Introduction

CS 475: Concurrent & Distributed Systems (Fall 2021)

Lecture 1

Yue Cheng

Course staff

- Instructor
  - Dr. Yue Cheng (web: cs.gmu.edu/~yuecheng)
  - Email: yuecheng@gmu.edu
  - Research interests: Distributed systems, cloud computing, operating systems

Course staff

- GTA:
  - Rui Yang: ryang22@gmu.edu

Getting help

- My office hours
  - T 10:30 am – 12 pm, 5324 Engineering
  - By appointment
- Rui's office hours
  - MW 9 am – 11 am
  - Location: TBD

- Ed (trial): https://edstem.org
  - An alternative (maybe good?) place to ask and answer questions
    - About labs
    - About material from lectures
  - No anonymous posts or questions
  - Can send private messages to instructor/GTA

Course organization

Big picture course goals

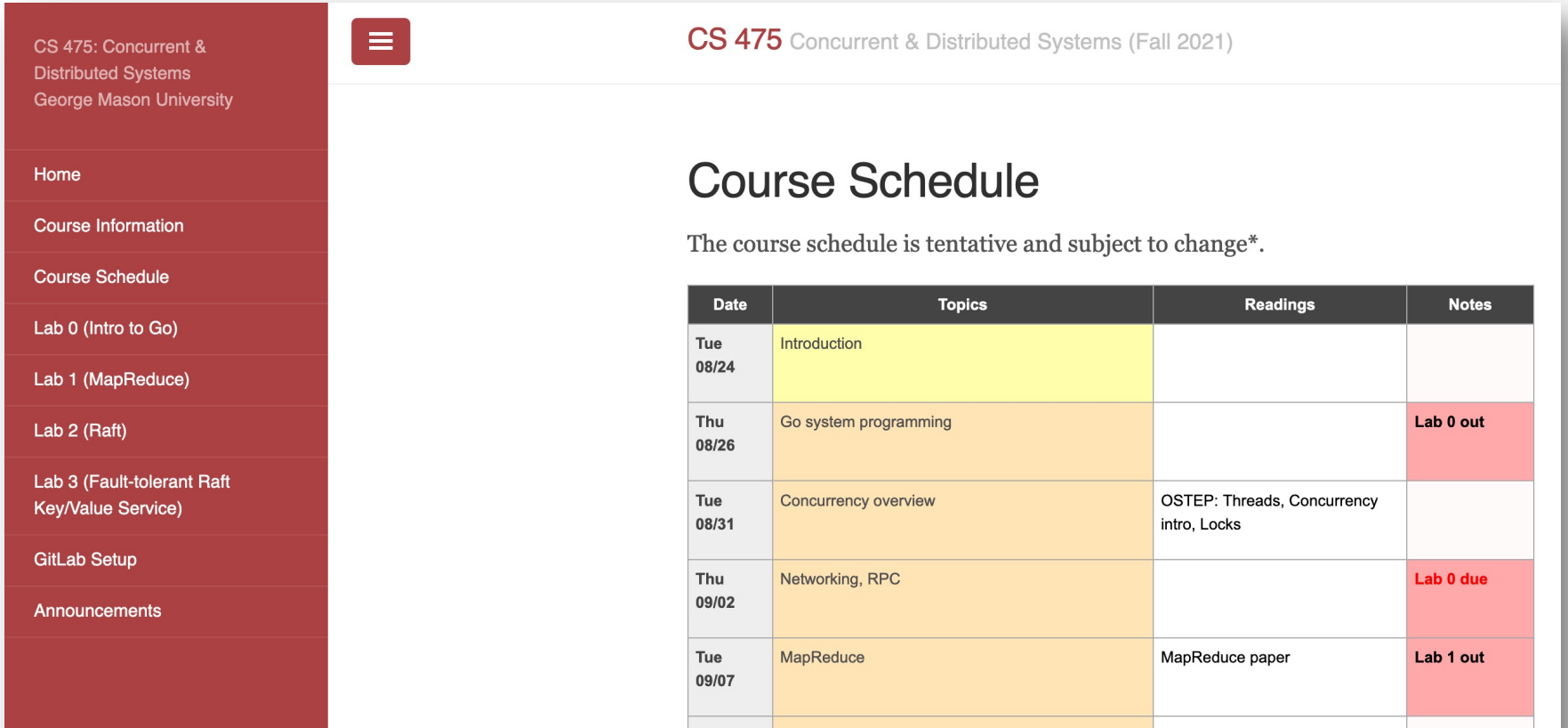
- Learn about some of the most influential works in distributed systems
- Learn how to approach, discuss, and communicate about difficult & technical subject matter
- Get a sense of how massive scale systems “fit” together
- Learn how to manage writing highly concurrent and non-deterministic code
  -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

Lectures

- (Review) + lecture + (lab tutorial)
- Slides available on course website (night before)
- First five weeks: Fundamentals of concurrent & distributed systems
- Week 5-8: Fault tolerance and consensus
- After midterm: Week 9-11: Consistency, scalability, transactions
- Week 11-15: Datacenter computing

Calendar (tentative)

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



CS 475: Concurrent & Distributed Systems
George Mason University

Home
Course Information
Course Schedule
Lab 0 (Intro to Go)
Lab 1 (MapReduce)
Lab 2 (Raft)
Lab 3 (Fault-tolerant Raft Key/Value Service)
GitLab Setup
Announcements

CS 475 Concurrent & Distributed Systems (Fall 2021)

Course Schedule

The course schedule is tentative and subject to change*.

Date	Topics	Readings	Notes
Tue 08/24	Introduction		
Thu 08/26	Go system programming		Lab 0 out
Tue 08/31	Concurrency overview	OSTEP: Threads, Concurrency intro, Locks	
Thu 09/02	Networking, RPC		Lab 0 due
Tue 09/07	MapReduce	MapReduce paper	Lab 1 out

Textbooks?

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

- Slides/lecture notes

- "Distributed Systems 3rd edition" by van Steen and Tenenbaum will supply optional alternate explanations

Programming labs

- Four lab assignments (Go) – all individually
  - **Lab 0:** Intro to Go
  - **Lab 1:** MapReduce
  - **Lab 2:** Raft
  - **Lab 3:** Raft fault-tolerant KV service (built on Lab 2)

- Require comfort with (Go) concurrency that takes awhile to acquire

- Your labs will be autograded; you can resubmit and view your score in real-time

Next 13 Weeks...

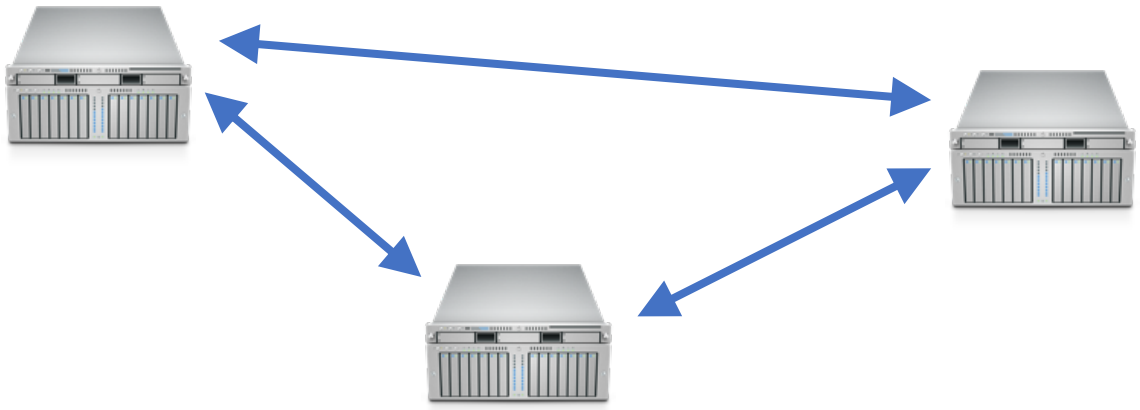
By the end of the semester...

- You will have built some sophisticated, functional, distributed systems using Go
- Get started early on labs
  - Each lab component is mostly out for 2-3 weeks: it will take 2 weeks to do the assignment
  - If you start the day before, there won't be enough hours in the day to complete the lab
- Labs are graded on functionality but not performance
  - Bad designs, however, may significantly affect the performance and thus force autograder to timeout

Grading

- Labs (60% total)
  - Late turnings are graded with 10% deducted each day; **no credit after 3 days**
- Quizzes and in-class activities (5%)
- Midterm exam (15%)
- Final exam (20%)

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

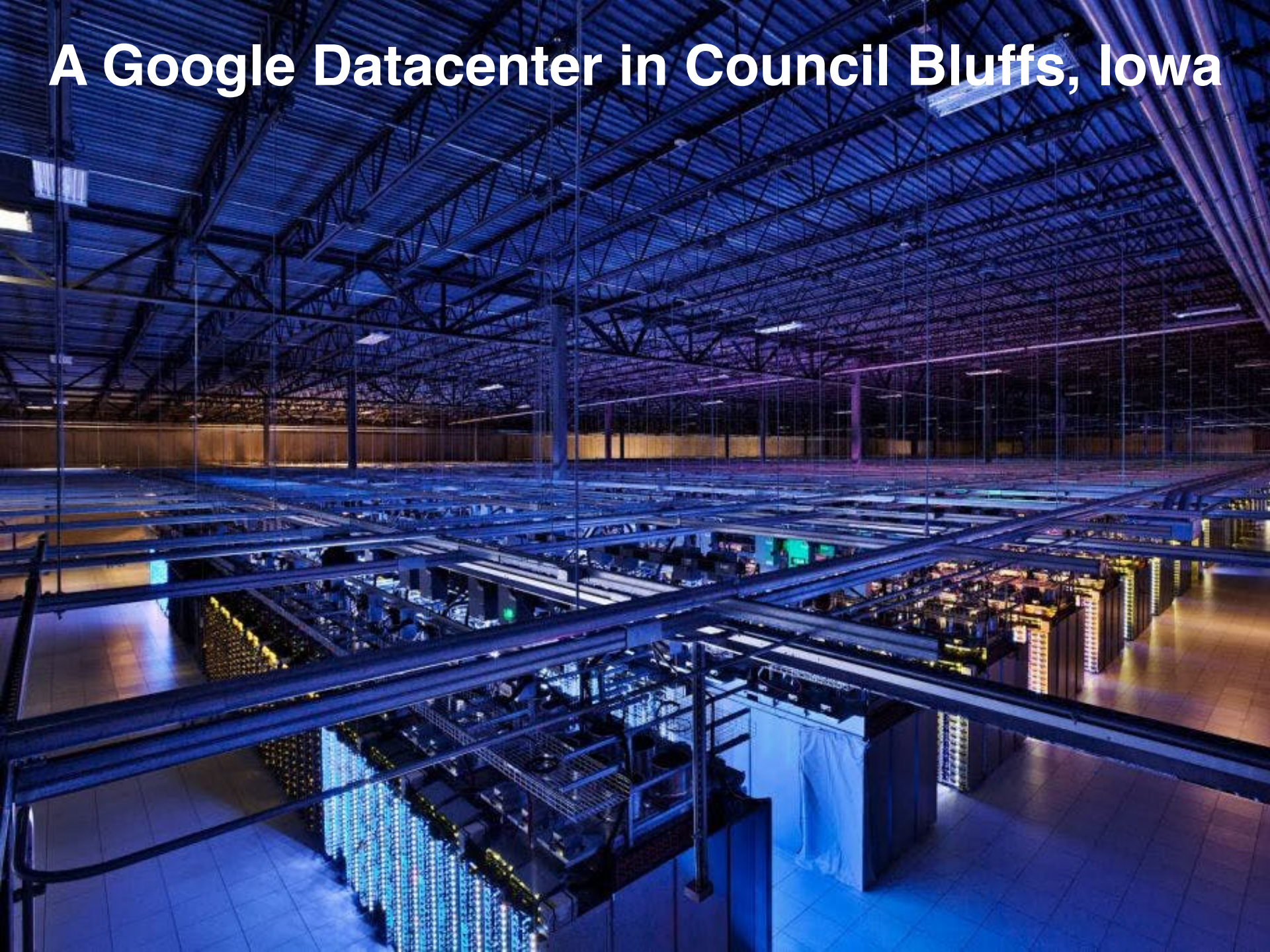
Distributed systems: Why?

- 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 Council Bluffs, Iowa



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?

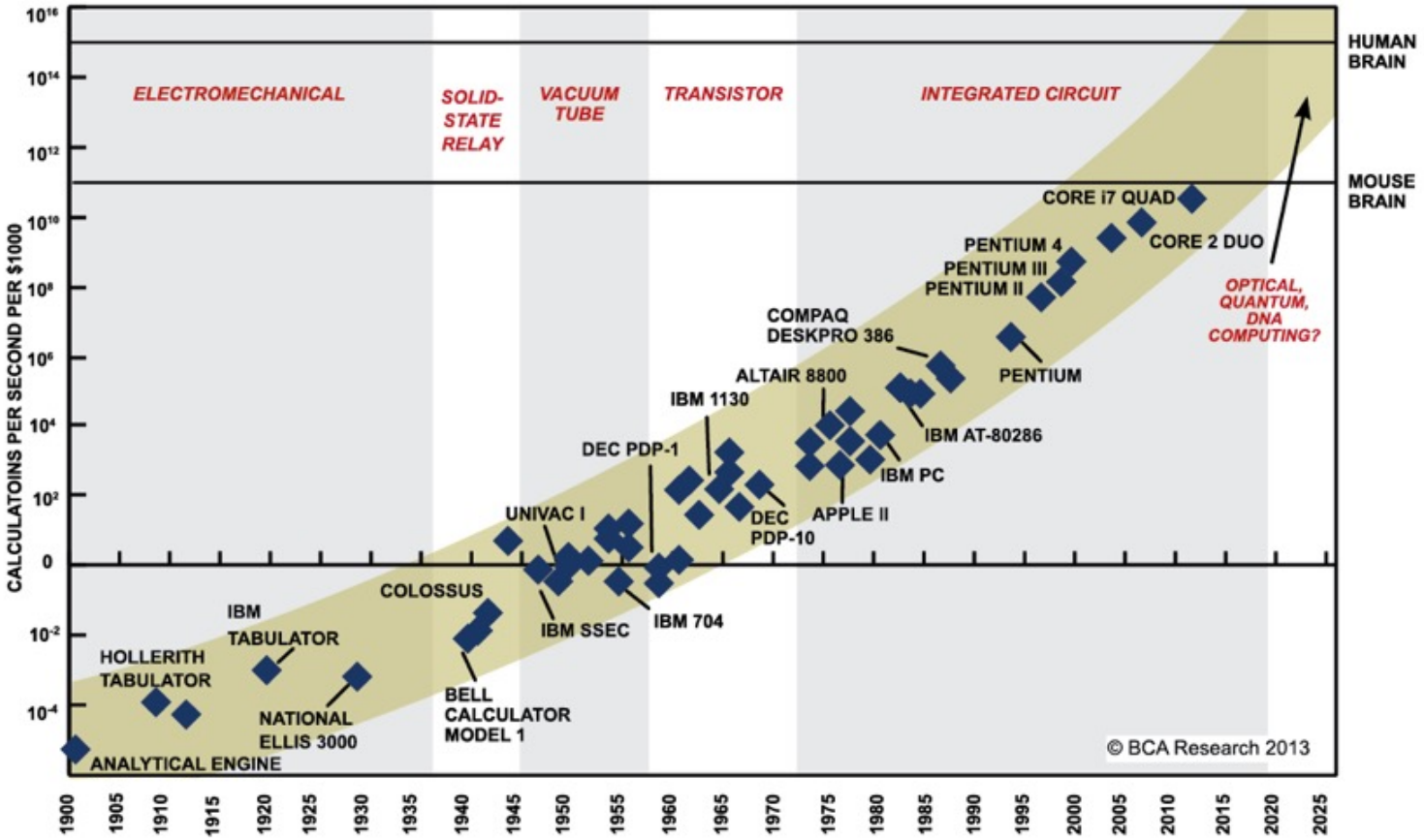
- Web search (e.g., Google, Bing)
- Shopping (e.g,. Amazon, Walmart)
- File sync (e.g., Dropbox, iCloud)
- Teleconferencing (e.g,. Zoom, WebEx)
- Music (e.g., Spotify, Apple Music)
- Ride sharing (e.g., Uber, Lyft)
- Video (e.g., Youtube, Netflix)

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 programming assignments (labs)

Exciting time in distributed systems research

Moore's law ending → many challenges



SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Datacenter evolution

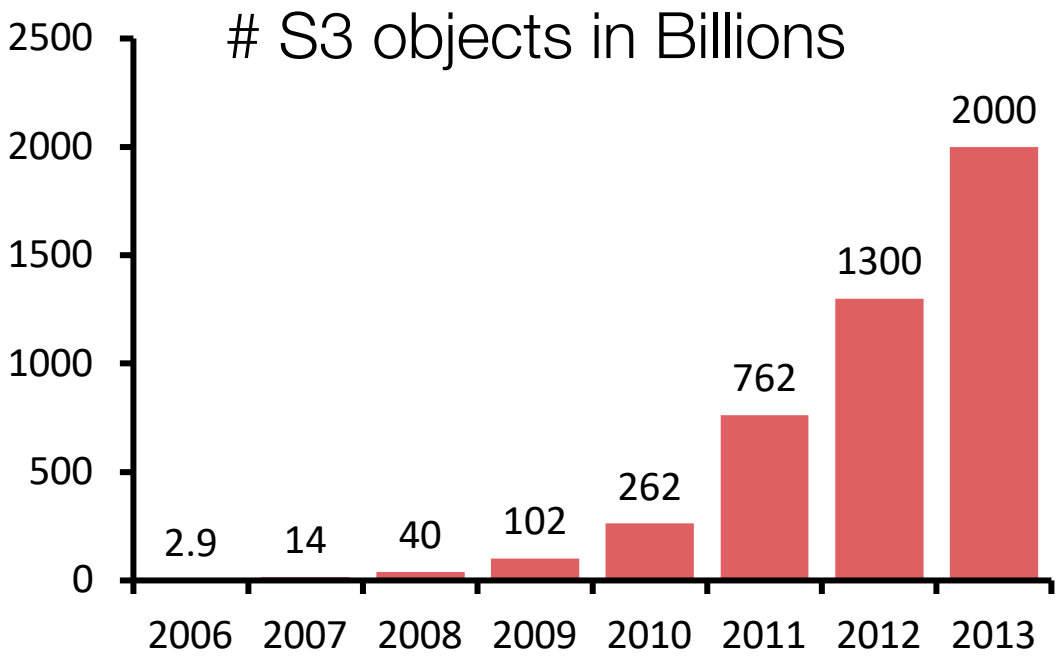
- Facebook's daily logs: 60 TB
- Google web index: 10+ PB

Datacenter evolution

AWS Blog

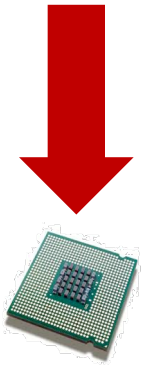
Amazon S3 – Two Trillion Objects, 1.1 Million Requests / Second

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



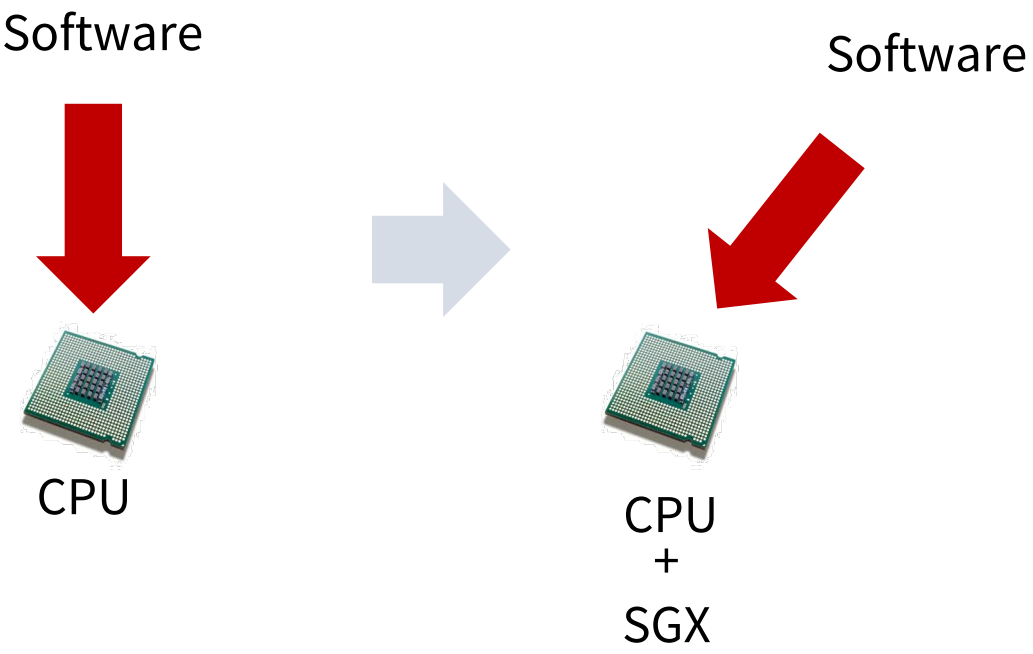
Increased complexity – Computation

Software

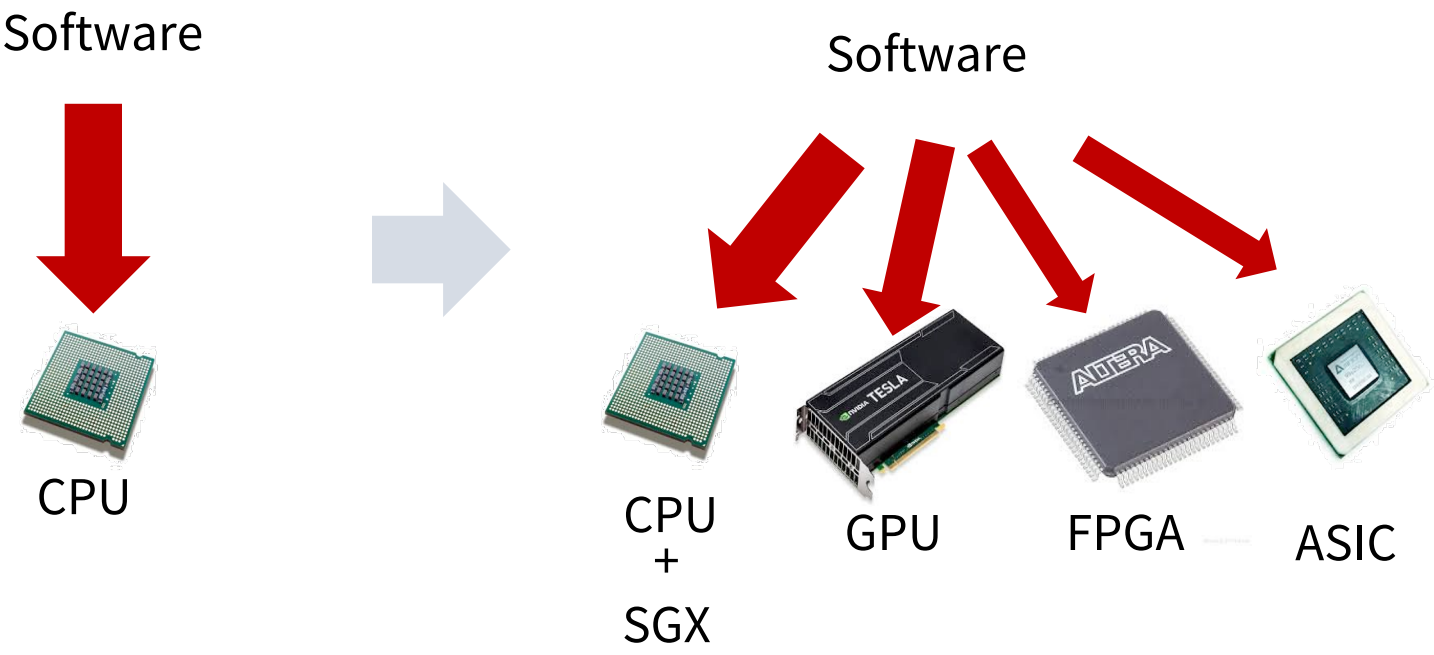


CPU

Increased complexity – Computation



Increased complexity – Computation



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

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

2020

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

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 (highly concurrent, physically distributed)
- Latency
- 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

Cows responsible for short outages to Google fiber network

Abner Li - May. 21st 2020 2:20 pm PT @technacity

...al Hardware

...ecover)

...denly d... ...ack)

...00 machines...

...nes down over 2-d...

...ly disappear, 1-6 hours t...

...50% packetloss)

...~30-minute random connectivity los...

...xternal vips for a couple minutes)

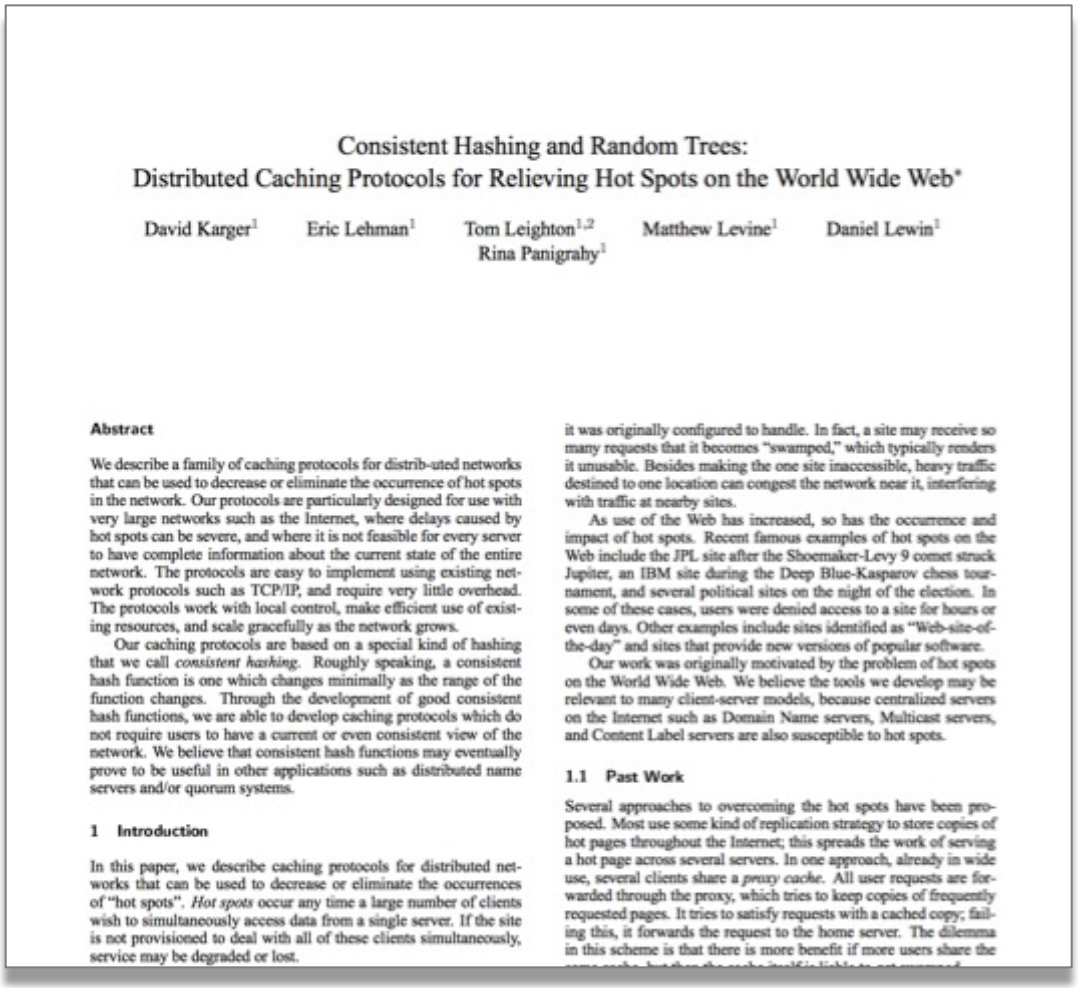
...ll traffic for an hour)

...achines, flaky machines, etc.

...ad 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

Consisten
Distributed Caching Protocol

David Karger[1] Eric Lehman[1]

Abstract

We describe a family of caching protocols for distrib-
that can be used to decrease or eliminate the occurrenc
in the network. Our protocols are particularly designe
very large networks such as the Internet, where dela
hot spots can be severe, and where it is not feasible fo
to have complete information about the current state
network. The protocols are easy to implement using
work protocols such as TCP/IP, and require very lit
The protocols work with local control, make efficient
ing resources, and scale gracefully as the network gro

Our caching protocols are based on a special kin
that we call *consistent hashing*. Roughly speaking,
hash function is one which changes minimally as the
function changes. Through the development of go
hash functions, we are able to develop caching protoc
not require users to have a current or even consisten
network. We believe that consistent hash functions m
prove to be useful in other applications such as dist
servers and/or quorum systems.

1 Introduction

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

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

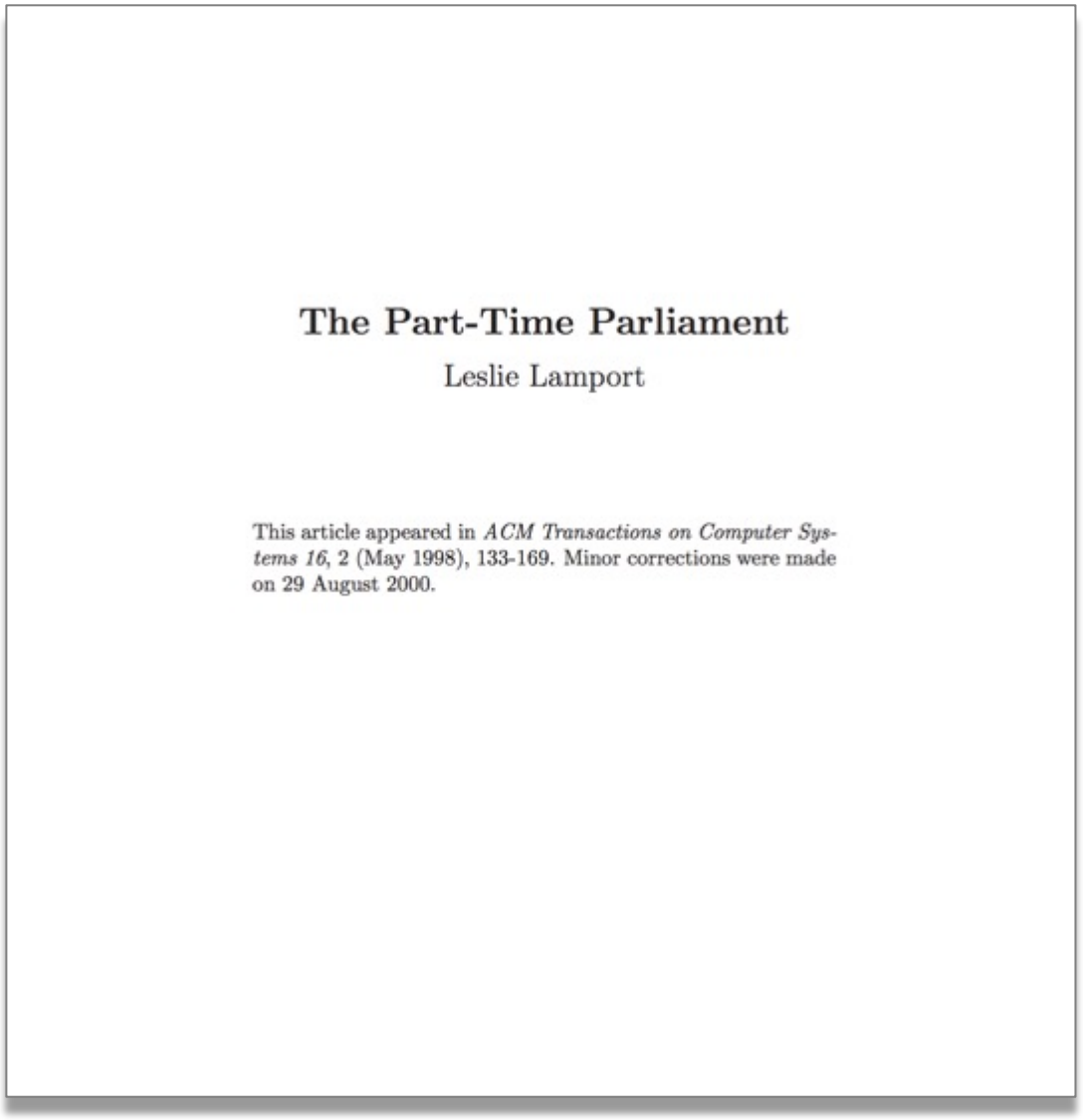
Algorithms, Management, Measurement, Performance, Design,

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

Research results matter: Consensus



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: Consensus

Th

This articl
tems 16, 2
on 29 Aug

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.

1 Introduction

This paper describes a *lock service* called Chubby. It is intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small ma-

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 significant improvement in availability in systems that no longer required human intervention on failure.

Readers familiar with distributed computing will recognize the election of a primary among peers as an instance of the *distributed consensus* problem, and realize

Y. Cheng

Research results matter: Consensus

The Chubby lock service for loosely-coupled distributed systems

In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.

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

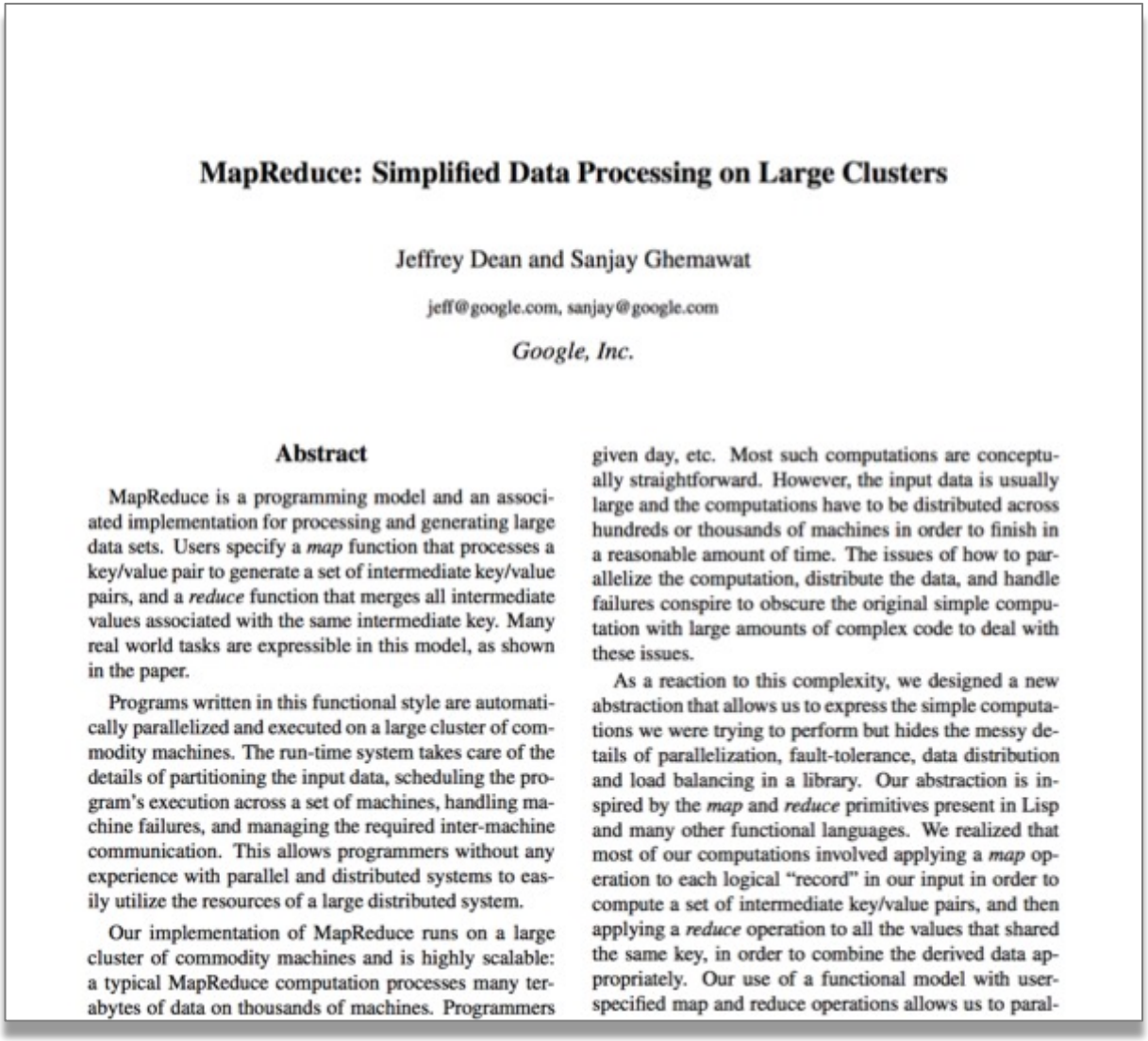
MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with 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-

hadoop

Spark

kafka

Flink

S4 distributed stream computing platform

STORM

Assignment 0

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

- Next class: Go systems programming tutorial