

CS5630

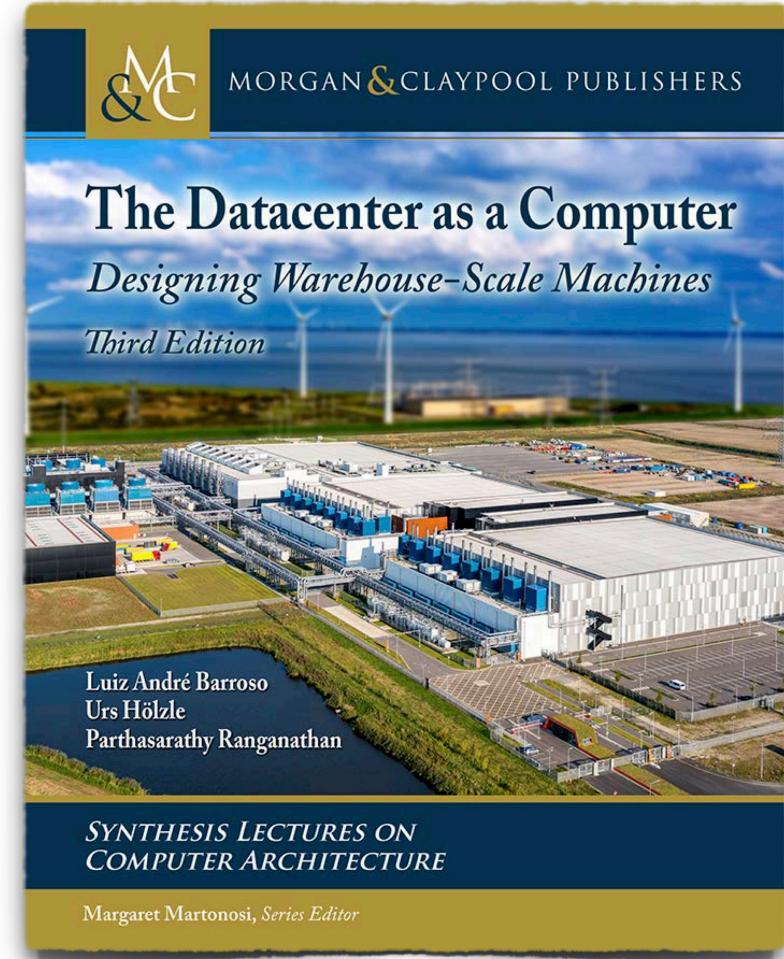
Datacenters (2): Software Infrastructure

Prof. Supreeth Shastri
Computer Science
The University of Iowa

Lecture goals

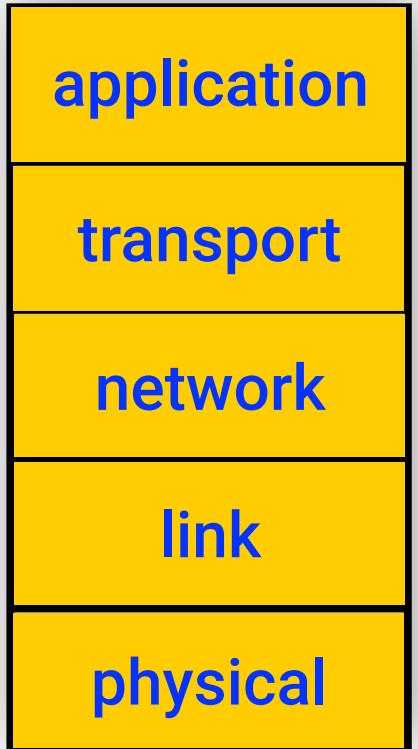
*Deep dive into software systems that make
WSC and datacenters work*

- Datacenter systems stack
- WSC applications
- WSC Software tradeoffs



Chapter 2

Systems stack (that we already know)



What is **layering**?

An approach to **designing complex systems**

- allows *identifying system's components and explicitly defining their relationship*

Modularization eases maintenance and updating of system

- *change in layer's service implementation: transparent to rest of system*

Why is layering useful for **computer networks**?

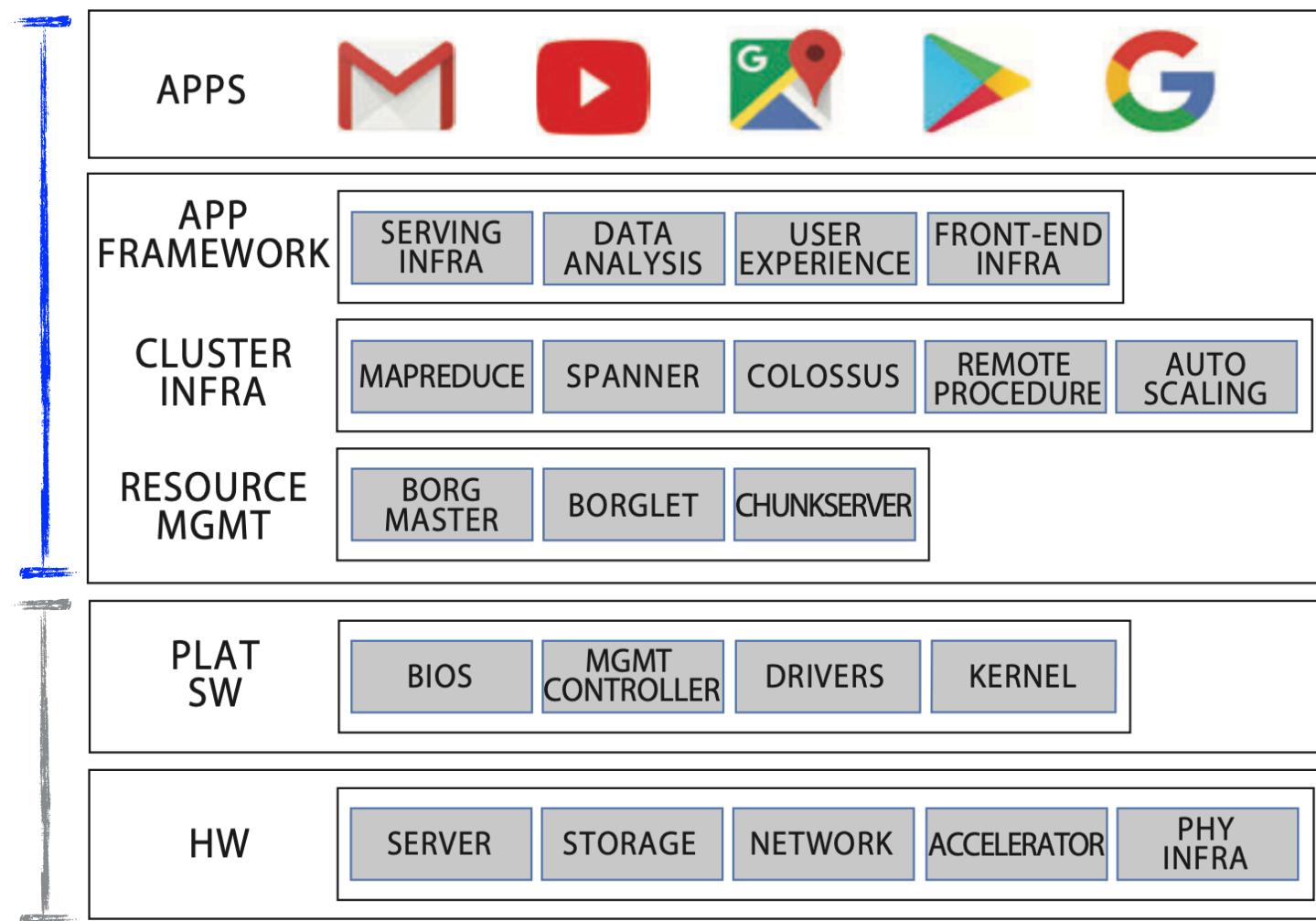
Computer networks have multitude of components interacting with each other

- *host devices, routers, links, protocols, applications, policies, and so on*

Internet is arguably the **largest engineered system** ever created by humans!

Datacenter systems stack

Focus of TODAY's lecture



Focus of NEXT lecture

Resource Management

The layer that maps user tasks to hardware resources

Key responsibilities

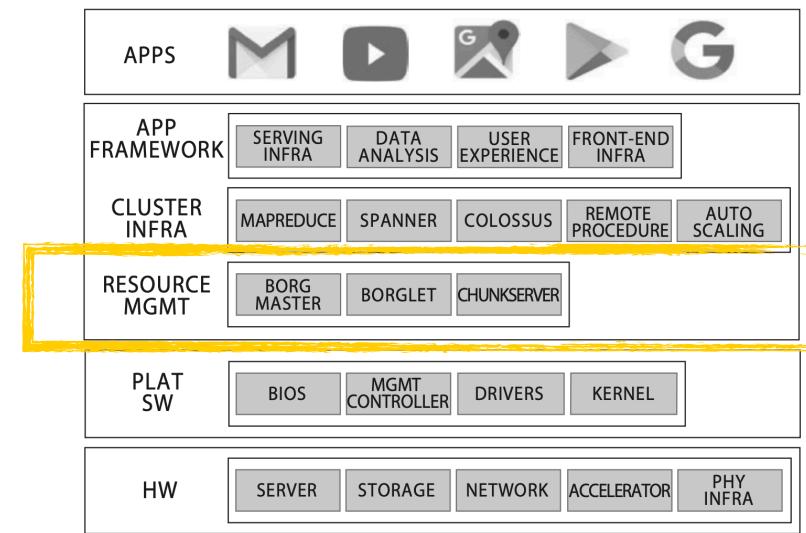
- ▶ Basic task management services
- ▶ Enforcing resource utilization quotas and priorities
- ▶ Efficient use of hardware resources
- ▶ Failure management (both for hardware/user jobs)

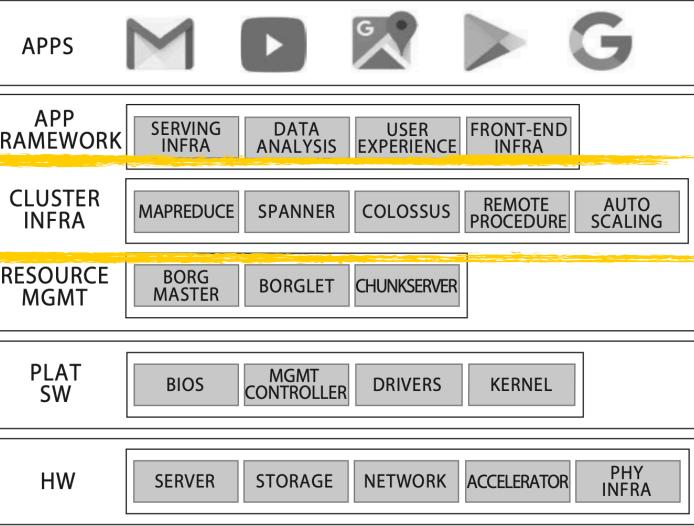
Service-levels

- ▶ In *simpler* forms, it would allow manual allocation of groups of machines to a given user or job
- ▶ In *advanced* forms, it would allow automate allocation and enable sharing at finer granularity

Real-world examples

- ▶ Google's Kubernetes, Borg
- ▶ Mesos, YARN, and Twine (all these three are in our set-2 research papers!)





Cluster Infrastructure

The layer that provides cluster-level functionalities

Key responsibilities

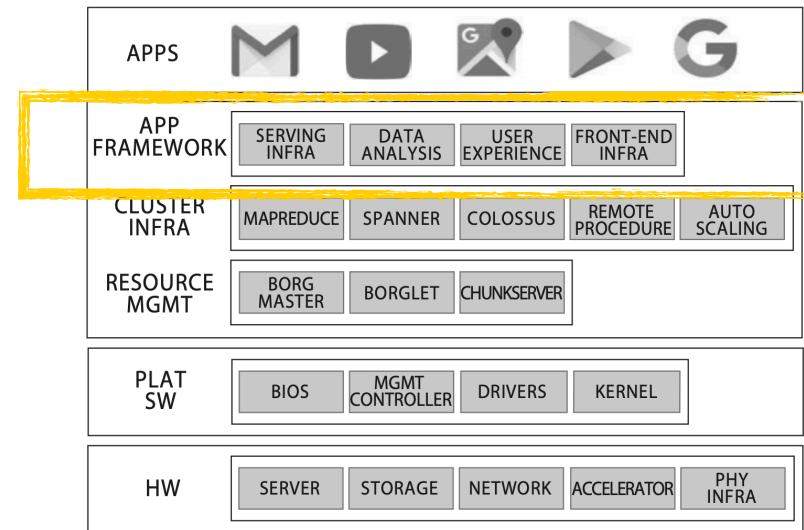
- ▶ Distributed computation and distributed storage
- ▶ Remote procedure calls (RPCs)
- ▶ Message passing and synchronization
- ▶ Cluster-level management (software images, tools for debugging/optimization)

Real-world examples

- ▶ *MapReduce, Hadoop, Spark* for distributed computing (included in set-2 research papers!)
- ▶ *Google File System* for distributed storage (included in set-2 research papers!)
- ▶ *X-Trace* for performance debugging

Application Framework

The layer that provides WSC abstraction for applications



Key responsibilities

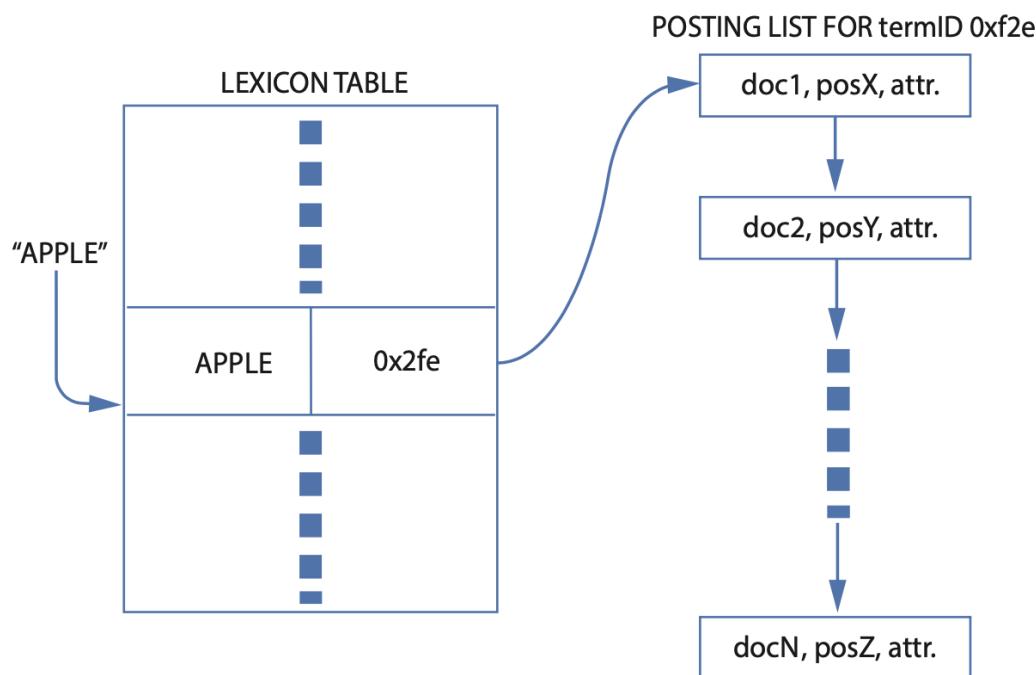
- ▶ Hide the inherent complexity of underlying WSC infrastructure
- ▶ Provide programming frameworks that ease of application development

Real-world examples

- ▶ *MapReduce, Hadoop, Spark* for distributed computing (included in set-2 research papers!)
- ▶ *Amazon RDS* for creating/operating SQL databases on the cloud
- ▶ *Google AppEngine* for developing scalable web apps

Applications on WSC

Web Search



Application design

- ➡ Search engines crawl and collect WWW documents into a repository
- ➡ Next, they build an index by inverting the documents into a lexicon table
- ➡ Each lexicon entry identifies a list of documents that contain that term + metadata (such as its position etc)

Scale (*only a ballpark since WWW is always growing*)

- ➡ WWW contains 100 billion documents (a conservative estimate)
- ➡ Average document size of 4KB (after compression)
- ➡ Crawl DB will be 400TB
- ➡ Inverted index will be on the same order of magnitude

Web Search

Runtime operation

- ▶ Receives a user query
- ▶ Looks up invested index to find all documents that contain all search terms
- ▶ Rank the resulting documents (e.g., PageRank)
- ▶ Return the highest ranked documents to the user

Key metrics

- Latency (user's experience)*
- High throughput (volume of queries)*
- Scaling up/down (workload variation)*

Mechanics under the hood

- ▶ *Index partitioning*: Given the massive scale of data, the index is split across a large number of machines
- ▶ *Replication*: For throughput and fault tolerance, multiple copies of index files are placed on different machines
- ▶ *Load-balancing*: A front-end server distributes the queries across multiple machines, combines the ranked lists, procures the actual documents identified in the top results, and returns a search-result view

Video Serving

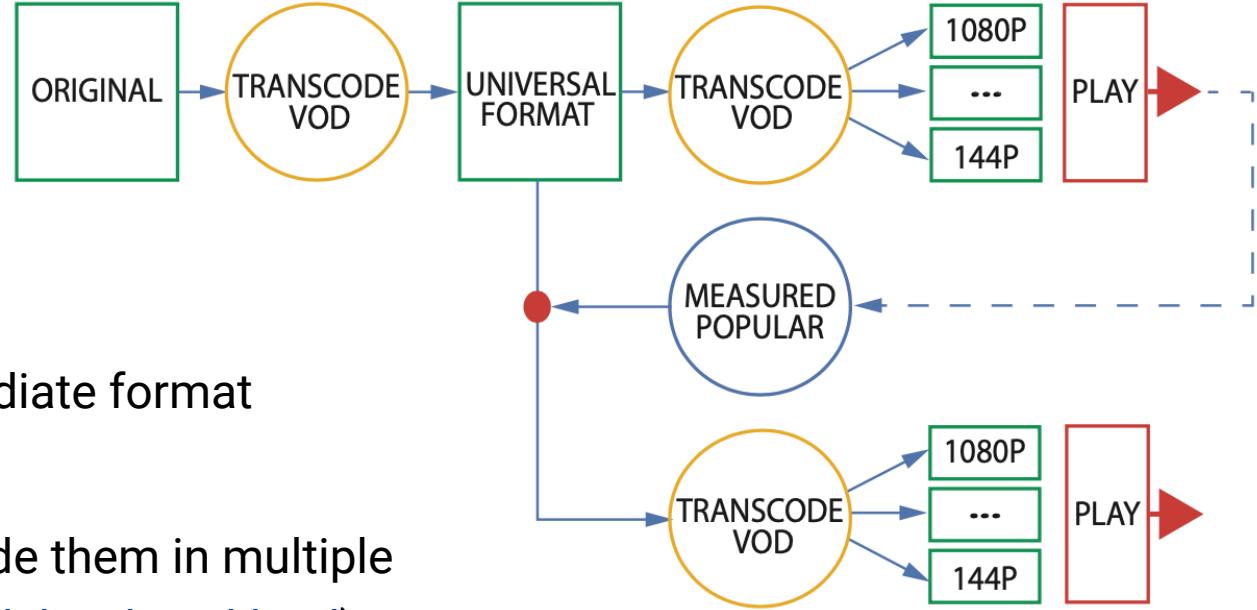
Key challenges

- ▶ *Data volume*: video traffic represents 73% of the global Internet traffic
- ▶ *Scale*: every minute, users upload 400 hours of video, and watch a million hours of video
- ▶ *Storage*: total estimated storage for YouTube videos = 10 million TB (circa 2020)
- ▶ *Heterogeneity*: user devices have wide range of capabilities and bandwidth (both of which may vary w/ time)

Key functionalities

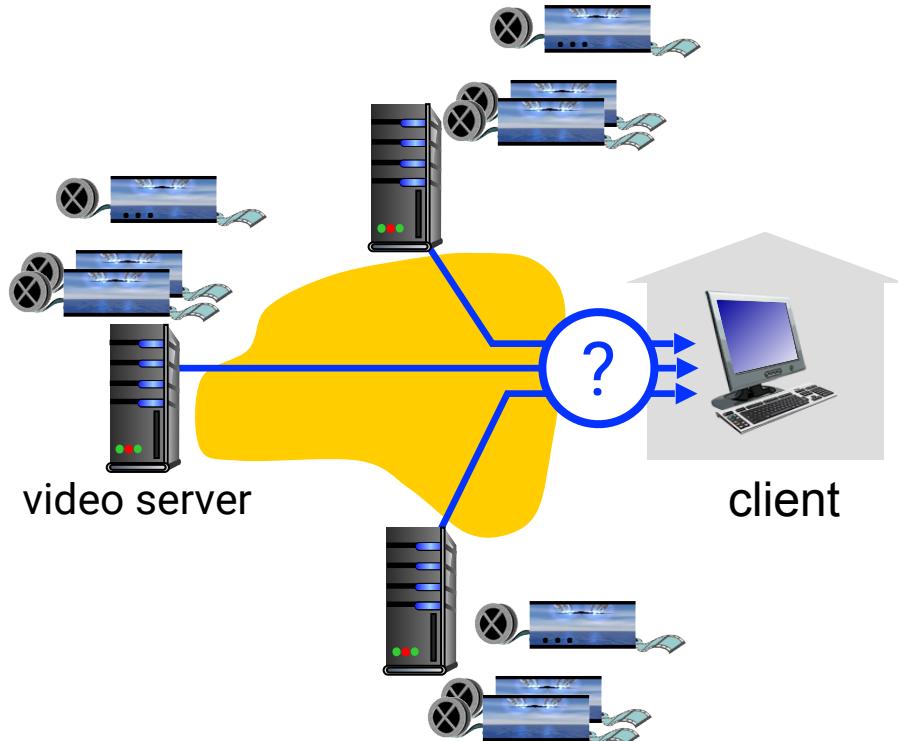
- ▶ *Transcoding* the video: convert the uploaded video into standard set of codecs, resolutions, formats, and frame rates
- ▶ *Storing* the video: both originals and transcoded versions are stored; compression is employed to save storage space
- ▶ *Transmitting* the video: identify the best choice of format, and serve using the closest CDN/edge network

Video Serving (provider perspective)



- ➡ Convert user video into a common intermediate format (*for ease of processing and storage*)
- ➡ Split all videos into small chunks and encode them in multiple output codecs and resolutions (*highly parallelized workload*)
- ➡ Transfer content to appropriate datacenters, CDNs, and edge locations (*for low latency streaming to users*)
- ➡ Identify popular videos and perform enhanced compression/encoding (*to reduce network transmission costs*)

Video Serving (client perspective)



Dynamic Adaptive Streaming over HTTP:

client determines

- **when** to request chunk (to avoid both buffer starvation and buffer overflow)
- **what encoding rate** to request (to maximize video quality when bandwidth available)
- **where** to request chunk (to minimize latency to the destination CDN server)

Tradeoffs in WSC Software

Datacenter vs. Desktop

Ample Parallelism

WSC applications exhibit both data-level and request-level parallelism

- ➡ *E.g., crawling and processing of billions of web pages*
- ➡ *E.g., search requests are independent and are read-only operations*

Workload Churn

Users are shielded from Internet application's implementation (via well-defined APIs and protocols)

- ➡ *"Move fast and break things" i.e., applications are updated more frequently than desktop software*

Fault-free Operation

Failures are much more likely at WSC scale than individual machines (i.e. multiplicative effect)

- ➡ *E.g., WSC applications should expect faults every few hours (not months and years as in the case of desktop software)*

Buy vs. Build

In-house Software

WSC applications and cluster management software are mostly written in-house

- ➡ *E.g., Google has in-house software systems in all the layers of WSC stack*
- ➡ *Traditional IT relies heavily on third-party software like DBMS, OS, utility software etc*

Flexibility and efficiency

These metrics mean different things for WSC vs. traditional IT environments

- ➡ *Flexibility for Google might mean responding to market changes quickly*
- ➡ *For desktop users, flexibility (and economy) comes from the choice of DBMS/OS/utilities*

Lack of third-party software

This trend is largely driven by the fact that WSCs are closely guarded (as opposed to desktop platforms)

- ➡ *How does one design a network stack for Google without the operational knowledge of its WSCs?*

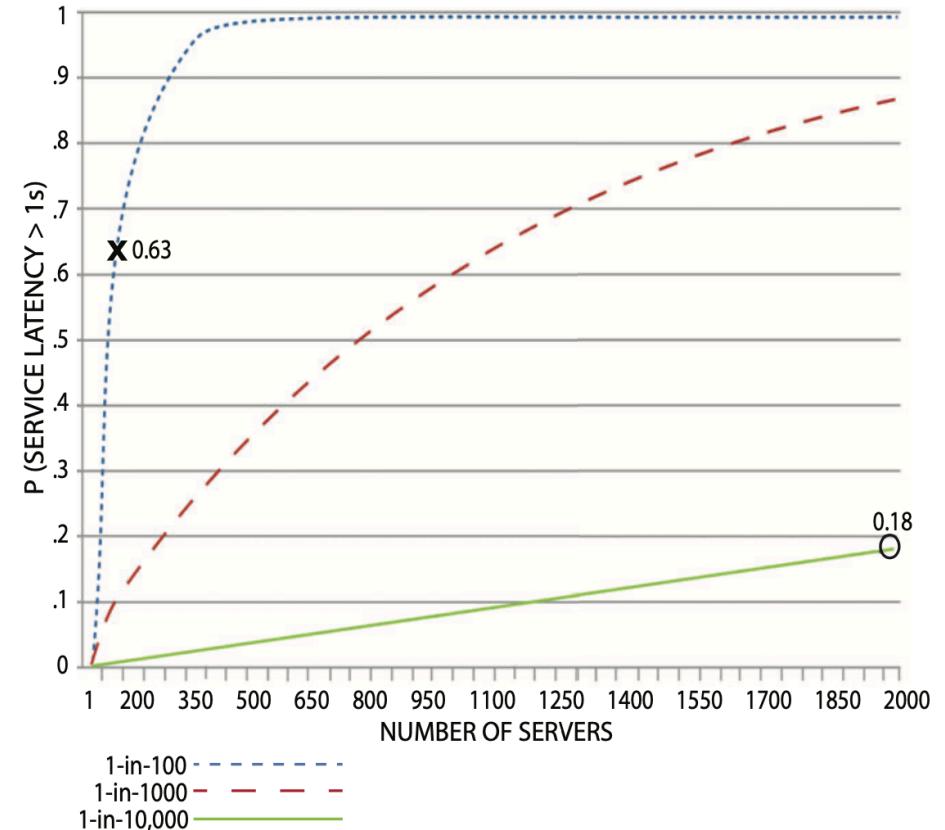
Tail Tolerance

Tail latency refers to the latency of the slowest responses

At WSC scale, variance cannot be fully eliminated

Consider a system where each server responds in 10ms but with a 99th percentile latency of 1s (i.e., 1 out of 100 requests will experience 1s latency)

- If a WSC application uses 100 servers for its work, then 63% of its requests will take 1s (*blue curve in CDF*)
- Let's change this to 1 in 10K requests experiencing 1s delay! Now, if your application uses 2000 such servers, you should expect 18% of requests to violate 1s mark (*green curve in CDF*)



Probability of >1s service level response time as the system scale varies

Performance vs. Availability

| Technique | Advantage | Tradeoff |
|-----------------------------|-----------------------------|---|
| Replication | Throughput and Availability | Adds to complexity, decreases resource utilization |
| Sharding | Throughput and Availability | Adds to application complexity |
| Compression/encoding | Space savings | Adds to computation load, hurts runtime performance |
| Redundant execution | Performance guarantees | Degree of redundancy determines waste of resources |
| Load balancing | Lower latency | Application complexity, requires idempotency/parallelizability |
| Eventual consistency | Performance | Stronger consistency guarantees increase complexity, and hurt performance |