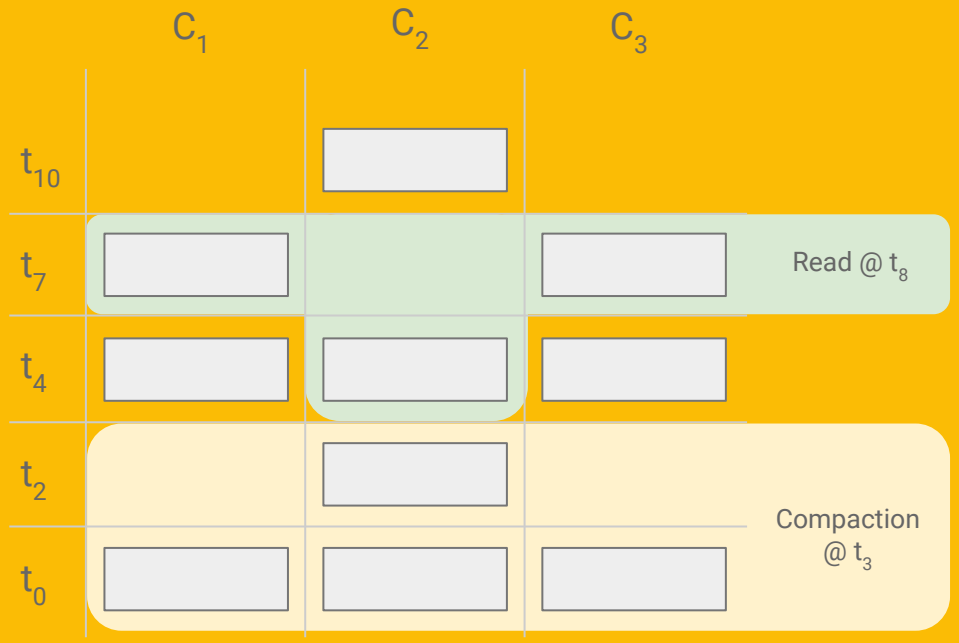
MVCC: Multiversion concurrency control

Key features:

- For recent writes, stores several timestamped versions
- Allows interleaved reads and writes without exclusive locks
- Requires garbage collection to clean up duplicative versions

Examples:

- Percolator Distributed Transactions
- Spanner Snapshot Reads
- Many DBs like MySQL, BDB, etc.



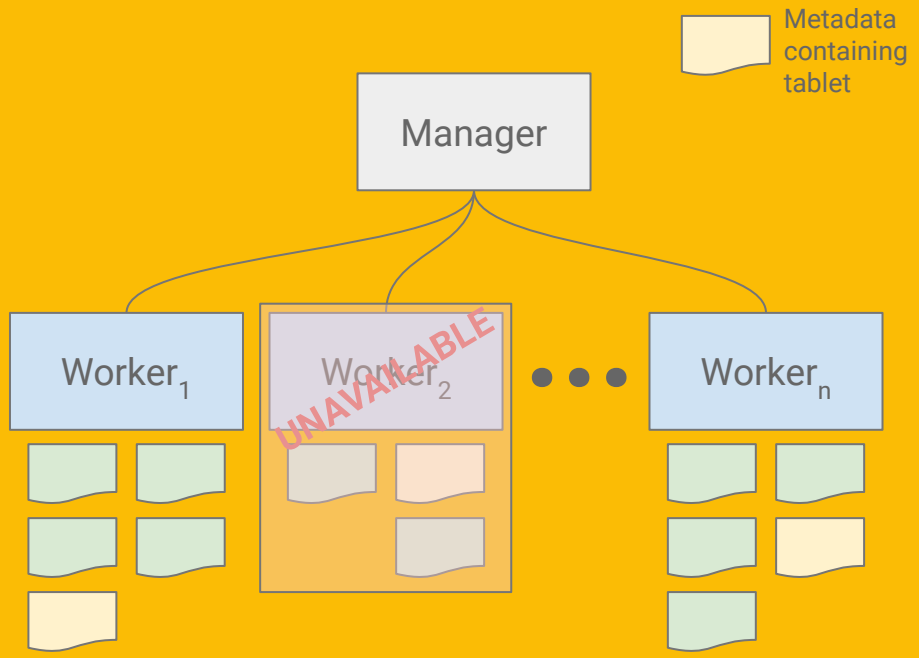
Bad: Storing metadata in the system itself

Key mis-features:

- When system is loaded heavily, metadata may not be accessible
- Causes you to invent ever-higher QoS tiers to make sure metadata read/write operations succeeded

Examples:

- Bigtable metadata



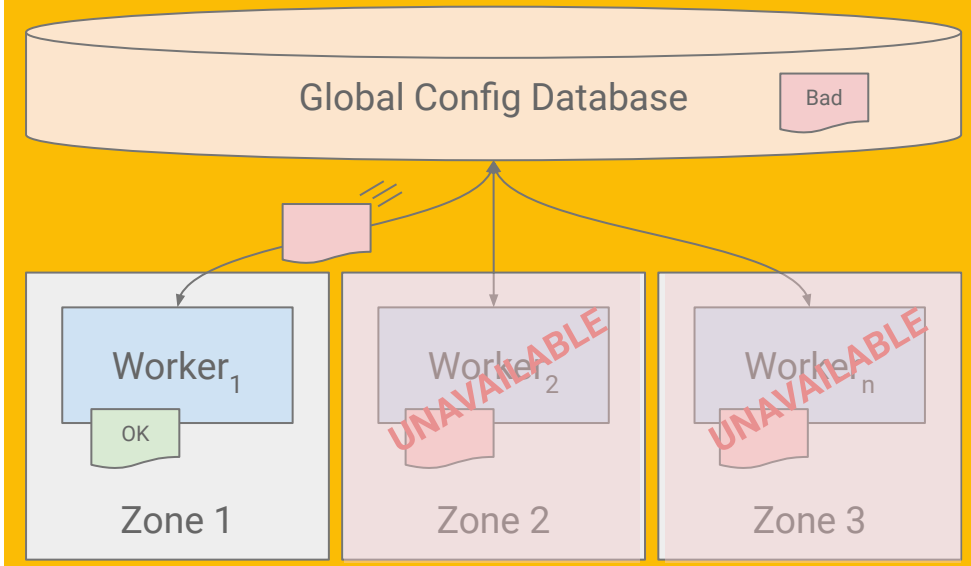
Bad: Synchronous global config rollout

Key mis-features:

- All processes consult a central config oracle for current config values
- A bad config is “instantaneously” picked up by all processes in the system
- Can cause global simultaneous failure

Examples:

- Bad load balancers



Known Hard Problems

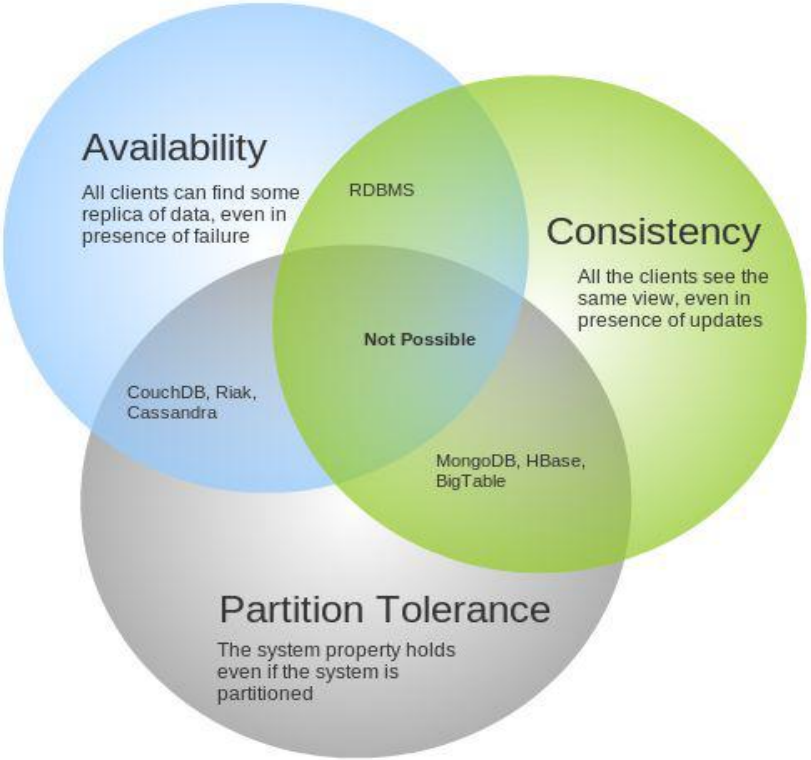
A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

-- Leslie Lamport

Known hard problems

- There are many problems in distributed systems that are known to be hard
- It is useful to keep these in mind when designing a system, and address them as early as possible

CAP theorem



You can only have 2 of the following:

- Consistency
- Availability
- Partition tolerance

However, all of these are rather important.

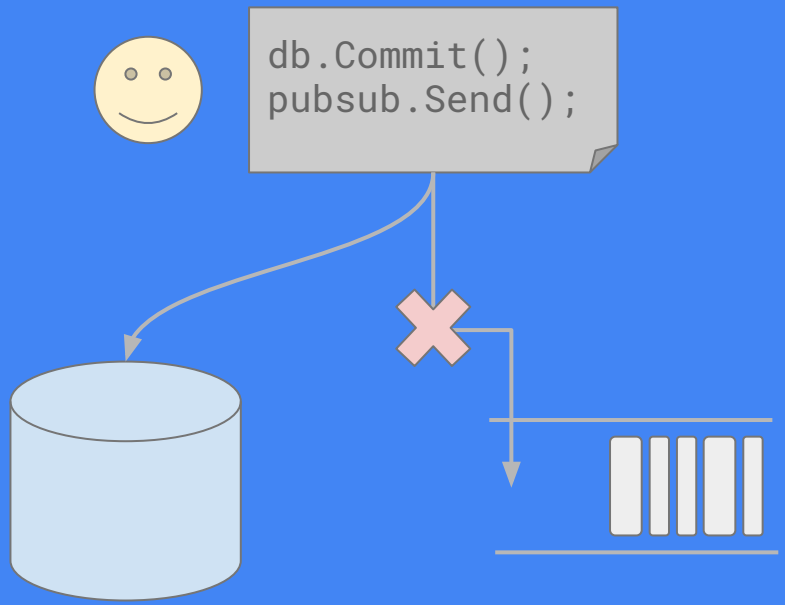
- Need to keep these failure modes in mind when designing your application
- Need to understand tradeoffs your backends are making in this space

Commits across atomicity domains

How do you order two operations to safely deal with crashes?

- If the `db.Commit()` succeeds but the `pubsub.Send()` fails, is that ok?
- How about the opposite? What if the subscriber receives a message but nothing is in the database?

Typically requires making downstream operations idempotent (ie, they can be safely replayed), but that can be a challenge



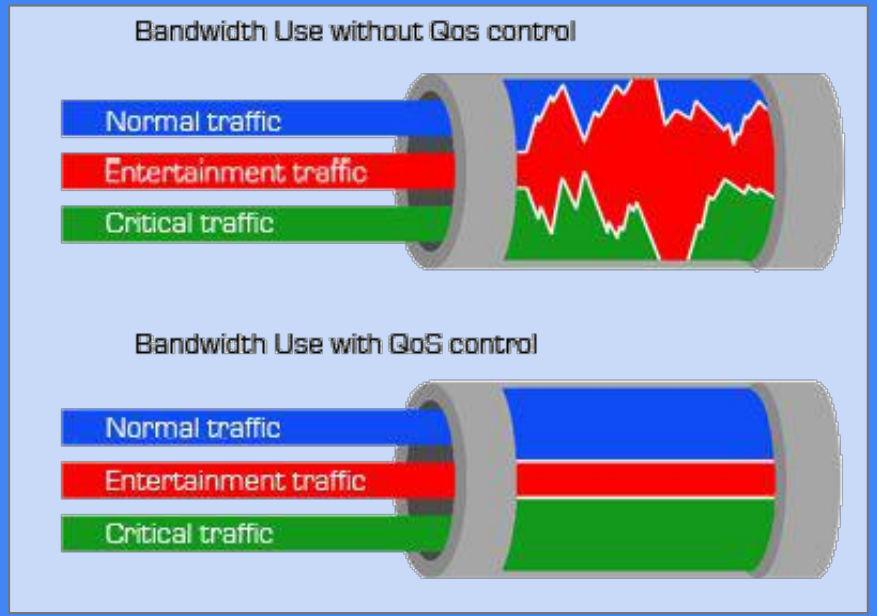
Balancing QoS and utilization

If you don't have QoS:

- You can't control resource allocation
- A low-priority client can dominate resource usage, causing priority inversion

If you do have QoS:

- Your expensive resources may be underutilized when high-priority client doesn't need their allocation



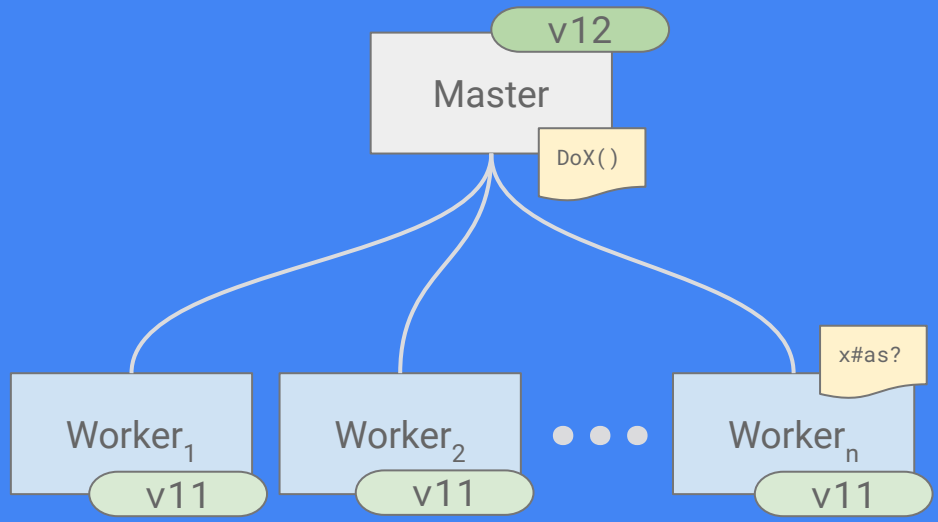
Rollouts and protocol upgrades

Temptation: Stop the world, push new binaries, then restart the world

- Instantaneous protocol upgrades
- Simpler intermediate semantics

But:

- Rarely possible in practice
- Makes rollbacks dangerous



Instead:

- Make a sequence of backwards compatible changes
- Requires more care when designing the system's architecture

Reprocessing

You want data to appear as if all results were computed with the latest code

- But code changes pretty rapidly
- Too much data to recompute in a reasonable amount of time/budget
- For example, if your op takes 0.1 cpu-sec and you have 1B input objects, that will take 1000 cpu-days

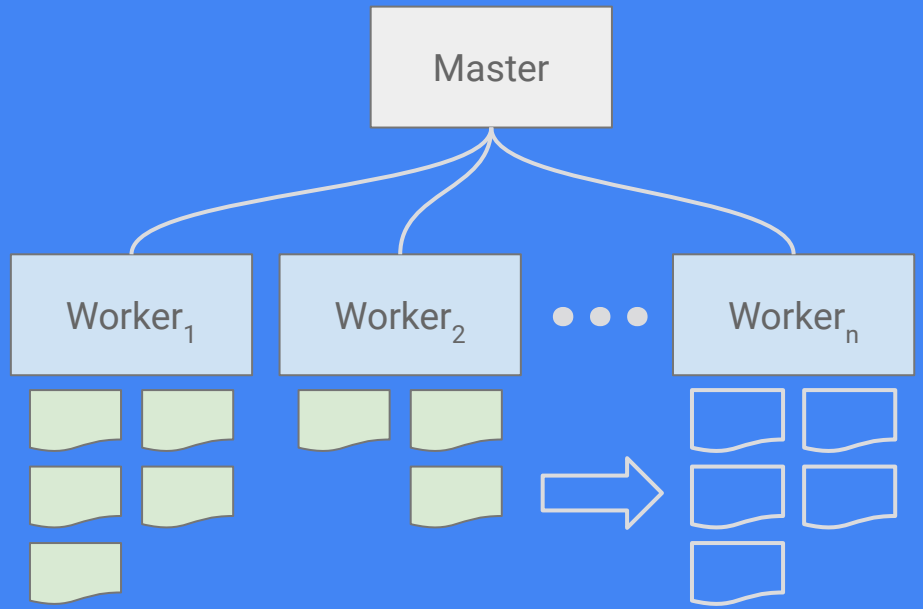
Need to selectively recompute the right part of the dataset

Data Moves

Sometimes must move whole system to another physical location

- Hard to move while the system continues to run
- Usually high bandwidth to move your application incrementally
- For example, Moving 1PB over a 10Gbit link would take ~7d

Often, it would be faster and cheaper to just physically ship hardware…



Common Failure Modes of Distributed Systems

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

-- Leslie Lamport

Common failure modes of distributed systems

Correctness bugs:

- Can often be detected in regression tests
- These systems are often data-parallel, so you can test correctness with limited resources

Performance bugs:

- Hard to catch in synthetic tests
- Typically needs full scale and representative load to properly test

Above: Common error screen around the time you were a baby

Producer-consumer rate mismatch

Even if you have admission control, a rate mismatch can be very problematic

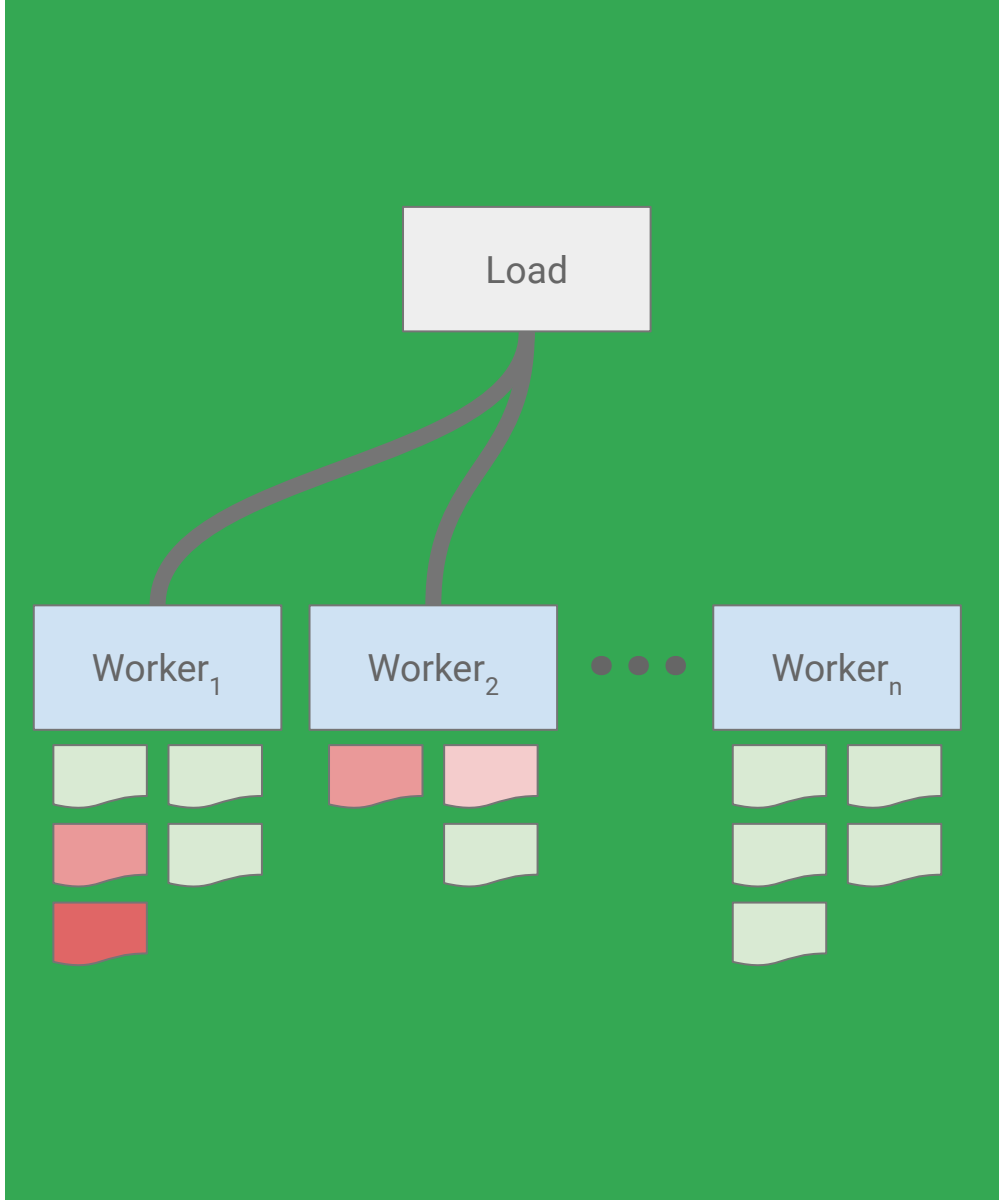
- Best case, the back pressure propagates through your system and the system stalls or the failure is visible to the user
- Worst case, some node in the middle of your computation fails because it is overloaded

Hotspots

The load on your system can be concentrated onto a small working set.

- Results in the machines hosting that fraction of the data getting hotter than average and increasing queuing delay
- Even with load shedding, the node can “heat up” faster than you’d normally rebalance

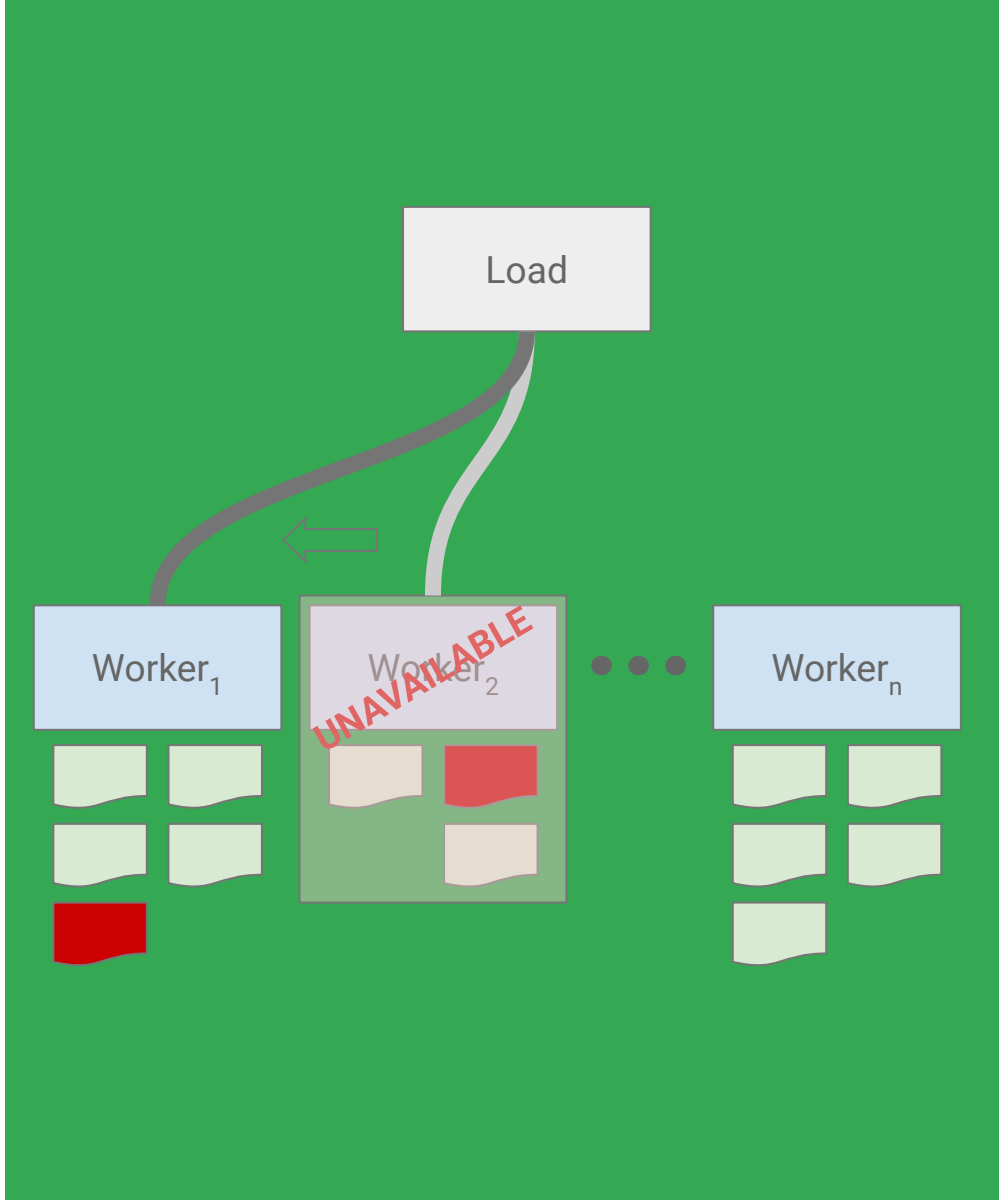
Need to seek ways to make your algorithms hotspot resistant.



Death ray

The evil cousin of the hotspot...

- Typically happens when the load is so overwhelming that admission control can't prevent the process from failing
- However, the load doesn't cease
- When your system recovers and loads those resources onto another node, the death ray follows



Amplified rare events

Even rare things tend to be common at scale:

- Bug triggers 1 in 1M operations, running 10 ops/sec, this bug will trigger daily!

These types of failures abound. A few examples that we've observed:

- Rare checksum computation failure because of processor issue
- Remote cache restart causes tablet server failure

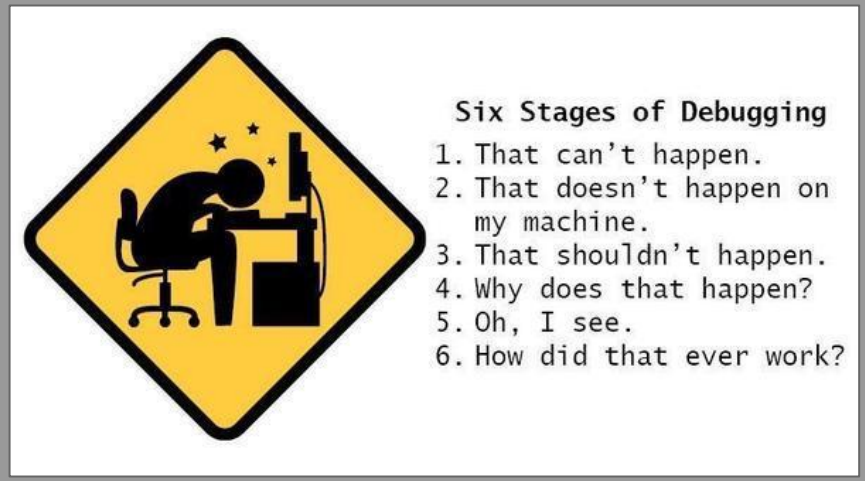
Debugging Distributed Systems

Debugging Distributed Systems

Once you've built a system, you often live with it for a long time.

You need tools to help you understand what the system is doing:

- At a micro scale: per job
- At a macro scale: interactions between jobs



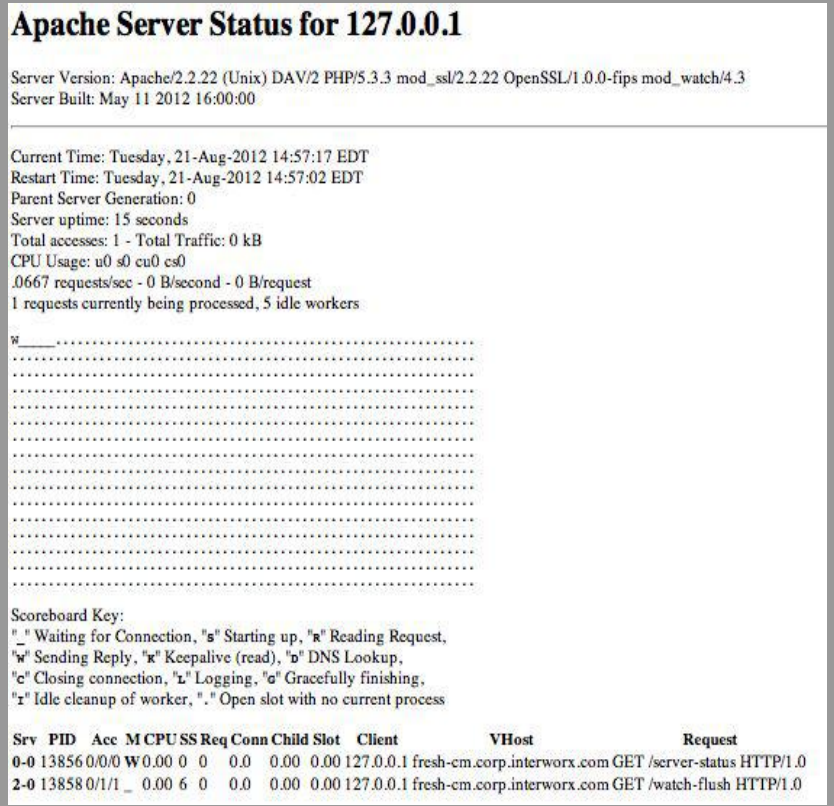
Status Pages

Key features:

- Each process serves over HTTP
- Gives you a local understanding of what is going on in a single process
- Can expose state of internal data or reasons for recent decisions

Especially useful status pages:

- Stacktrace of all currently live threads
- State of all currently pending RPCs



Apache Server Status for 127.0.0.1

Server Version: Apache/2.2.22 (Unix) DAV/2 PHP/5.3.3 mod_ssl/2.2.22 OpenSSL/1.0.0-fips mod_watch/4.3
Server Built: May 11 2012 16:00:00

Current Time: Tuesday, 21-Aug-2012 14:57:17 EDT
Restart Time: Tuesday, 21-Aug-2012 14:57:02 EDT
Parent Server Generation: 0
Server uptime: 15 seconds
Total accesses: 1 - Total Traffic: 0 kB
CPU Usage: u0 s0 cu0 cs0
.0667 requests/sec - 0 B/second - 0 B/request
1 requests currently being processed, 5 idle workers

W____.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Scoreboard Key:
"_" Waiting for Connection, "S" Starting up, "R" Reading Request,
"W" Sending Reply, "K" Keepalive (read), "D" DNS Lookup,
"C" Closing connection, "L" Logging, "G" Gracefully finishing,
"I" Idle cleanup of worker, "." Open slot with no current process

Srv	PID	Acc	M	CPU	SS	Req	Conn	Child	Slot	Client	VHost	Request
0-0	13856	0/0/0	W	0.00	0	0	0.0	0.00	0.00	127.0.0.1	fresh-cm.corp.interworx.com	GET /server-status HTTP/1.0
2-0	13858	0/1/1	_	0.00	6	0	0.0	0.00	0.00	127.0.0.1	fresh-cm.corp.interworx.com	GET /watch-flush HTTP/1.0

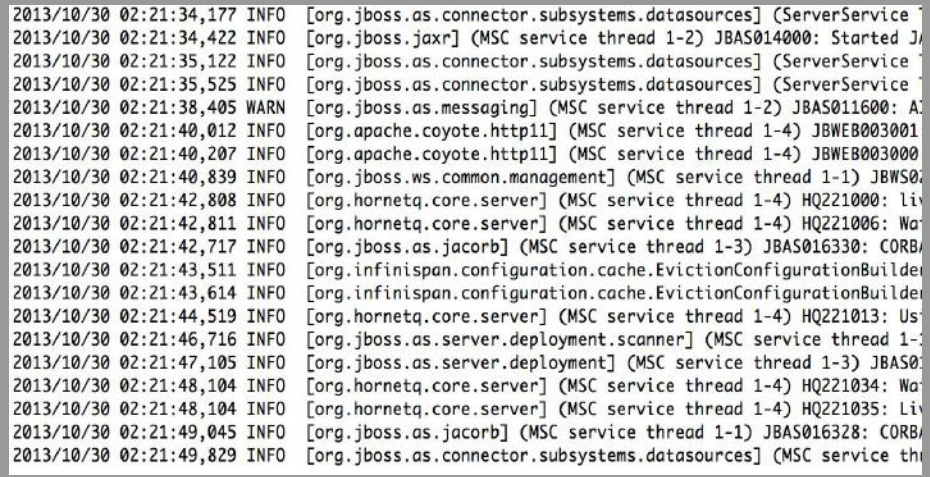
Logging

It is critical to produce useful logs.

- Logs are very useful for debugging failures
- Since you don't know when failures are going to happen, you need to log all the time

What to log:

- When and what requests arrive
- Details about erroneous conditions
- What part of the code is involved



Local Request Tracing

Logs are useful, but they interleave events

- To remedy this, you can maintain request specific logs, often called traces
- Typically only kept for active requests, so they are less useful for retrospective debugging
- Very useful for answering the question: “What is going on right now?”

```
// Annotate a request trace.

const string& request = ...;
if (HitCache()) {
  TRACEPRINTF("cache hit for %s",
              request.c_str());
} else {
  TRACEPRINTF("cache miss for %s",
              request.c_str());
}
```

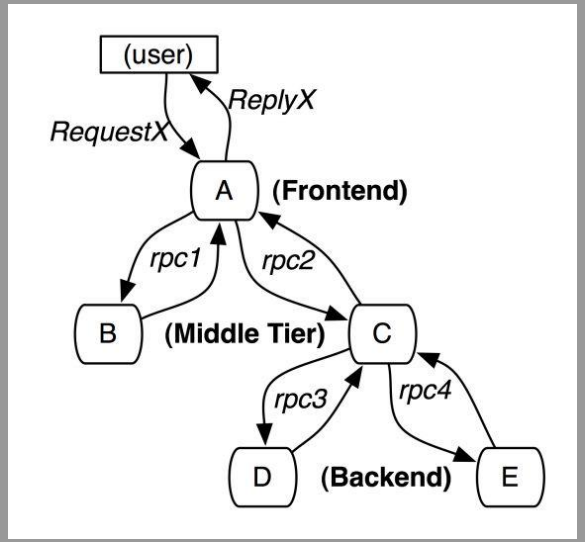
Distributed Tracing

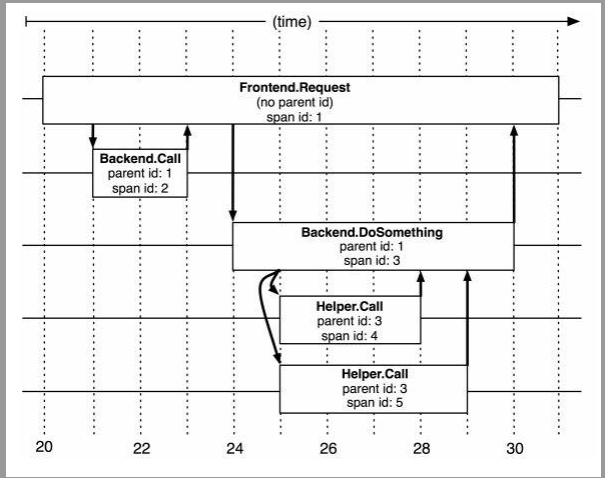
Key features:

- Integrated into each component of the whole system
- A token, representing a logical operation, is passed along each RPC
- Stats are logged by each process and a system aggregates and visualizes the data
- Tend to be verbose, so they are sampled

Types of stats:

- Latency
- Resource Usage (CPU, Disk, etc)





Paper: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure

https://bit.ly/3N1KEqK

- Tweet: What anti-pattern is your webserver using?
- Tweet: What anti-pattern is your team following?

  Team dynamics is important!