CSE 452/M552
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

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

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.

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

We’ve Made Some Progress

Leslie Lamport, circa 1990:

“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

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
- Lab 1: exactly once RPC, key-value store
  - **Next** Wednesday, **individually**
- Lab 2: primary backup (tolerate failures)
  - 452 students: pairs, 552 students: individually
- 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, not Go
  - 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

Readings and Blogs

- There exists no (even partially) adequate distributed systems textbook
  - Sabbatical: check back next year!
- Instead, we've assigned:
  - A few tutorials/book chapters
  - 14 research papers (first one a week from Wed.)
- How do you read a research paper?
- Blog **seven** papers
  - Write a short **unique** thought about the paper to the Canvas discussion thread (one per section)

Problem Sets

- Three problem sets
  - Done **individually**

- No midterm
- No final

The Science of Computers in the Classroom

- Don’t

Why Distributed Systems?

- Conquer geographic separation
  - 2.3B 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

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?

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

Typical Year in a Data Center

- ~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

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

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)

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 9)
- 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