CSE 452
Distributed Systems

Tom Anderson
Distributed Systems

• How to make a set of computers work together
  – Reliably
  – Efficiently
  – At (huge) scale
  – With high availability

• Despite messages being lost and/or taking a variable amount of time

• Despite nodes crashing or behaving badly, or being offline
A Thought Experiment

Suppose there is a group of people, standing in a circle, two have green dots on their foreheads.

Without using a mirror or directly asking, can anyone tell if they themselves have a green dot?
A Thought Experiment

Suppose there is a group of people, standing in a circle, two have green dots on their foreheads.

Without using a mirror or directly asking, can anyone tell if they themselves have a green dot?

What if I say: someone has a green dot
  – Something everyone already knows!
There’s a difference between what you know and what you know others know.

And what others know you know.
What is a Distributed System?

A group of computers that work together to accomplish some task

– Independent failure modes
– Connected by a network with its own failure modes
Distributed Systems, 1990

Leslie Lamport:

“A distributed system is one where you can’t get your work done because some machine you’ve never heard of is broken.”
We’ve Made Some Progress

Today a distributed system is one where you can get your work done (almost always):

– wherever you are
– whenever you want
– even if parts of the system aren’t working
– no matter how many other people are using it
– as if it was a single dedicated system just for you
– that (almost) never fails
Concurrency is Fundamental

- **CSE 451: Operating Systems**
  - How to make a single computer work reliably
  - With many users and processes
- **CSE 461: Computer Networks**
  - How to connect computers together
  - Networks are a type of distributed system
- **CSE 444: Database System Internals**
  - How to manage (big) data reliably and efficiently
  - Primary focus is single node databases
Course Project

Build a sharded, linearizable, available key-value store, with dynamic load balancing and atomic multi-key transactions
Course Project

Build a sharded, linearizable, available key-value store, with dynamic load balancing and atomic multi-key transactions

- Key-value store: distributed hash table
- Linearizable: equivalent to a single node
- Available: continues to work despite failures
- Sharded: keys on multiple nodes
- Dynamic load balancing: keys move between nodes
- Multi-key atomicity: linearizable for multi-key ops
Project Mechanics

• Lab 0: introduction to framework and tools
  – Do Lab 0 **before** section this week (ungraded)
• Lab 1: exactly once RPC, key-value store
  – **Next** Thursday, **individually**
  – Lab 2-4: **pairs or individually**
• Lab 2: primary backup (tolerate failures)
• Lab 3: paxos (tolerate even more failures)
• Lab 4: sharding, load balancing, transactions
Project Tools

• Automated testing
  – Run tests: all the tests we can think of
  – Model checking: try all possible message deliveries and node failures

• Visual debugger
  – Control and replay over message delivery, failures

• Java, with restrictions
  – Model checker needs to collapse equivalent states
Project Rules

• OK
  – Consult with us or other students in the class

• Not OK
  – Look at other people’s code (in class or out)
  – Cut and paste code
Some Career Advice

Knowledge >> grades
Capability vs. Time

Capability (log scale)

Lotus 123

Microsoft Excel

Time
Capability vs. Time

Capability (log scale)

Time

Lotus 123

Microsoft Excel
Readings

• There is no adequate distributed systems textbook

• Instead, we’ve assigned:
  – Some tutorials/book chapters
  – A dozen+ research papers

• Both are important

• Read **before** class
  – See course web calendar page
Blogs

• How do you read a research paper?
  – An important skill, because research ideas often make it into practice

• Practice by blogging about papers
  – Write a short thought about the paper to the Canvas discussion thread; learn from other people’s blog entries

• Blog seven papers (one per week)
Some More Career Advice

The Technical Ladder

Knowing what should be built
Knowing what can be built
Knowing how to build it
Problem Sets

• Three problem sets
  – Done individually

• No midterm

• No final

• Course is not curved
Logistics

• Zoom for lectures, sections, office hours
  – Links in canvas/zoom

• Gitlab for lab assignments
  – Largely self-graded

• Ed for project Q&A

• Gradescope for problem sets and lab turn-ins

• Canvas for blog posts
Why Distributed Systems?

• Conquer geographic separation
  – 3.5B smartphone users; locality is crucial

• Availability despite unreliable components
  – System shouldn’t fail when one computer does

• Scale up capacity
  – Cycles, memory, disks, network bandwidth

• Customize computers for specific tasks
  – Ex: disaggregated storage, email, backup
End of Dennard Scaling

- Moore’s Law: transistor density improves at an exponential rate (2x/2 years)
- Dennard scaling: as transistors get smaller, power density stays constant
- Recent: power increases with transistor density
  - Scale out for performance
- All large scale computing is distributed
Example

• 2004: Facebook started on a single server
  – Web server front end to assemble each user’s page
  – Database to store posts, friend lists, etc.

• 2008: 100M users

• 2010: 500M

• 2012: 1B

• 2020: 2.5B

How do we scale up beyond a single server?
Facebook Scaling

• One server running both webserver and DB
• Two servers: webserver, DB
  – System is offline 2x as often!
• Server pair for each social community
  – E.g., school or college
  – What if friends cross servers?
  – What if server fails?
Two-tier Architecture

• Scalable number of front-end web servers
  – Stateless (“RESTful”): if crash can reconnect the user to another server
  – Run application code that is rapidly changing
  – Q: how does user find a front-end?

• Scalable number of back-end database servers
  – Run carefully designed distributed systems code
  – If crash, system remains available
  – Q: how do servers coordinate updates?
Three-tier Architecture

• Scalable number of front-end web servers
  – Stateless (“RESTful”): if crash can reconnect the user to another server

• Scalable number of cache servers
  – Lower latency (better for front end)
  – Reduce load (better for database)
  – Q: how do we keep the cache layer consistent?

• Scalable number of back-end database servers
  – Run carefully designed distributed systems code
And Beyond

• Worldwide distribution of users
  – Cross continent Internet delay ~ half a second
  – Amazon: reduction in sales if latency > 100ms

• Many data centers
  – One near every user
  – Smaller data centers just have web and cache layer
  – Larger data centers include storage layer as well
  – Q: how do we coordinate updates across DCs?
Properties We Want
(Google Paper)

• Fault-Tolerant: It can recover from component failures without performing incorrect actions. (Lab 2)

• Highly Available: It can restore operations, permitting it to resume providing services even when some components have failed. (Lab 3)

• Scalable: It can operate correctly even as some aspect of the system is scaled to a larger size. (Lab 4)
Typical Year in a Data Center

- ~0.5 data centers fail per year due to overheating
- ~1 power distribution failure (~500-1000 machines offline)
- ~1 rack-move (~500-1000 machines powered down)
- ~1 network rewiring (rolling outage of ~5% of machines down)
- ~20 rack failures (40-80 machines instantly disappear)
- ~5 racks go wonky (40-80 machines see 50% packet loss)
- ~8 network maintenances (random connectivity losses)
- ~12 router reloads
- ~3 router failures
- ~dozens of 30-second DNS outages
- ~1000 individual machine failures
- ~1000+ hard drive failures
- slow disks, bad memory, misconfigured machines, flaky machines, ...
Other Properties We Want (Google Paper)

• Consistent: The system can coordinate actions by multiple components often in the presence of concurrency and failure. (Labs 2-4)

• Predictable Performance: The ability to provide desired responsiveness in a timely manner. (Week 8)

• Secure: The system authenticates access to data and services (CSE 484)
Next Time: Remote Procedure Call

• Remote procedure call (RPC)
  – Abstraction of a procedure call, with arguments and return values
  – Executed on a remote node

• Challenges
  – Remote node might have failed
  – Network may have failed
  – Request may be dropped
  – Reply may be dropped
Thought Experiment

• Client sends a request to Amazon
• Network is flaky
  – Don’t hear back for a second
• Can you tell?
  – Request was lost
  – Server was down
  – Request got through, reply was lost
• Should the client resend?
Thought Experiment

• The client resends
• But the original packet got through
• What should the server do?
  – Crash?
  – Do the operation twice?
  – Something else?
Why Is DS So Hard?

- **System design**
  - Partitioning of responsibilities: what should client do, the caching layer, the storage layer?

- **Failures are endemic, partial and ambiguous**
  - If a server doesn’t reply, how do you tell if it is (a) the network, (b) the server, or c) neither: they are both just being slow?

- **Concurrency and consistency**
  - Distributed state, replicated state, caching
  - How do we keep this state consistent?
Why Is DS So Hard?

• Performance
  – Generating a single FB page involves calls to hundreds of different machines
  – Performance can be variable and unpredictable
  – Tail latency: limited by slowest machine

• Implementation and testing
  – Nearly impossible to test/reproduce all failure cases

• Security
  – Adversary can silently compromise machines and manipulate messages
Why Are Distributed Systems Hard?

• Asynchrony
  – Different nodes run at different speeds
  – Messages can be unpredictably, arbitrarily delayed
• Failures (partial and ambiguous)
  – Parts of the system can crash
  – Can’t tell crash from slowness
• Concurrency and consistency
  – Replicated state, cached on multiple nodes
  – How to keep many copies of data consistent?
Why Are Distributed Systems Hard?

• Performance
  – Have to efficiently coordinate many machines
  – Performance is variable and unpredictable
  – Tail latency: only as fast as slowest machine

• Testing and verification
  – Almost impossible to test all failure cases
  – Proofs (emerging field) are really hard

• Security
  – Need to assume adversarial nodes
Another Thought Experiment: Local vs. Remote Operations

• How long does it take to do a simple procedure call on a modern server?

• How long does it take to do the same operation on a different server in the same data center?

• On a server in a remote data center?
  – Speed of light is ~ 5us/mile