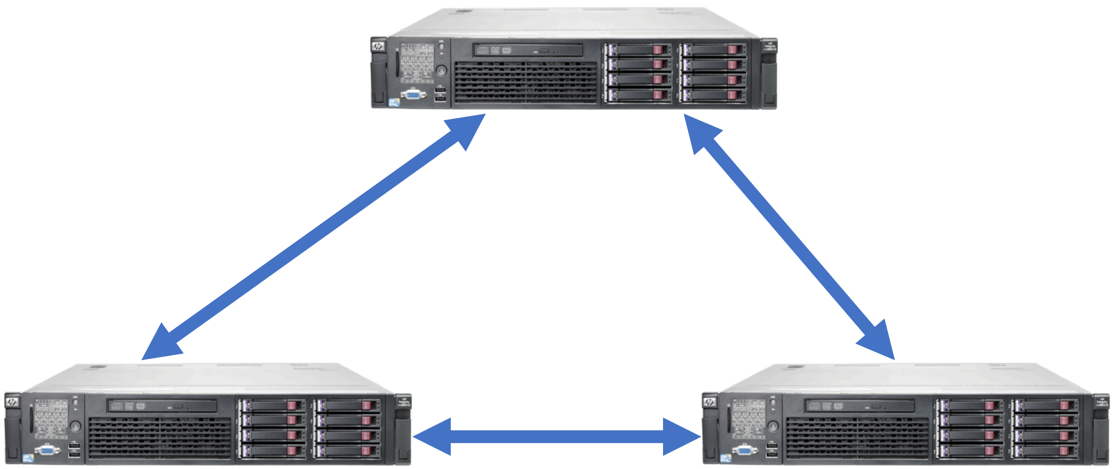
Distributed Systems Intro and Course Overview

COS 418: Distributed Systems
Lecture 1

Wyatt Lloyd

Distributed Systems, What?



1) Multiple computers
2) Connected by a network
3) Doing something together

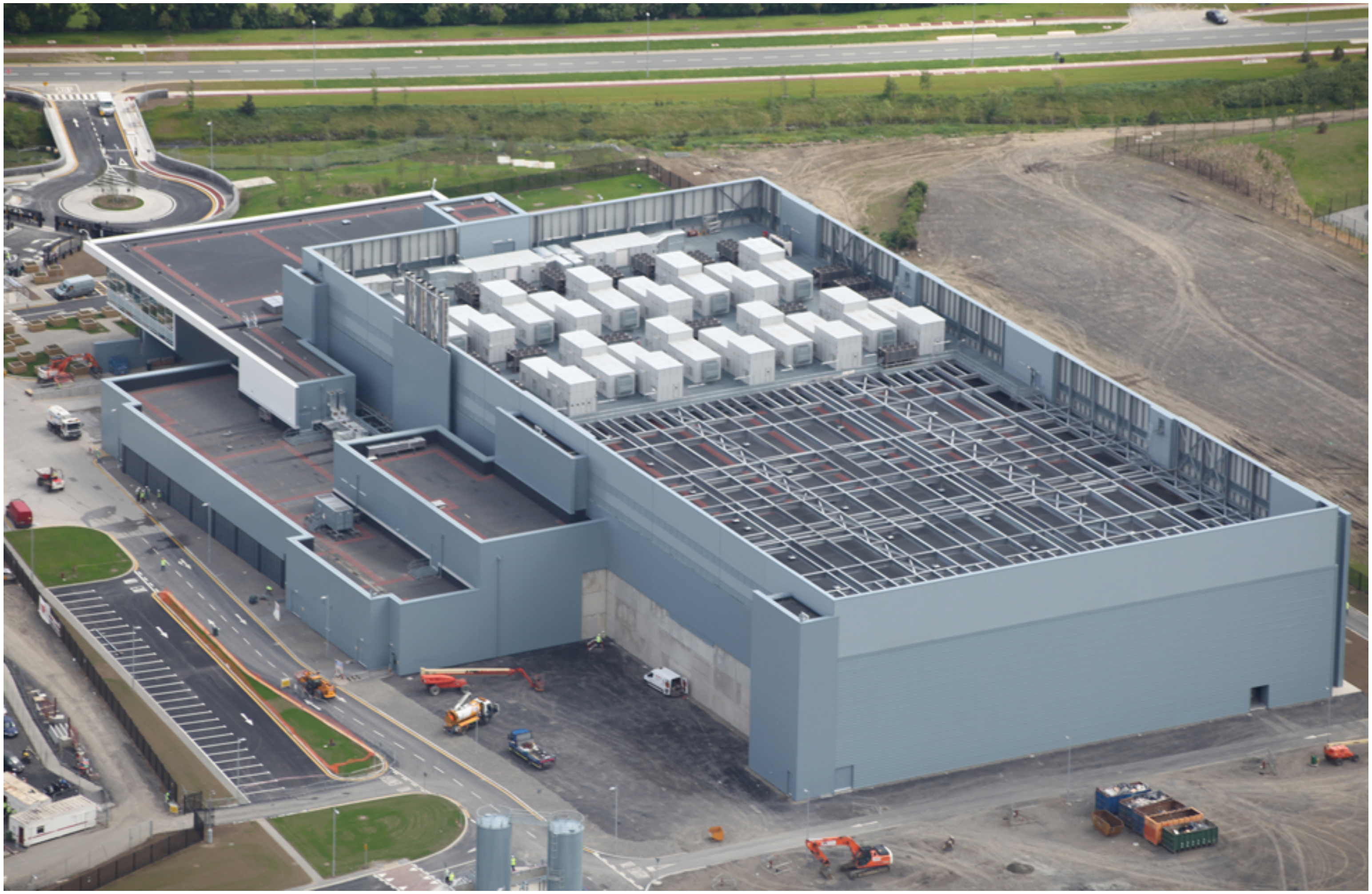
Distributed Systems, Why?

- Or, why not 1 computer to rule them all?
- Failure
- Limited computation/storage/…
- Physical location

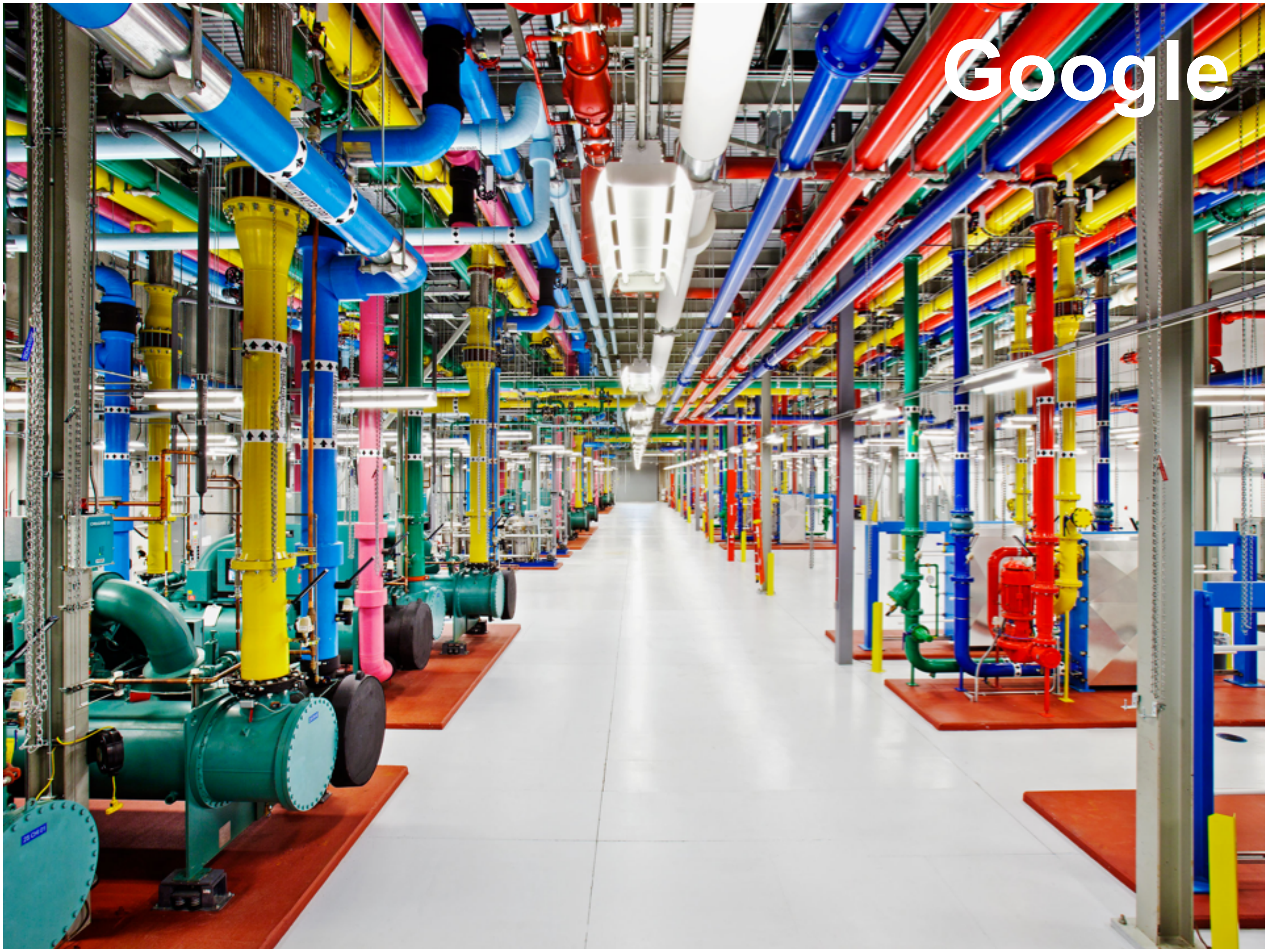
Distributed Systems, Where?

- Web Search (e.g., Google, Bing)
- Shopping (e.g., Amazon, Walmart)
- File Sync (e.g., Dropbox, iCloud)
- Social Networks (e.g., Facebook, Twitter)
- Music (e.g., Spotify, Apple Music)
- Ride Sharing (e.g., Uber, Lyft)
- Video (e.g., Youtube, Netflix)
- Online gaming (e.g., Fortnite, DOTA2)
- …

“The Cloud” is not amorphous

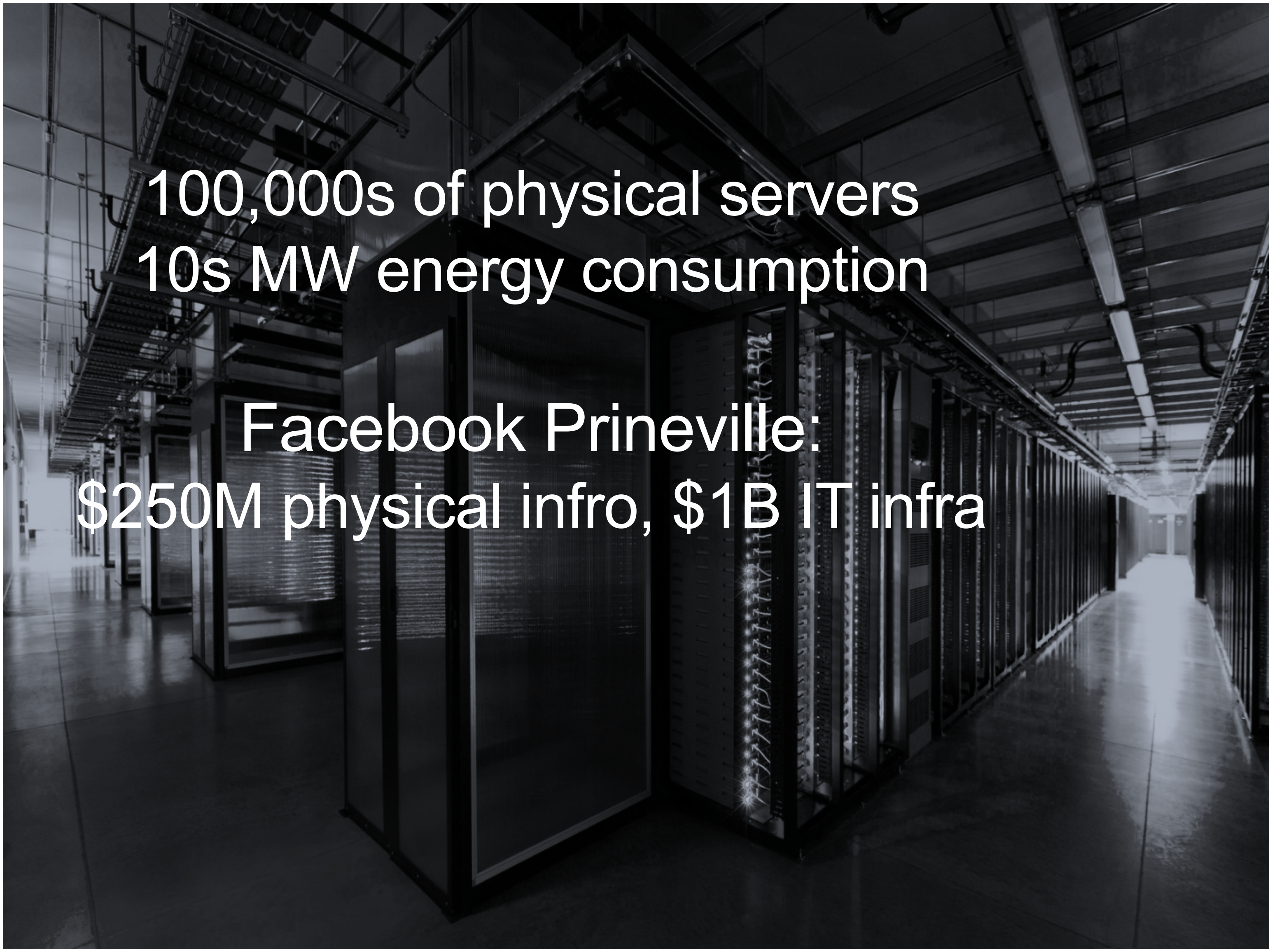


Microsoft


Google


Facebook

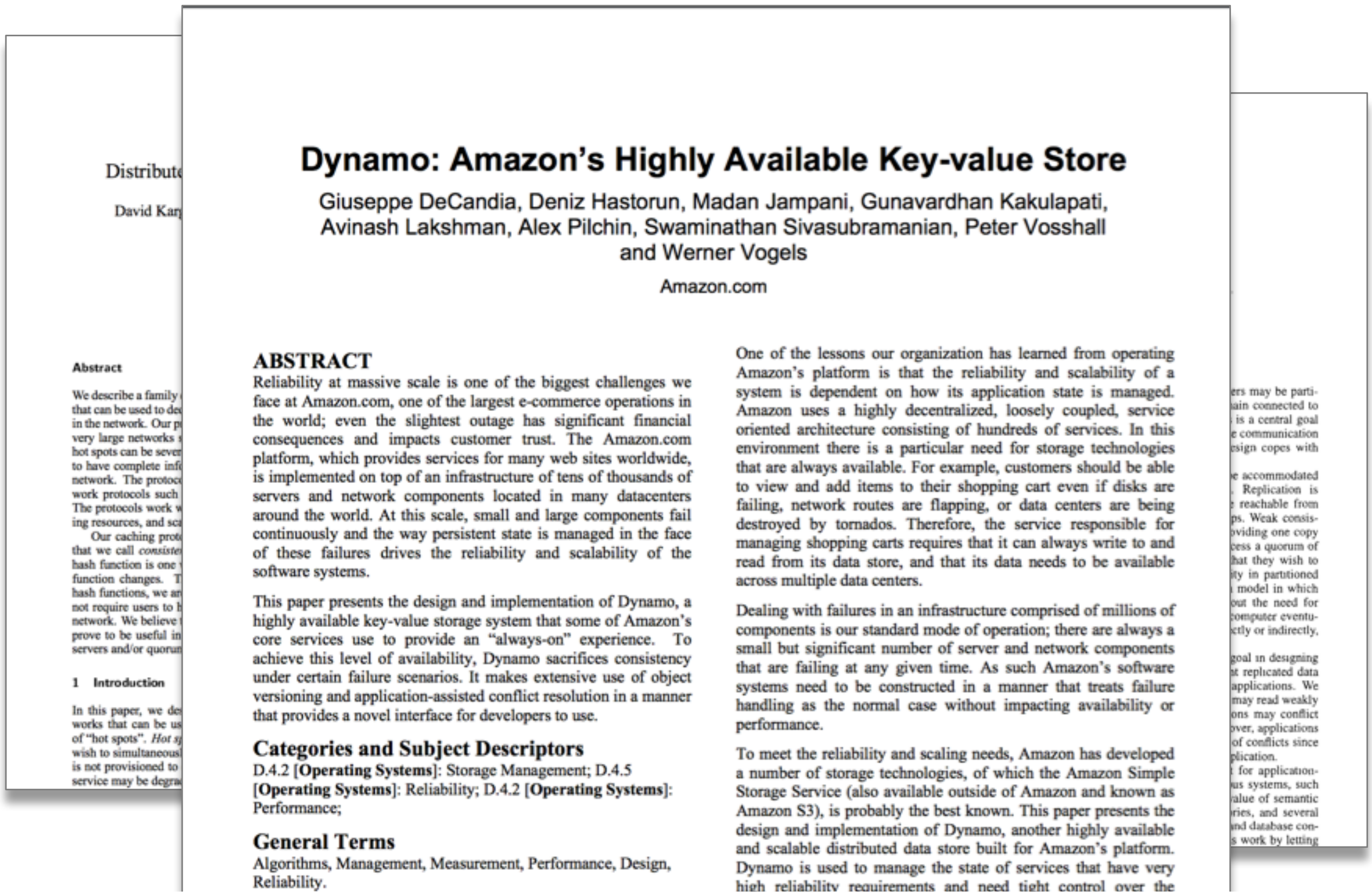

Facebook


100,000s of physical servers
10s MW energy consumption
Facebook Prineville:
$250M physical infro, $1B IT infra

Distributed Systems Goal

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

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

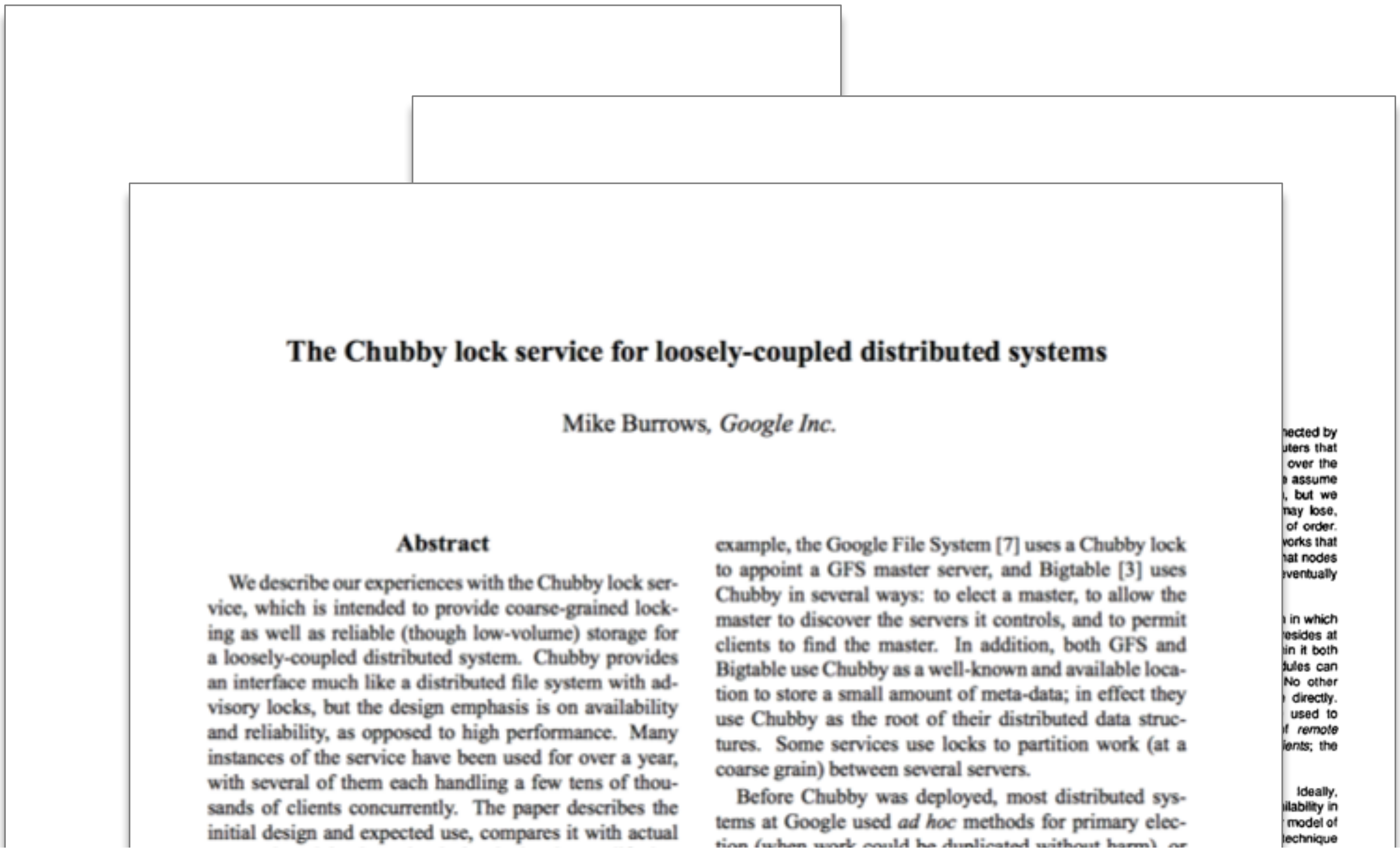
Algorithms, Management, Measurement, Performance, Design, Reliability.

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

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

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 elec-

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

Course Organization

Learning the material: People

- Lecture M/W 10-1050
  - Professor Wyatt Lloyd
  - Slides available on course website
- Precept:
  - TAs Theano Stavrinos, Zhenyu Song, and Chris Hodsdon
- Main Q&A forum: www.piazza.com
  - No anonymous posts or questions
  - Can send private messages to instructors
  - (Extra credit for answering)

Learning the Material: Lectures!

- Attend lectures and precepts and take notes!
  - Lecture slides posted day/night before
  - Recommendation: Print slides & take notes
  - Not everything covered in class is on slides
  - You are responsible for everything covered in class
- No required textbooks
  - Links to Go Programming textbook and two other distributed systems textbooks on website

Grading

- Five assignments (10% each)
  - 90% 24 hours late, 80% 2 days late, 50% >5 days late
  - **Three** free late days (we'll figure which one is best)

- Two exams (50% total)
  - Midterm exam before fall recess (25%)
  - Final exam during exam period (25%)

Policies: Write Your Own Code

Programming is an individual creative process. At first, discussions with friends is fine. When writing code, however, the program must be your own work.

Do not copy another person's programs, comments, or any part of submitted assignment. This includes character-by-character transliteration but also derivative works. Cannot use another's code, etc. even while "citing" them.

Writing code for use by another or using another's code is academic fraud in context of coursework.

Do not publish your code e.g., on github, during/after course!

Policies: Write Your Own Code

Programming is an individual creative process. At first, discussions with friends is fine. When writing code, however, the program must be your own work.

Do not copy another person's programs, comments, README description, or any part of submitted assignment. This includes character-by-character transliteration but also derivative works. Cannot use another's code, etc. even while "citing" them.

Writing code for use by another or using another's code is academic fraud in context of coursework.

Do not publish your code e.g., on github, during/after course!

Don't Plagiarize!

Assignment 1 (in three parts)

- Learn how to program in Go
  - Basic Go assignment (due **Sept 20**)
  - “Sequential” Map Reduce (due Sept 27)
  - Distributed Map Reduce (due Oct 4)

Warnings

This is a 400-level course, with expectations to match.

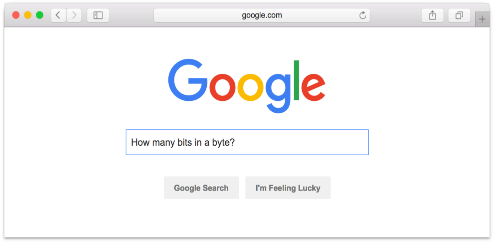
Warning #1:
Assignments are a LOT of work

- Assignment 1 is purposely easy to teach Go. Don't be fooled.
- Last year they gave 3-4 weeks for later assignments;
  many students started 3-4 days before deadline. **Disaster.**
- Distributed systems are hard
  - Need to understand problem and protocol, carefully design
  - Can take 5x more time to debug than "initially program"
- Assignment #4 builds on your Assignment #3 solution, i.e., you can't do #4 until your own #3 is working! (That's the real world!)

Warning #2:
Software engineering, not just programming

- COS126, 217, 226 told you how to design & structure your programs. This class doesn't.
- Real software engineering projects don't either.
- You need to learn to do it.
- If your system isn't designed well, can be *significantly* harder to get right.
- Your friend: test-driven development
  - We'll supply tests, bonus points for adding new ones

Warning #3:
Don’t expect 24x7 answers



- Try to figure out yourself
- Piazza not designed for debugging
  - Utilize right venue: Go to office hours
  - Send detailed Q’s / bug reports, not “no idea what’s wrong”
- Instructors are not on pager duty 24 x 7
  - Don’t expect response before next business day
  - Questions Friday night @ 11pm should not expect fast responses. Be happy with something before Monday.
- Implications
  - Students should answer each other (+ it’s worth credit)
  - Start your assignments early!

Topics Preview

Fundamentals

- Lectures
  - Network communication and Remote Procedure Calls
  - Time, logical clocks
  - Vector clocks, distributed snapshots

- Precepts
  - Lots of Go
  - Mapreduce (assignment 1)

Eventual Consistency and Scaling Out

- Lectures
  - Eventual consistency and Bayou
  - Peer-to-peer systems and Distributed Hash Tables
  - Scale-out key-value storage and Dynamo
- Precepts
  - More Go
  - Distributed snapshots (assignment 2)

Replicated State Machines

- Lectures
  - Replicated State Machines, Primary-Backup
  - Reconfiguration and View Change Protocols
  - Consensus and Paxos (and FLP)
  - Leader Election and RAFT
  - Byzantine Fault Tolerance

- Precepts
  - Viewstamped replication
  - RAFT (Assignments 3,4)

Strong Consistency and Scaling Out with Transactions

- Lectures
  - Strong consistency and the CAP Theorem
  - Atomic commit
  - Pessimistic concurrency control
  - Optimistic concurrency control
  - Spanner (Concurrency control + Paxos!)
  - The SNOW Theorem and Systems

- Precepts
  - Consistency
  - Concurrency control
  - Spanner and SNOW

Various Topics

- Lectures
  - Blockchains
  - Big data processing
  - Cluster scheduling and fairness
  - Cluster load testing
  - Content delivery networks

- Precepts
  - Big data systems