Introduction

CS 675: Distributed Systems (Spring 2020)

Lecture 1

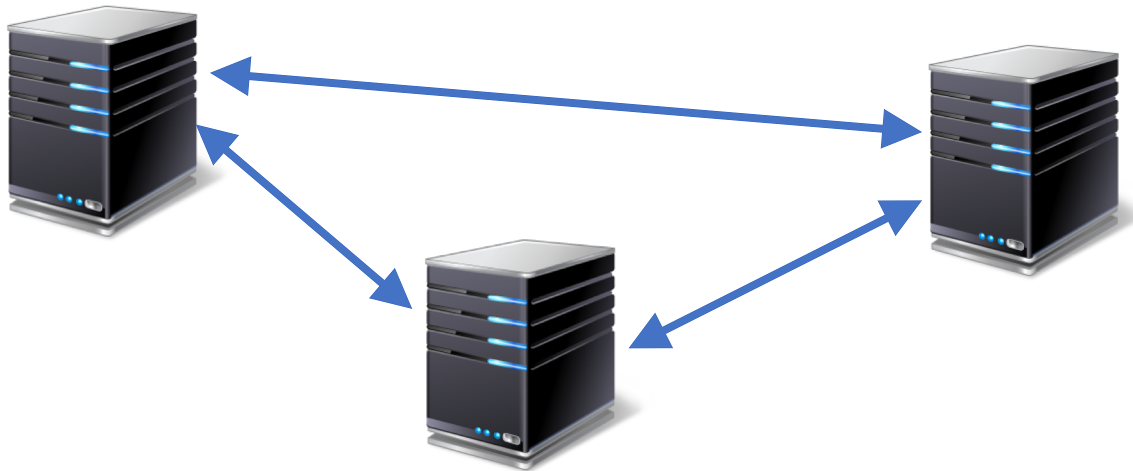
Yue Cheng

Some material taken/derived from:

- Princeton COS-418 materials created by Michael Freedman and Wyatt Lloyd.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.
- Utah CS6450 by Ryan Stutsman.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

Distributed systems: What?

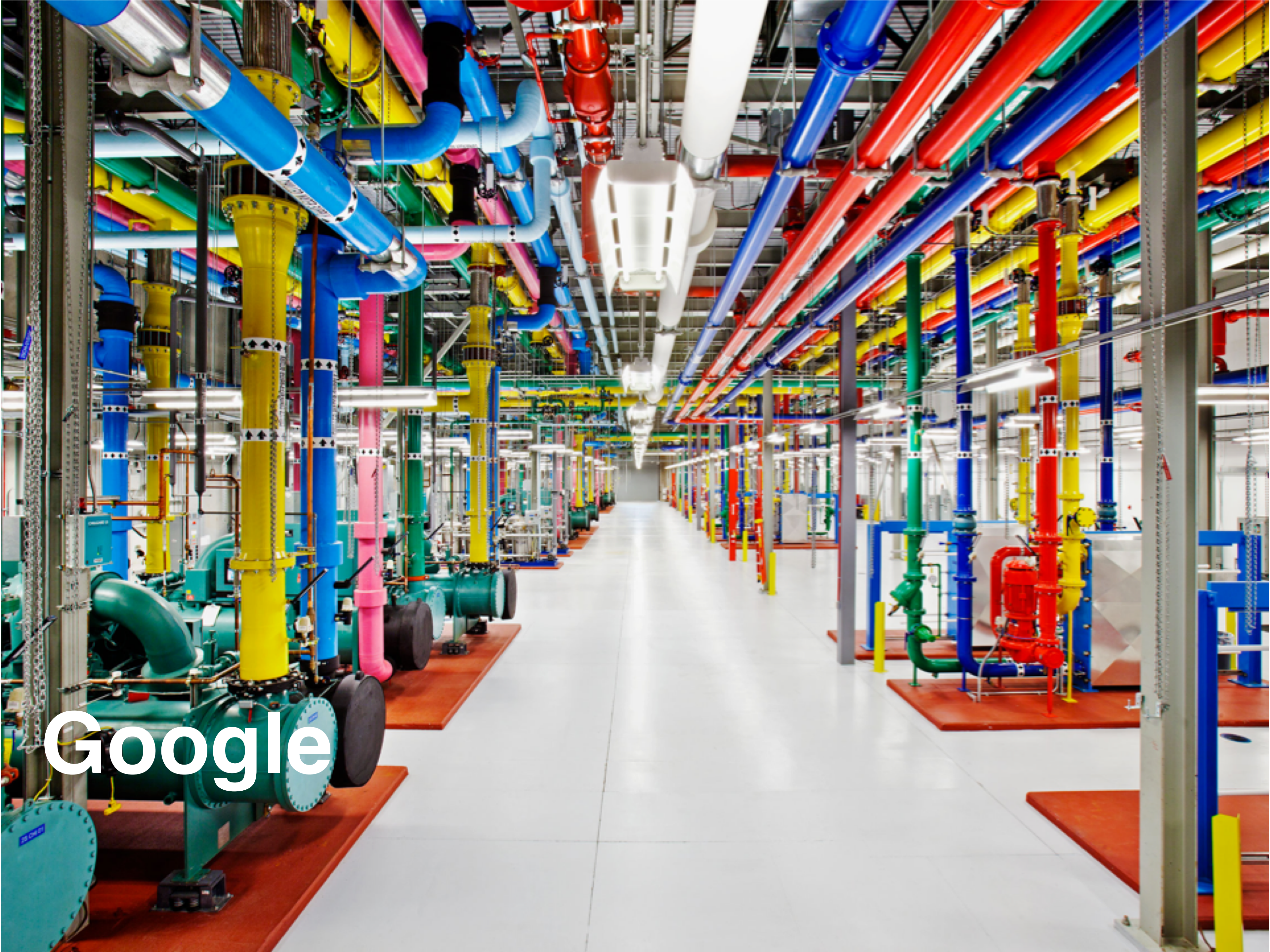


- Multiple cooperating computers
  - Connected by a network
  - Doing something together
- Storage for big websites, MapReduce, etc.
- Lots of critical infrastructure is distributed

Distributed systems: Why?

- Or, why not 1 computer to rule them all?

- To organize physically separate entities
- To tolerate faults via replication
- To scale up throughput via parallel CPUs/mem/disk/net


Google

A Google Datacenter in Hamina, Finland

Facebook

Microsoft's Datacenter to be deployed on the seafloor

Goals of "distributed systems"

- Service with higher-level abstractions/interface
  - E.g., file system, database, key-value store, programming model, …
- High complexity
  - Scalable (scale-out)
  - Reliable (fault-tolerant)
  - Well-defined semantics (consistent)
- Do "heavy lifting" so app developer doesn't need to

Distributed systems: Where?

Distributed systems: Where?

- Web search (e.g., Google, Bing)

- Shopping (e.g,. Amazon, Walmart)

- File sync (e.g., Dropbox, iCloud)

- Social networks (e.g,. Facebook, TikTok)

- Music (e.g., Spotify, Apple Music)

- Ride sharing (e.g., Uber, Lyft)

- Video (e.g., Youtube, Netflix)

Why take this course?

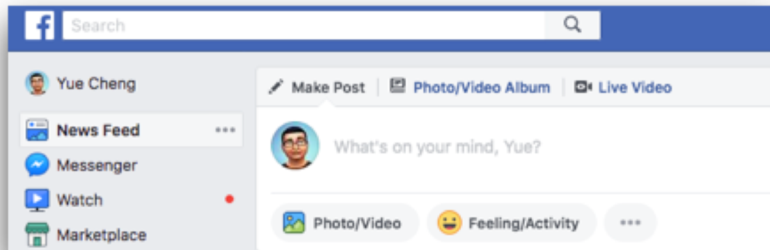
- Interesting – hard problems, powerful solutions
- Used by real systems – driven by the rise of big websites (e.g., Amazon, Facebook)
- Active research area – lots of progress + big unsolved problems
- Hands-on – you'll build serious systems in the labs

Dropbox

- A data sync startup founded back in 2008
- Became popular so quickly
  - Latest number of users: 500+ Million
  - Overall amount of data stored: 500 PB
- Initially stored all data on public clouds (AWS)
- Seriously considered to move data out of AWS
- Cloud vendor lock-in
  - Egress costs
- Now still parts of its data services sit on AWS

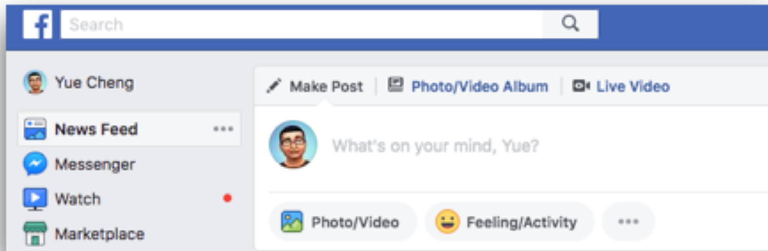
Facebook web queries

1. User clicks on a link



Facebook web queries

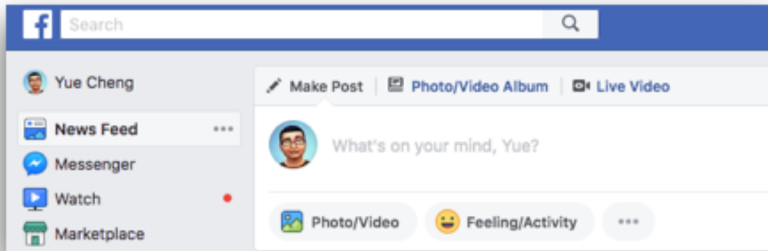
1. User clicks on a link



2. Browser sends requests to FB's frontend web servers

Facebook web queries

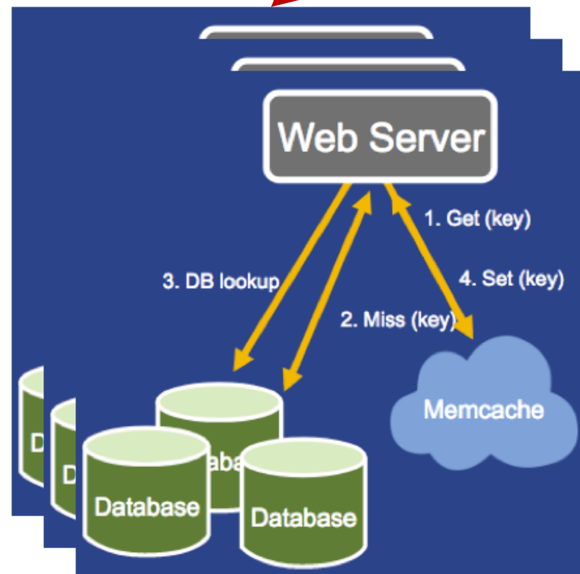
1. User clicks on a link

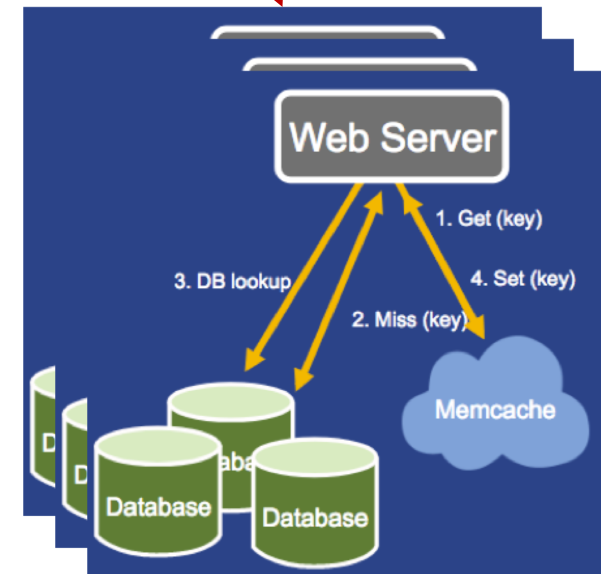


2. Browser sends requests to FB's frontend web servers

3. Web servers convert requests to:

- Database queries
- Memcache queries

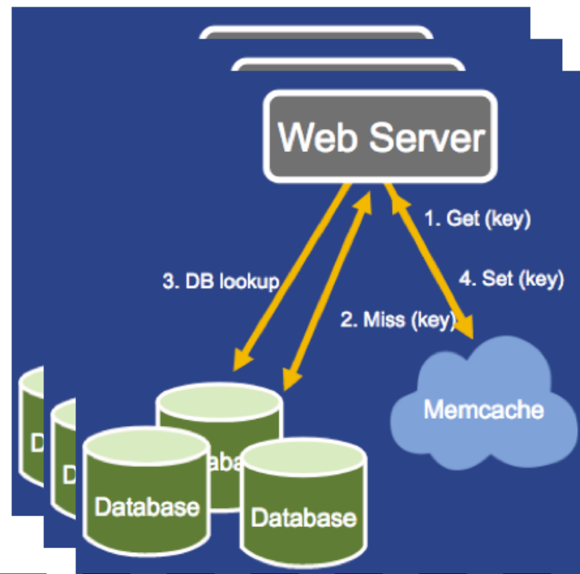


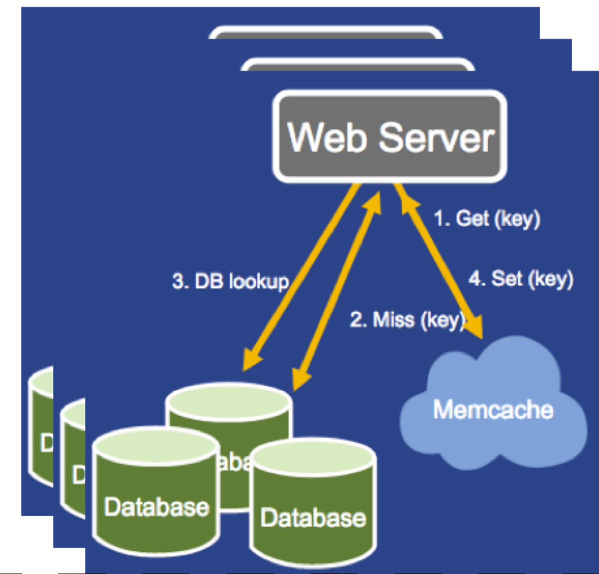


Facebook web queries

Our focus: To learn how backend distributed systems manage physically separate resources

3. Web servers convert requests to:
- Database queries
- Memcache queries





But how do we program this?

With the help of distributed systems

Applications

Web apps | Data processing | Data storage | Emerging apps?

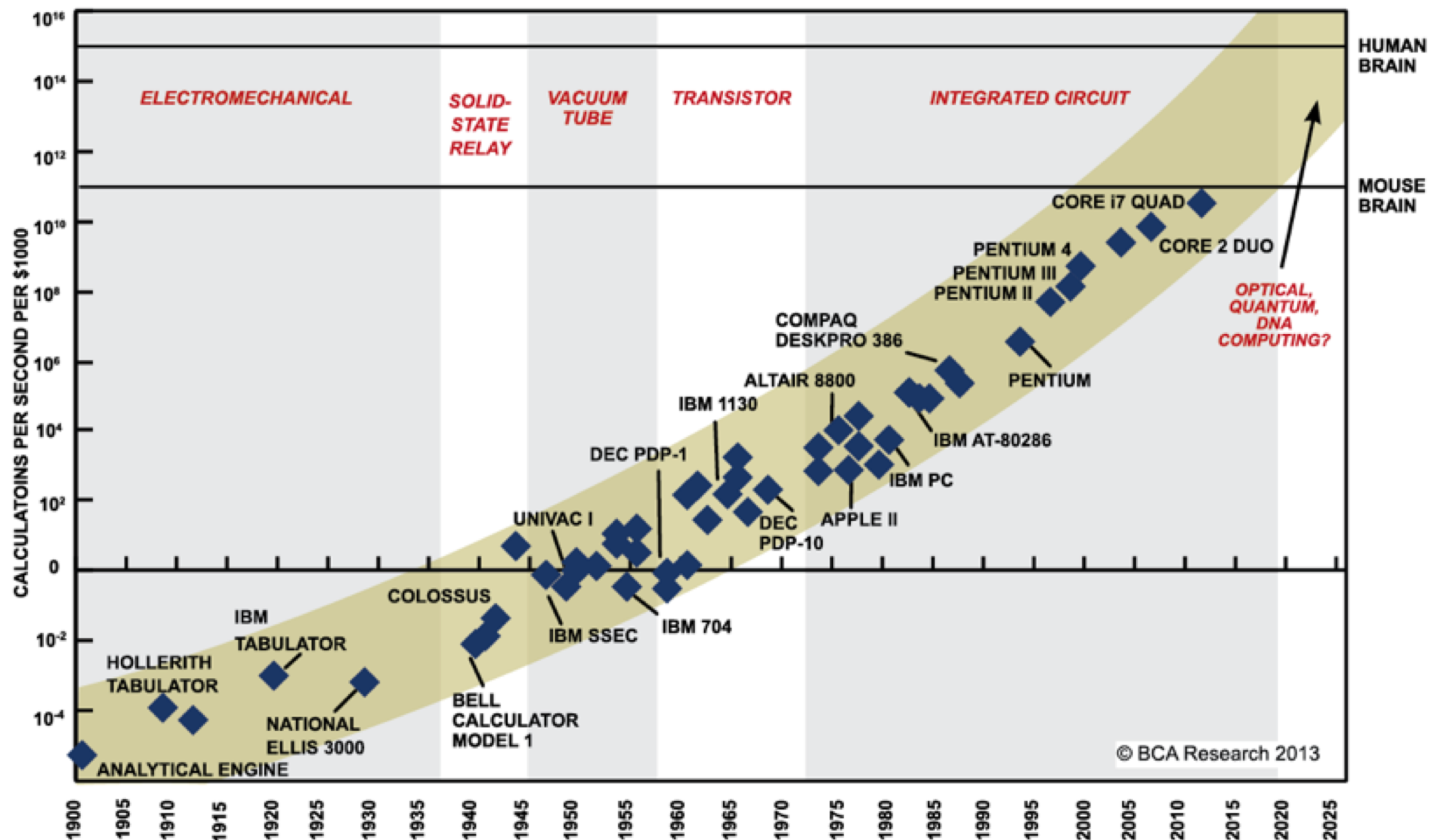
Resource management

Compute resources | Memory resources | Storage resources | Network resources

Datacenter infrastructure

Exciting time in distributed systems research

Moore's law ending → many challenges



Datacenter evolution

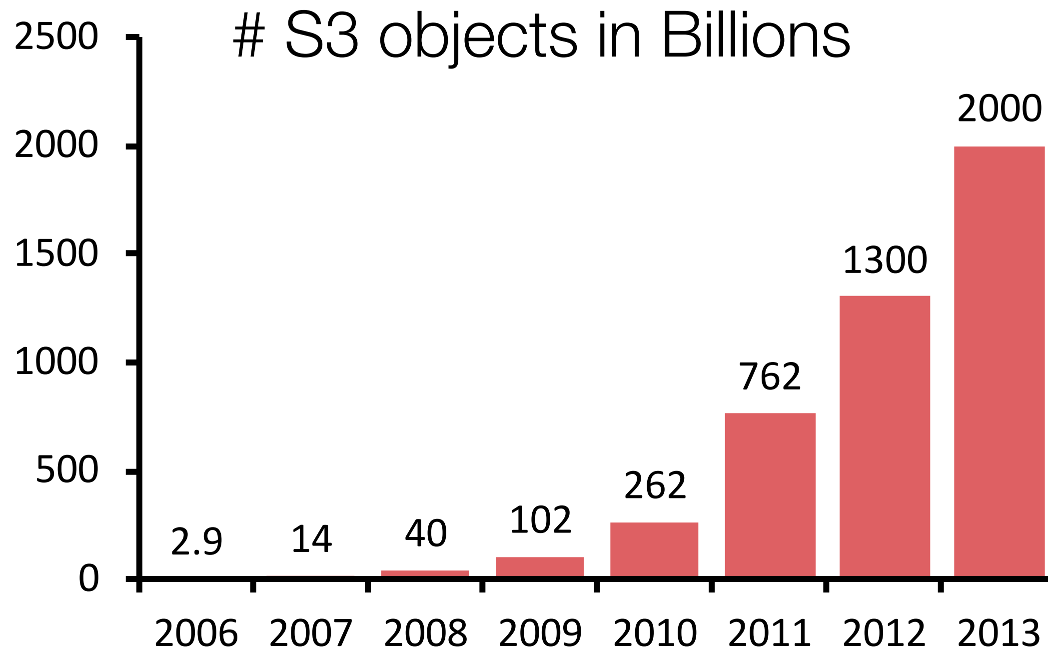
- Facebook's daily logs: 60 TB
- Google web index: 10+ PB

Datacenter evolution

AWS Blog

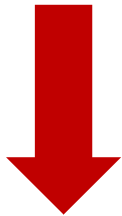
Amazon S3 – Two Trillion Objects, 1.1 Million Requests / Second

by Jeff Barr | on 18 APR 2013 | in Amazon S3 | Permalink | Comments



Increased complexity – Computation

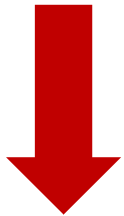
Software



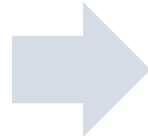
CPU

Increased complexity – Computation

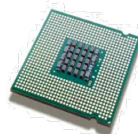
Software



CPU



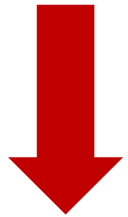
Software

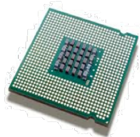


CPU
+
SGX

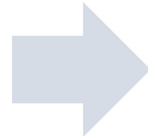
Increased complexity – Computation

Software

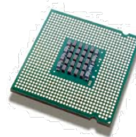




CPU



Software



CPU
+
SGX

GPU

FPGA

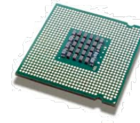
ASIC

Increased complexity – Memory

2015

L1/L2 cache	~1 ns
L3 cache	~10 ns
Main memory	~100 ns / ~80 GB/s / ~100GB
NAND SSD	~100 usec / ~10 GB/s / ~1 TB
Fast HDD	~10 msec / ~100 MB/s / ~10 TB

Increased complexity – Memory

2015		2020	
			
L1/L2 cache	~1 ns	L1/L2 cache	~1 ns
L3 cache	~10 ns	L3 cache	~10 ns
		HBM	~10 ns / ~1TB/s / ~10GB
Main memory	~100 ns / ~80 GB/s / ~100GB	Main memory	~100 ns / ~80 GB/s / ~100GB
		NVM (Intel Optane DC)	~1 usec / ~10GB/s / ~1TB
NAND SSD	~100 usec / ~10 GB/s / ~1 TB	NAND SSD	~100 usec / ~10 GB/s / ~10 TB
Fast HDD	~10 msec / ~100 MB/s / ~10 TB	Fast HDD	~10 msec / ~100 MB/s / ~100

Increased complexity – more and more choices

Basic tier: A0, A1, A2, A3, A4
Optimized Compute : D1, D2, D3, D4, D11, D12, D13
D1v2, D2v2, D3v2, D11v2,…
Latest CPUs: G1, G2, G3, …
Network Optimized: A8, A9
Compute Intensive: A10, A11,…

Microsoft AZURE

t2.nano, t2.micro, t2.small
m4.large, m4.xlarge, m4.2xlarge, m4.4xlarge, m3.medium,
c4.large, c4.xlarge, c4.2xlarge, c3.large, c3.xlarge, c3.4xlarge, r3.large, r3.xlarge, r3.4xlarge, i2.2xlarge, i2.4xlarge, d2.xlarge
d2.2xlarge, d2.4xlarge,…

Amazon EC2

n1-standard-1, ns1-standard-2, ns1-standard-4, ns1-standard-8, ns1-standard-16, ns1highmem-2, ns1-highmem-4, ns1-highmem-8, n1-highcpu-2, n1-highcpu-4, n1-highcpu-8, n1-highcpu-16, n1-highcpu-32, f1-micro, g1-small…

Google Cloud Engine

Increased complexity – more and more requirements

- Scale (physically distributed)
- Latency
- Accuracy
- Cost
- Security
- And a lot more…

The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)

~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packetloss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vips for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for dns

~1000 individual machine failures

~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

* Jeff Dean, LADIS'09

Research results matter: NoSQL

Consistent Hashing and Random Trees:
Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*

David Karger[1] Eric Lehman[1] Tom Leighton[1,2] Matthew Levine[1] Daniel Lewin[1]
Rina Panigrahy[1]

Abstract

We describe a family of caching protocols for distrib-uted networks that can be used to decrease or eliminate the occurrence of hot spots in the network. Our protocols are particularly designed for use with very large networks such as the Internet, where delays caused by hot spots can be severe, and where it is not feasible for every server to have complete information about the current state of the entire network. The protocols are easy to implement using existing network protocols such as TCP/IP, and require very little overhead. The protocols work with local control, make efficient use of existing resources, and scale gracefully as the network grows.

Our caching protocols are based on a special kind of hashing that we call *consistent hashing*. Roughly speaking, a consistent hash function is one which changes minimally as the range of the function changes. Through the development of good consistent hash functions, we are able to develop caching protocols which do not require users to have a current or even consistent view of the network. We believe that consistent hash functions may eventually prove to be useful in other applications such as distributed name servers and/or quorum systems.

1 Introduction

In this paper, we describe caching protocols for distributed networks that can be used to decrease or eliminate the occurrences of "hot spots". *Hot spots* occur any time a large number of clients wish to simultaneously access data from a single server. If the site is not provisioned to deal with all of these clients simultaneously, service may be degraded or lost.

it was originally configured to handle. In fact, a site may receive so many requests that it becomes "swamped," which typically renders it unusable. Besides making the one site inaccessible, heavy traffic destined to one location can congest the network near it, interfering with traffic at nearby sites.

As use of the Web has increased, so has the occurrence and impact of hot spots. Recent famous examples of hot spots on the Web include the JPL site after the Shoemaker-Levy 9 comet struck Jupiter, an IBM site during the Deep Blue-Kasparov chess tournament, and several political sites on the night of the election. In some of these cases, users were denied access to a site for hours or even days. Other examples include sites identified as "Web-site-of-the-day" and sites that provide new versions of popular software.

Our work was originally motivated by the problem of hot spots on the World Wide Web. We believe the tools we develop may be relevant to many client-server models, because centralized servers on the Internet such as Domain Name servers, Multicast servers, and Content Label servers are also susceptible to hot spots.

1.1 Past Work

Several approaches to overcoming the hot spots have been proposed. Most use some kind of replication strategy to store copies of hot pages throughout the Internet; this spreads the work of serving a hot page across several servers. In one approach, already in wide use, several clients share a *proxy cache*. All user requests are forwarded through the proxy, which tries to keep copies of frequently requested pages. It tries to satisfy requests with a cached copy; failing this, it forwards the request to the home server. The dilemma in this scheme is that there is more benefit if more users share the

Research results matter: NoSQL

Consistent Hashing and Random
Distributed Caching Protocols for Relieving Hot Spot

David Karger[1] Eric Lehman[1] Tom Leighton[1,2] Matt
Rina Panigrahy[1]

Abstract

We describe a family of caching protocols for distrib-uted networks that can be used to decrease or eliminate the occurrence of hot spots in the network. Our protocols are particularly designed for use with very large networks such as the Internet, where delays caused by hot spots can be severe, and where it is not feasible for every server to have complete information about the current state of the entire network. The protocols are easy to implement using existing network protocols such as TCP/IP, and require very little overhead. The protocols work with local control, make efficient use of existing resources, and scale gracefully as the network grows.

Our caching protocols are based on a special kind of hashing that we call *consistent hashing*. Roughly speaking, a consistent hash function is one which changes minimally as the range of the function changes. Through the development of good consistent hash functions, we are able to develop caching protocols which do not require users to have a current or even consistent view of the network. We believe that consistent hash functions may eventually prove to be useful in other applications such as distributed name servers and/or quorum systems.

1 Introduction

In this paper, we describe caching protocols for distributed networks that can be used to decrease or eliminate the occurrences of "hot spots". *Hot spots* occur any time a large number of clients wish to simultaneously access data from a single server. If the site is not provisioned to deal with all of these clients simultaneously, service may be degraded or lost.

it was originally c
many requests that
it unusable. Beside
destined to one loc
with traffic at near
As use of the
impact of hot spot
Web include the J
Jupiter, an IBM s
nament, and sever
some of these case
even days. Other e
the-day" and sites
Our work was
on the World Wid
relevant to many c
on the Internet su
and Content Label

1.1 Past Work

Several approache
posed. Most use s
hot pages through
a hot page across s
use, several client
warded through th
requested pages. I
ing this, it forward
in this scheme is t

Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System

Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer and Carl H. Hauser

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304 U.S.A.

Abstract

Bayou is a replicated, weakly consistent storage system designed for a mobile computing environment that includes portable machines with less than ideal network connectivity. To maximize availability, users can read and write any accessible replica. Bayou's design has focused on supporting application-specific mechanisms to detect and resolve the update conflicts that naturally arise in such a system, ensuring that replicas move towards eventual consistency, and defining a protocol by which the resolution of update conflicts stabilizes. It includes novel methods for conflict detection, called dependency checks, and per-write conflict resolution based on client-provided merge procedures. To guarantee eventual consistency, Bayou servers must be able to rollback the effects of previously executed writes and redo them according to a global serialization order. Furthermore, Bayou permits clients to observe the results of all writes received by a server, including tentative writes whose conflicts have not been ultimately resolved. This paper presents the motivation for and design of these mechanisms and describes the experiences gained with an initial implementation of the system.

1. Introduction

The Bayou storage system provides an infrastructure for collaborative applications that manages the conflicts introduced by concurrent activity while relying only on the weak connectivity available for mobile computing. The advent of mobile computers, in the form of laptops and personal digital assistants (PDAs) enables the use of computational facilities away from the usual work setting of users. However, mobile computers do not enjoy the connectivity afforded by local area networks or the telephone sys-

"connectedness" are possible. Groups of computers may be partitioned away from the rest of the system yet remain connected to each other. Supporting disconnected workgroups is a central goal of the Bayou system. By relying only on pair-wise communication in the normal mode of operation, the Bayou design copes with arbitrary network connectivity.

A weak connectivity networking model can be accommodated only with weakly consistent, replicated data. Replication is required since a single storage site may not be reachable from mobile clients or within disconnected workgroups. Weak consistency is desired since any replication scheme providing one copy serializability [6], such as requiring clients to access a quorum of replicas or to acquire exclusive locks on data that they wish to update, yields unacceptably low write availability in partitioned networks [5]. For these reasons, Bayou adopts a model in which clients can read and write to any replica without the need for explicit coordination with other replicas. Every computer eventually receives updates from every other, either directly or indirectly, through a chain of pair-wise interactions.

Unlike many previous systems [12, 27], our goal in designing the Bayou system was *not* to provide transparent replicated data support for existing file system and database applications. We believe that applications must be aware that they may read weakly consistent data and also that their write operations may conflict with those of other users and applications. Moreover, applications must be involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application.

To this end, Bayou provides system support for application-specific conflict detection and resolution. Previous systems, such as Locus [30] and Coda [17], have proven the value of semantic conflict detection and resolution for file directories, and several systems are exploring conflict resolution for file and database contents [8, 18, 26]. Bayou's mechanisms extend this work by letting

Research results matter: NoSQL

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of

Research results matter: Paxos

The Part-Time Parliament

Leslie Lamport

This article appeared in *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133-169. Minor corrections were made on 29 August 2000.

Research results matter: Paxos

The Part-Ti

Leslie

This article appeared in *ACM*
tems 16, 2 (May 1998), 133-1
on 29 August 2000.

Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Oki
Barbara H. Liskov

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

Abstract

One of the potential benefits of distributed systems is their use in providing highly-available services that are likely to be usable when needed. Availability is achieved through replication. By having more than one copy of information, a service continues to be usable even when some copies are inaccessible, for example, because of a crash of the computer where a copy was stored. This paper presents a new replication algorithm that has desirable performance properties. Our approach is based on the primary copy technique. Computations run at a primary, which notifies its backups of what it has done. If the primary crashes, the backups are reorganized, and one of the backups becomes the new primary. Our method works in a general network with both node crashes and partitions. Replication causes little delay in user computations and little information is lost in a reorganization; we use a special kind of timestamp called a viewstamp to detect lost information.

1 Introduction

One of the potential benefits of distributed systems is their use in providing highly-available services, that is, services that are likely to be up and accessible when needed. Availability is essential to many computer-based services; for example, in airline reservation systems the failure of a single computer can prevent ticket sales for a considerable time, causing a loss of revenue and passenger goodwill.

Availability is achieved through replication. By having more than one copy of important information, the service continues to be usable even when some copies are inaccessible, for example, because of a crash of the computer where a copy was stored. Various replication

Our algorithm runs on a system consisting of nodes connected by a communication network. Nodes are independent computers that communicate with each other only by sending messages over the network. Although both nodes and the network may fail, we assume these failures are not byzantine [24]. Nodes can crash, but we assume they are failstop processors [34]. The network may lose, delay, and duplicate messages, or deliver messages out of order. Link failures may cause the network to partition into subnetworks that are unable to communicate with each other. We assume that nodes eventually recover from crashes and partitions are eventually repaired.

Our replication method assumes a model of computation in which a distributed program consists of *modules*, each of which resides at a single node of the network. Each module contains within it both data objects and code that manipulates the objects; modules can recover from crashes with some of their state intact. No other module can access the data objects of another module directly. Instead, each module provides procedures that can be used to access its objects; modules communicate by means of *remote procedure calls*. Modules that make calls are called *clients*; the called module is a *server*.

Modules are the unit of replication in our method. Ideally, programmers would write programs without concern for availability in some (distributed) programming language that supports our model of computation. The language implementation then uses our technique to replicate individual modules automatically; the resulting programs are highly available.

We assume that computations run as atomic transactions [14]. Our method guarantees the *one-copy serializability* correctness

Research results matter: Paxos

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

Abstract

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to accommodate the differences.

example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

Before Chubby was deployed, most distributed systems at Google used *ad hoc* methods for primary election (when work could be duplicated without harm), or required operator intervention (when correctness was essential). In the former case, Chubby allowed a small saving in computing effort. In the latter case, it achieved a

Research results matter: MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to paral-

Research results matter: MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to paral-

Course organization

Big picture course goals

- Learn about some of the most influential works in distributed systems
- Learn how to read distributed systems papers
- Learn how to manage writing highly concurrent and non-deterministic code
  - In my opinion, much harder than “just” parallel programming
- Get a sense of how massive scale systems “fit” together

Course format: Lectures

- Some lecture + some discussion
  - Slides available on course website (night before)
  - First five weeks: lectures about fundamentals of distributed systems
    - Some from textbook, but most from research papers
  - After midterm (after Spring Recess): paper discussion
    - 2 papers per lecture
    - I introduce necessary background knowledge
    - Each paper will be discussed as a case study
    - Review forms to help drive focus of lecture discussion

- Key: read required papers before lectures

Course format: Review forms

- Goal: read and be prepared for discussion
- Review form will be posted on Piazza few days before lecture
  - You need to fill out review form by 9am same day of lecture
  - Review form will cover required reading with a strong focus on stimulating a fruitful discussion

- No late submission will be accepted
- Contact instructor for exceptions in severe circumstances only

How to read a paper: GFS Example

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures

How to read a paper: Summary

- 1st pass: Read abstract, introduction, section headings, conclusion
- 2nd pass: Read all sections, make notes

- Some key points (examples):
  - What is the problem being considered?
  - What are the main contributions? How do they compare to prior work?
  - What workloads, setups were considered in the evaluation?
  - What parts of the claims are adequately backed up?
  - …

Course format: Discussion

- Your participation is very important
  - As an indicator of how well you've prepared

- Paper review examples
  - One or two sentence summary of the paper
  - Description of the problem or assumptions made
  - Comparison to other papers discussed in class?
  - One flaw or thing that can be improved?
  - Experimental setup and what the results mean?
  - …

Course format: Labs

- Three individual Labs (building a simple & flexible serverless framework using Go)
  - **Lab 0:** Intro to Go
  - **Lab 1:** Serverless and RPC
  - **Lab 2:** MapReduce atop Lab 1
    - Phase 1: Implementing Map and Reduce Lambda plugins
    - Phase 2: Distributing and scheduling Lambda plugin functions
    - Phase 3: Adding fault tolerance support
  - Lab 3 (optional – open-ended): Build a microservice app (e.g., Hotel reservation) by expanding on Lab 1

- Get started early on labs, especially Lab 1 and 2
- Require comfort with (Go) concurrency that takes awhile to acquire

Course project

- Goal: Explore new research ideas or significant implementation in the area of distributed systems
  - Define the problem
  - Execute the research
  - Write up and present your research

- Ideally, best projects will have the potential to:
  - Become workshop/conference paper (research-oriented)
  - Contribute towards open source community (implementation-oriented)

Course project steps

- I will distribute a list of project ideas (Week of Feb 26)
  - You can either choose one or come up with your own
- (Not mandatory) Pick your partner: a team of at most 2 students
- Milestones (tentative)
  - Project proposal due Friday, March 20
  - Project checkpoint due Friday, April 17
  - Final presentation on April 29
  - Final project report & src due May 6

Calendar

- Readings, assignments, due dates
- Less concrete further out; don't get too far ahead

CS 675: Distributed Systems
George Mason University

- Home
- Course Information
- Course Schedule
- Final Project
- Announcements
- GitLab

CS 675 Distributed Systems (Spring 2020)

Course Schedule

The course schedule is tentative and subject to change.

Date	Topic	Required reading/assignment	Optional reading
Jan 22	**Lec 1:** Introduction, Go systems programming	**Lab 0** out [syntax] [go_systems_programming]	XXX
Jan 29	**Lec 2:** Remote procedure call, RPCs in Go	**Lab 0 due** **Lab 1** out	XXX
Feb 5	**Lec 3:** MapReduce, concurrency in Go	MapReduce	Google File System
Feb 12	**Lec 4:** Time & clocks, primary-backup	**Lab 1 due** **Lab 2** out XXX	XXX
Feb 19	**Lec 5:** Distributed consensus (Paxos + Raft)	Paxos, Raft	Paxos made live, Chain replication
Feb 26	**Traveling to FAST** (NO CLASS) Hack Day!	Project ideas released: Pick your project	

Course staff

- Mainly me
  - Yue Cheng: yuecheng@gmu.edu

Getting help

- Office hours
  - Wednesday 2 – 3pm, Engineering 5324

- Piazza
  - Good place to ask and answer questions
    - About labs
    - About project
    - About material from lecture
    - About paper reviews and discussion
  - No anonymous posts or questions
  - Can send private messages to instructor

Reading materials

- Slides/lecture notes

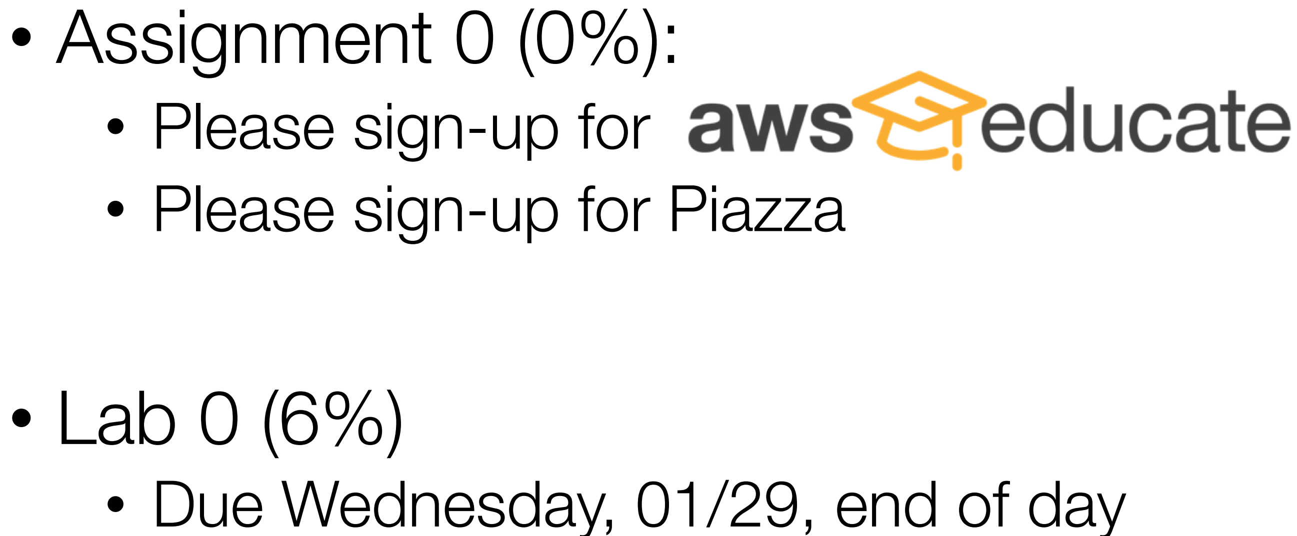
- Papers (required or optional) also serve as reference for many topics that aren't covered directly by a text

- (If needed) "Distributed Systems" v3.01 by van Steen and Tanenbaum will supply optional alternate explanations; will try to keep alternate readings up-to-date

Grading (tentative)

- Three labs (30% total)
  - THREE free late days, must tell me (instructor) using them
  - Otherwise, late turnings are graded with 15% deducted each day; no credit after 3 days
- Midterm exam (10%)
- Paper review forms (10%)
- Class participation (5%)
- Final exam (10%)
- Final projects (35%)

Assignment 0 and Lab 0

- Assignment 0 (0%):
  - Please sign-up for 

  - Please sign-up for Piazza

- Lab 0 (6%)
  - Due Wednesday, 01/29, end of day

Your turn…