An introduction to distributed systems

This outline accompanies a 12-16 hour overview class on distributed systems fundamentals. The course aims to introduce
software engineers to the practical basics of distributed systems, through lecture and discussion. Participants will gain an intuitive understanding of key distributed systems terms, an overview of the algorithmic landscape, and explore production concerns.

What makes a thing distributed?

Lamport, 1987:

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

- First glance: *nix boxen in our colo, running processes communicating via TCP or UDP.
- Or boxes in EC2, Rackspace, etc
- Maybe communicating over InfiniBand
- Separated by inches and a LAN
- Or by kilometers and the internet
- Most mobile apps are also taking part in a distributed system
- Communicating over a truly awful network
- Same goes for desktop web browsers
- It's not just servers–it's clients too!
- More generally: distributed systems are
- Made up of parts which interact
- Slowly
- And often unreliably
- Whatever those mean for you
- So also:
- Redundant CPUs in an airplane
- ATMs and Point-of-Sale terminals
- Space probes
- Paying bills
- Doctors making referrals
- Drunk friends trying to make plans via text message
- Every business meeting ever

Nodes and networks

- We call each part of a distributed system a *node*

- Also known as *processes*, *agents*, or *actors*

Nodes

- Characteristic latency
- Operations inside a node are "fast"
- Operations between nodes are "slow"
- What's fast or slow depends on what the system does
- Nodes are reliable
- Fail as a unit
- You know when problems occur
- State is coherent
- State transitions occur in a nice, orderly fashion
- Typically modeled as some kind of single-threaded state machine
- A node could *itself* be a distributed system
- But so long as that system as a whole provides "fast, coherent" operations, we can treat it as a single node.
- Formal models for processes
- Communicating Sequential Processes
- Pi-calculus
- Ambient calculus
- Actor model
- Formal models for node failure
- Crash-stop
- Crash-recover
- Crash-amnesia
- Byzantine

Networks as message flows

- Nodes interact via a *network*
- Humans interact via spoken words
- Particles interact via fields
- Computers interact via IP, or UDP, or SCTP, or …
- We model those interactions as discrete *messages* sent between nodes
- Messages take *time* to propagate
- This is the "slow" part of the distributed system
- We call this "latency"
- Messages can often be lost
- This is another "unreliable" part of the distributed system
- Network is rarely homogenous
- Some links slower/smaller/more-likely-to-fail than others

Causality diagrams

- We can represent the interaction of nodes and the network as a diagram
- Time flows left-to-right, or top-to-bottom
- Nodes are lines in the direction of time (because they stay in place)
- Messages as slanted paths *connecting* nodes

Synchronous networks

- Nodes execute in lockstep: time between node steps is always 1.
- Message delay is bounded
- Effectively a perfect global clock
- Easy to prove stuff about
- You probably don't have one

Semi-synchronous networks

- Like synchronous, but the clock is only approximate, e.g. in [c, 1]

Asynchronous networks

- Execute independently, whenever: step time is anywhere in [0, 1]
- Unbounded message delays
- No global clocks
- Weaker than semi- or synchronous networks
- Implies certain algorithms can't be as efficient
- Implies certain algorithms are *impossible*
- See e.g. Attiya & Mavronicolas, "Efficiency of Semi-Synchronous vs Asynchronous Networks"
- IP networks are definitely asynchronous
- But *in practice* the really pathological stuff doesn't happen
- Most networks recover in seconds to weeks, not "never"
  - Conversely, human timescales are on the orders of seconds to weeks
  - So we can't pretend the problems don't exist

When networks go wrong

- Asynchronous networks are allowed to
- Duplicate
- Delay
- Drop
- Reorder
- Drops and delays are indistinguishable

- Byzantine networks are allowed to mess with messages *arbitrarily*
- Including rewriting their content
- They mostly don't happen in real networks
  - Mostly

Low level protocols

TCP

- TCP *works*. Use it.
- Not perfect; you can go faster
- But you'll know when this is the case
- In practice, TCP prevents duplicates and reorders in the context of a single TCP conn
- But you're probably gonna open more than one connection
- If for no other reason than TCP conns eventually fail
- And when that happens, you'll either a.) have missed messages or b.) retry
- You can *reconstruct* an ordering by encoding your own sequence numbers on
  top of TCP

UDP

- Same addressing rules as TCP, but no stream invariants
- Lots of people want UDP "for speed"
- Don't consider that routers and nodes can and will arbitrarily drop packets
- Don't consider that their packets *will* be duplicated
- And reordered
- "But at least it's unbiased right?"
  - WRONG!
- This causes all kinds of havoc in, say, metrics collection
- And debugging it is *hard*
- TCP gives you flow control and repacks logical messages into packets
  - You'll need to re-build flow-control and backpressure
- TLS over UDP is a thing, but tough
- UDP is really useful where TCP FSM overhead is prohibitive
- Memory pressure
- Lots of short-lived conns and socket reuse
- Especially useful where best-effort delivery maps well to the system goals
- Voice calls: people will apologize and repeat themselves
- Games: stutters and lag, but catch up later
- Higher-level protocols impose sanity on underlying chaos

Clocks

- When a system is split into independent parts, we still want some kind of *order* for events
- Clocks help us order things: first this, THEN that

Wall Clocks

- In theory, the operating system clock gives you a partial order on system events
- Caveat: NTP is probably not as good as you think
- Caveat: Definitely not well-synced between nodes
- Caveat: Hardware can drift
- Caveat: By *centuries*
  - NTP might not care
  - http://rachelbythebay.com/w/2017/09/27/2153/
- Caveat: POSIX time is not monotonic by *definition*
  - Cloudflare 2017: Leap second at midnight UTC meant time flowed backwards
  - At the time, Go didn't offer access to CLOCK_MONOTONIC
  - Computed a negative duration, then fed it to rand.int63n(), which paniced
  - Caused DNS resolutions to fail: 1% of HTTP requests affected for several hours
  - https://blog.cloudflare.com/how-and-why-the-leap-second-affected-cloudflare-dns/
- Caveat: The timescales you want to measure may not be attainable
- Caveat: Threads can sleep
- Caveat: Runtimes can sleep
- Caveat: OS's can sleep
- Caveat: "Hardware" can sleep
- Just don't.

Lamport Clocks

- Lamport 1977: "Time, Clocks, and the Ordering of Events in a Distributed System"
- One clock per process
- Increments monotonically with each state transition: `t' = t + 1`
- Included with every message sent
- `t' = max(t, t_msg + 1)`
- If we have a total ordering of processes, we can impose a total order on events
- But that order could be pretty unintuitive

Vector Clocks

- Generalizes Lamport clocks to a vector of all process clocks
- `t_i' = max(t_i, t_msg_i)`
- For every operation, increment that process' clock in the vector
- Provides a partial causal order
- A < B iff all A_i <= B_i, and at least one A_i < B_i
- Specifically, given a pair of events, we can determine causal relationships
  - A in causal past of B implies A < B
  - B in causal past of A implies B < A
  - Independent otherwise
- Pragmatically: the past is shared; the present is independent
- Only "present", independent states need to be preserved
- Ancestor states can be discarded
- Lets us garbage-collect the past
- O(processes) in space
- Requires coordination for GC
- Or sacrifice correctness and prune old vclock entries
- Variants
- Dotted Version Vectors - for client/server systems, orders *more* events
- Interval Tree Clocks - for when processes come and go

GPS & Atomic Clocks

- Much better than NTP
- Globally distributed total orders on the scale of milliseconds
- Promote an asynchronous network to a semi-synchronous one
- Unlocks more efficient algorithms
- Only people with this right now are Google
- Spanner: globally distributed strongly consistent transactions
- And they're not sharing
- More expensive than you'd like
- Several hundred per GPS receiver
- Atomic clocks for local corroboration: $$$$?
- Need multiple types of GPS: vendors can get it wrong
- I don't know who's doing it yet, but I'd bet datacenters in the future will offer dedicated HW interfaces for bounded-accuracy time.

Review

We've covered the fundamental primitives of distributed systems. Nodes exchange messages through a network, and both nodes and networks can fail in various ways. Protocols like TCP and UDP give us primitive channels for

processes to communicate, and we can order events using clocks. Now, we'll discuss some high-level *properties* of distributed systems.

Availability

- Availability is basically the fraction of attempted operations which succeed.

Total availability

- Naive: every operation succeeds
- In consistency lit: every operation on a non-failing node succeeds
- Nothing you can do about the failing nodes

Sticky availability

- Every operation against a non-failing node succeeds
- With the constraint that clients always talk to the same nodes

High availability

- Better than if the system *weren't* distributed.
- e.g. tolerant of up to f failures, but no more
- Maybe some operations fail

Majority available

- Operations succeed *if* they occur on a node which can communicate with a majority of the cluster
- Operations against minority components may fail

Quantifying availability

- We talk a lot about "uptime"
- Are systems up if nobody uses them?
- Is it worse to be down during peak hours?
- Can measure "fraction of requests satisfied during a time window"
- Then plot that fraction over windows at different times
- Timescale affects reported uptime
- Apdex
- Not all successes are equal
- Classify operations into "OK", "meh", and "awful"

- Apdex = P(OK) + P(meh)/2
- Again, can report on a yearly basis
  - "We achieved 99.999 apdex for the year"
- And on finer timescales!
  - "Apdex for the user service just dropped to 0.5; page ops!"
- Ideally: integral of happiness delivered by your service?

Consistency

- A consistency model is the set of "safe" histories of events in the system

Monotonic Reads

- Once I read a value, any subsequent read will return that state or later values

Monotonic Writes

- If I make a write, any subsequent writes I make will take place *after* the first write

Read Your Writes

- Once I write a value, any subsequent read I perform will return that write (or later values)

Writes Follow Reads

- Once I read a value, any subsequent write will take place after that read

Serializability

- All operations (transactions) appear to execute atomically
- In some order
- No constraints on what that order is
- Perfectly okay to read from the past, for instance

Causal consistency

- Suppose operations can be linked by a DAG of causal relationships
- A write that follows a read, for instance, is causally related
  - Assuming the process didn't just throw away the read data

- Operations not linked in that DAG are *concurrent*
- Constraint: before a process can execute an operation, all its precursors must have executed on that node
- Concurrent ops can be freely reordered

Sequential consistency

- Like causal consistency, constrains possible orders
- All operations appear to execute atomically
- Every process agrees on the order of operations
- Operations from a given process always occur in order
- But nodes can lag behind

Linearizability

- All operations appear to execute atomically
- Every process agrees on the order of operations
- Every operation appears to take place *between* its invocation and completion
  times
- Real-time, external constraints let us build very strong systems

ACID isolation levels

- ANSI SQL's ACID isolation levels are weird
- Basically codified the effects of existing vendor implementations
- Definitions in the spec are ambiguous
- Adya 1999: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions
- Each ANSI SQL isolation level prohibits a weird phenomenon
- Read Uncommitted
  - Prevents P0: *dirty writes*
  - w1(x) … w2(x)
  - Can't write over another transaction's data until it commits
  - Can read data while a transaction is still modifying it
  - Can read data that will be rolled back
- Read Committed
  - Prevents P1: *dirty reads*
  - w1(x) … r2(x)
  - Can't read a transaction's uncommitted values
- Repeatable Read
  - Prevents P2: *fuzzy reads*
  - r1(x) … w2(x)

  - Once a transaction reads a value, it won't change until the transaction commits
- Serializable
  - Prevents P3: *phantoms*
  - Given some predicate P
  - r1(P) … w2(y in P)
  - Once a transaction reads a set of elements satisfying a query, that set won't change until the transaction commits
  - Not just values, but *which values even would have participated.*
- Cursor Stability
  - Transactions have a set of cursors
  - A cursor refers to an object being accessed by the transaction
  - Read locks are held until cursor is removed, or commit
  - At commit time, cursor is upgraded to a writelock
  - Prevents lost-update
- Snapshot Isolation
  - Transactions always read from a snapshot of committed data, taken before
    the transaction begins
  - Commit can only occur if no other committed transaction with an overlapping [start..commit] interval has written to any of the objects *we* wrote
  - First-committer-wins

Tradeoffs

- Ideally, we want total availability and linearizability
- Consistency requires coordination
- If every order is allowed, we don't need to do any work!
- If we want to disallow some orders of events, we have to exchange messages
- Coordinating comes (generally) with costs
- More consistency is slower
- More consistency is more intuitive
- More consistency is less available

Availability and Consistency

- CAP Theorem: Linearizability OR total availability
- But wait, there's more!
- Bailis 2014: Highly Available Transactions: Virtues and Limitations
- Other theorems disallow totally or sticky available…
  - Strong serializable
  - Serializable
  - Repeatable Read

  - Cursor Stability
  - Snapshot Isolation
- You can have *sticky* available…
  - Causal
  - PRAM
  - Read Your Writes
- You can have *totally* available…
  - Read Uncommitted
  - Read Committed
  - Monotonic Atomic View
  - Writes Follow Reads
  - Monotonic Reads
  - Monotonic Writes

Harvest and Yield

- Fox & Brewer, 1999: Harvest, Yield, and Scalable Tolerant Systems
- Yield: probability of completing a request
- Harvest: fraction of data reflected in the response
- Examples
  - Node faults in a search engine can cause some results to go missing
  - Updates may be reflected on some nodes but not others
  - Consider an AP system split by a partition
  - You can write data that some people can't read
  - Streaming video degrades to preserve low latency
- This is not an excuse to violate your safety invariants
  - Just helps you quantify how much you can *exceed* safety invariants
  - e.g. "99% of the time, you can read 90% of your prior writes"
- Strongly dependent on workload, HW, topology, etc
- Can tune harvest vs yield on a per-request basis
- "As much as possible in 10ms, please"
- "I need everything, and I understand you might not be able to answer"

Hybrid systems

- So, you've got a spectrum of choices!
- Chances are different parts of your infrastructure have different needs
- Pick the weakest model that meets your constraints
  - But consider probabilistic bounds; visibility lag might be prohibitive
  - See Probabilistically Bounded Staleness in Dynamo Quorums
- Not all data is equal
- Big data is usually less important
- Small data is usually critical
- Linearizable user ops, causally consistent social feeds

Review

Availability is a measure of how often operations succeed. Consistency models are the rules that govern what operations can happen and when. Stronger consistency models generally come at the cost of performance and availability. Next, we'll talk about different ways to build systems, from weak to strong consistency.

Avoid Consensus Wherever Possible

CALM conjecture

- Consistency As Logical Monotonicity
- If you can prove a system is logically monotonic, it is coordination free
- What the heck is "coordination"
- For that matter, what's "monotonic"?
- Monotonicity, informally, is retraction-free
- Deductions from partial information are never invalidated by new information
- Both relational algebra and Datalog without negation are monotone
- Ameloot, et al, 2011: Relational transducers for declarative networking
- Theorem which shows coordination-free networks of processes unaware of the
  network extent can compute only monotone queries in Datalog
  - This is not an easy read
- "Coordination-free" doesn't mean no communication
  - Algo succeeds even in face of arbitrary horizontal partitions
- In very loose practical terms
- Try to phrase your problem such that you only *add* new facts to the system
- When you compute a new fact based on what's currently known, can you ensure
  that fact will never be retracted?
- Consider special "sealing facts" that mark a block of facts as complete
- These "grow-only" algorithms are usually easier to implement
- Likely tradeoff: incomplete reads
- Bloom language
- Unordered programming with flow analysis
- Can tell you where coordination *would* be required

Gossip

- Message broadcast system
- Useful for cluster management, service discovery, CDNs, etc
- Very weak consistency

- Very high availability
- Global broadcast
- Send a message to every other node
- O(nodes)
- Mesh networks
- Epidemic models
- Relay to your neighbors
- Propagation times on the order of max-free-path
- Spanning trees
- Instead of a mesh, use a tree
- Hop up to a connector node which relays to other connector nodes
- Reduces superfluous messages
- Reduces latency
- Plumtree (Leitão, Pereira, & Rodrigues, 2007: Epidemic Broadcast Trees)

CRDTs

- Order-free datatypes that converge
- Counters, sets, maps, etc
- Tolerate dupes, delays, and reorders
- Unlike sequentially consistent systems, no "single source of truth"
- But unlike naive eventually consistent systems, never *lose* information
- Unless you explicitly make them lose information
- Works well in highly-available systems
- Web/mobile clients
- Dynamo
- Gossip
- INRIA: Shapiro, Preguiça, Baquero, Zawirski, 2011: "A comprehensive study of
  Convergent and Commutative Replicated Data Types"
- Composed of a data type X and a merge function m, which is:
  - Associative: m(x1, m(x2, x3)) = m(m(x1, x2), x3)
  - Commutative: m(x1, x2) = m(x2, x1)
  - Idempotent: m(x1, x1) = m(x1)
- Easy to build. Easy to reason about. Gets rid of all kinds of headaches.
- Did communication fail? Just retry! It'll converge!
- Did messages arrive out of order? It's fine!
- How do I synchronize two replicas? Just merge!
- Downsides
- Some algorithms *need* order and can't be expressed with CRDTs
- Reads may be arbitrarily stale
- Higher space costs

HATs

- Bailis, Davidson, Fekete, et al, 2013: "Highly Available Transactions, Virtues and Limitations"
- Guaranteed responses from any replica
- Low latency (1-3 orders of magnitude faster than serializable protocols!)
- Read Committed
- Monotonic Atomic View
- Excellent for commutative/monotonic systems
- Foreign key constraints for multi-item updates
- Limited uniqueness constraints
- Can ensure convergence given arbitrary finite delay ("eventual consistency")
- Good candidates for geographically distributed systems
- Probably best in concert with stronger transactional systems
- See also: COPS, Swift, Eiger, Calvin, etc

Fine, We Need Consensus, What Now?

- The consensus problem:
- Three process types
  - Proposers: propose values
  - Acceptors: choose a value
  - Learners: read the chosen value
- Classes of acceptors
  - N acceptors total
  - F acceptors allowed to fail
  - M malicious acceptors
- Three invariants:
  - Nontriviality: Only values proposed can be learned
  - Safety: At most one value can be learned
  - Liveness: If a proposer p, a learner l, and a set of N-F acceptors are non-faulty and can communicate with each other, and if p proposes a
    value, l will eventually learn a value.
- Whole classes of systems are *equivalent* to the consensus problem
- So any proofs we have here apply to those systems too
- Lock services
- Ordered logs
- Replicated state machines

- FLP tells us consensus is impossible in asynchronous networks
- Kill a process at the right time and you can break *any* consensus algo
- True but not as bad as you might think
- Realistically, networks work *often enough* to reach consensus
- Moreover, FLP assumes deterministic processes
  - Real computers *aren't* deterministic
  - Ben-Or 1983: "Another Advantage of free choice"
  - Nondeterministic algorithms *can* achieve consensus
- Lamport 2002: tight bounds for asynchronous consensus
- With at least two proposers, or one malicious proposer, N > 2F + M
  - "Need a majority"
- With at least 2 proposers, or one malicious proposer, it takes at least 2 message delays to learn a proposal.
- This is a pragmatically achievable bound
- In stable clusters, you can get away with only a single round-trip to a majority of nodes.
- More during cluster transitions.

Paxos

- Paxos is the Gold Standard of consensus algorithms
- Lamport 1989 - The Part Time Parliament
  - Written as a description of an imaginary Greek democracy
- Lamport 2001 - Paxos Made Simple
  - "The Paxos algorithm for implementing a fault-tolerant distributed system
    has been regarded as difficult to understand, perhaps because the original presentation was Greek to many readers [5]. In fact, it is among the
    simplest and most obvious of distributed algorithms… The last section explains the complete Paxos algorithm, which is obtained by the straightforward
    application of consensus to the state machine approach for building a
    distributed system—an approach that should be well-known, since it is the
    subject of what is probably the most often-cited article on the theory of
    distributed systems [4]."
- Google 2007 - Paxos Made Live

  - Notes from productionizing Chubby, Google's lock service
- Van Renesse 2011 - Paxos Made Moderately Complex
  - Turns out you gotta optimize
  - Also pseudocode would help
  - A page of pseudocode -> several thousand lines of C++
- Provides consensus on independent proposals
- Typically deployed in majority quorums, 5 or 7 nodes
- Several optimizations
- Multi-Paxos
- Fast Paxos
- Generalized Paxos
- It's not always clear which of these optimizations to use, and which can be safely combined
- Each implementation uses a slightly different flavor
- Used in a variety of production systems
- Chubby
- Cassandra
- Riak
- FoundationDB
- WANdisco SVN servers
- We're not even sure Chubby *is* Paxos, as described in the papers
- Paxos is really more of a *family* of algorithms than a well-described single entity
- Though we're pretty darn confident in the various proofs at this point

ZAB

- ZAB is the Zookeeper Atomic Broadcast protocol
- Junqueira, Reed, and Serafini 2011 - Zab: High-performance broadcast for
  primary-backup systems
- Differs from Paxos
- Provides sequential consistency (linearizable writes, lagging ordered reads)
- Useful because ZK clients typically want fast local reads
- But there's also a SYNC command that guarantees real-time visibility
- (SYNC + op) allows linearizable reads as well
- Again, majority quorum, 5 or 7 nodes

Humming Consensus

- Metadata store for managing distributed system reconfiguration
- Looks a little like CORFU's replicated log
- See also: chain replication

Viewstamped Replication

- Presented as a replication protocol, but also a consensus algorithm
- Transaction processing plus a view change algorithm
- Majority-known values are guaranteed to survive into the future
- I'm not aware of any production systems, but I'm sure they're out there
- Along with Paxos, inspired Raft in some ways

Raft

- Ongaro & Ousterhout 2014 - In Search of an Understandable Consensus Algorithm
- Lamport says it's easy, but we still have trouble grokking Paxos
- What if there were a consensus algorithm we could actually understand?
- Paxos approaches independent decisions when what we *want* is state machines
- Maintains a replicated *log* of state machine transitions instead
- Also builds in cluster membership transitions, which is *key* for real systems
- Very new, but we have a Coq proof of the core algorithm
- Can be used to write arbitrary sequential or linearizable state machines
- RethinkDB
- etcd
- Consul

Review

Systems which only add facts, not retract them, require less coordination to build. We can use gossip systems to broadcast messages to other processes, CRDTs to merge updates from our peers, and HATs for weakly isolated transactions. Serializability and linearizability require *consensus*, which we can obtain through Paxos, ZAB, VR, or Raft. Now, we'll talk about different *scales* of distributed systems.

Characteristic latencies

- Latency is *never* zero
- Bandwidth goes up and up but we're bumping up against the physical limits
  of light and electrons
- Latency budget shapes your system design
  - How many network calls can you afford?
- Different kinds of systems have different definitions of "slow"
- Different goals
- Different algorithms

Multicore systems

- Multicore (and especially NUMA) architectures are sort of like a distributed system
- Nodes don't fail pathologically, but message exchange is slow!
- Synchronous network provided by a bus (e.g. Intel QPI)
- Whole complicated set of protocols in HW & microcode to make memory look
  sane
- Non-temporal store instructions (e.g. MOVNTI)
- They provide abstractions to hide that distribution
- MFENCE/SFENCE/LFENCE
  - Introduce a serialization point against load/store instructions
  - Characteristic latencies: ~100 cycles / ~30 ns
  - Really depends on HW, caches, instructions, etc
- CMPXCHG Compare-and-Swap (sequentially consistent modification of memory)
- LOCK
  - Lock the full memory subsystem across cores!
- But those abstractions come with costs
- Hardware lock elision may help but is nascent
- Blog: Mechanical Sympathy
- Avoid coordination between cores wherever possible
- Context switches (process or thread!) can be expensive
- Processor pinning can really improve things
- When writing multithreaded programs, try to divide your work into independent chunks
  - Try to align memory barriers to work unit boundaries
  - Allows the processor to cheat as much as possible within a work unit

Local networks

- You'll often deploy replicated systems across something like an ethernet LAN
- Message latencies can be as low as 100 micros
- But across any sizable network (EC2), expect low millis
- Sometimes, packets could be delayed by *five minutes*
- Plan for this
- Network is within an order of mag compared to uncached disk seeks
- Or faster, in EC2
  - EC2 disk latencies can routinely hit 20ms
  - 200ms?
    * *20,000* ms???
    * Because EBS is actually other computers
    * LMAO if you think anything in EC2 is real

        - Wait, *real disks do this too*?
        - What even are IO schedulers?
- But network is waaaay slower than memory/computation
- If your aim is *throughput*, work units should probably take longer than a millisecond
- But there are other reasons to distribute
  - Sharding resources
  - Isolating failures

Geographic replication

- You deploy worldwide for two reasons
- End-user latency
  - Humans can detect ~10ms lag, will tolerate ~100ms
  - SF–Denver: 50ms
  - SF–Tokyo: 100 ms
  - SF–Madrid: 200 ms
  - Only way to beat the speed of light: move the service closer
- Disaster recovery
  - Datacenter power is good but not perfect
  - Hurricanes are a thing
  - Entire Amazon regions can and will fail
  - Yes, regions, not AZs
- Minimum of 1 round-trip for consensus
- Maybe as bad as 4 rounds
  - Maybe 4 rounds all the time if you have a bad Paxos impl (e.g. Cassandra)
- So if you do Paxos between datacenters, be ready for that cost!
- Because the minimum latencies are higher than users will tolerate
  - Cache cache cache
  - Queue writes and relay asynchronously
  - Consider reduced consistency guarantees in exchange for lower latency
  - CRDTs can always give you safe local writes
  - Causal consistency and HATs can be good calls here
- What about strongly consistent stuff?
- Chances are a geographically distributed service has natural planes of cleavage
  - EU users live on EU servers; US users live on US servers
  - Use consensus to migrate users between datacenters
- Pin/proxy updates to home datacenter
  - Which is hopefully the closest datacenter!
  - But maybe not! I believe Facebook still pushes all writes through 1 DC!
- Where sequential consistency is OK, cache reads locally!

– You probably leverage caching in a single DC already

Review

We discussed three characteristic scales for distributed systems: multicore processors coupled with a synchronous network, computers linked by a LAN, and
datacenters linked by the internet or dedicated fiber. CPU consequences are largely performance concerns: knowing how to minimize coordination. On LANs,
latencies are short enough for many network hops before users take notice. In geographically replicated systems, high latencies drive eventually consistent and datacenter-pinned solutions.

Common distributed systems

Outsourced heaps

- Redis, memcached, …
- Data fits in memory, complex data structures
- Useful when your language's built-in data structures are slow/awful
- Excellent as a cache
- Or as a quick-and-dirty scratchpad for shared state between platforms
- Not particularly safe

KV stores

- Riak, Couch, Mongo, Cassandra, RethinkDB, HDFS, …
- Often 1,2,3 dimensions of keys
- O(1) access, sometimes O(range) range scans by ID
- No strong relationships between values
- Objects may be opaque or structured
- Large data sets
- Often linear scalability
- Often no transactions
- Range of consistency models–often optional linearizable/sequential ops.

SQL databases

- Postgres, MySQL, Percona XtraDB, Oracle, MSSQL, VoltDB, CockroachDB, …
- Defined by relational algebra: restrictions of products of records, etc
- Moderate sized data sets

- Almost always include multi-record transactions
- Relations and transactions require coordination, which reduces scalability
- Many systems are primary-secondary failover
- Access cost varies depending on indexes
- Typically strong consistency (SI, serializable, strict serializable)

Search

- Elasticsearch, SolrCloud, …
- Documents referenced by indices
- Moderate-to-large data sets
- Usually O(1) document access, log-ish search
- Good scalability
- Typically weak consistency

Coordination services

- Zookeeper, etcd, Consul, …
- Typically strong (sequential or linearizable) consistency
- Small data sets
- Useful as a coordination primitive for stateless services

Streaming systems

- Storm, Spark…
- Usually custom-designed, or toolkits to build your own.
- Typically small in-memory data volume
- Low latencies
- High throughput
- Weak consistency

Distributed queues

- Kafka, Kestrel, Rabbit, IronMQ, ActiveMQ, HornetQ, Beanstalk, SQS, Celery, …
- Journals work to disk on multiple nodes for redundancy
- Useful when you need to acknowledge work now, and actually do it later
- Send data reliably between stateless services
- The *only* one I know that won't lose data in a partition is Kafka
- Maybe SQS?
- Queues do not improve end-to-end latency
- Always faster to do the work immediately
- Queues do not improve mean throughput

- Mean throughput limited by consumers
- Queues do not provide total event ordering when consumers are concurrent
- Your consumers are almost definitely concurrent
- Likewise, queues don't guarantee event order with async consumers
- Because consumer side effects could take place out of order
- So, don't rely on order
- Queues can offer at-most-once or at-least-once delivery
- Anyone claiming otherwise is trying to sell you something
- Recovering exactly-once delivery requires careful control of side effects
- Make your queued operations idempotent
- Queues do improve burst throughput
- Smooth out load spikes
- Distributed queues also improve fault tolerance (if they don't lose data)
- If you don't need the fault-tolerance or large buffering, just use TCP
- Lots of people use a queue with six disk writes and fifteen network hops where a single socket write() could have sufficed
- Queues can get you out of a bind when you've chosen a poor runtime

Review

We use data structure stores as outsourced heaps: they're the duct tape of distributed systems. KV stores and relational databases are commonly deployed as systems of record; KV stores use independent keys and are not well-suited to relational data, but offer improved scalability and partial failure vs SQL stores, which offer rich queries and strong transactional guarantees. Distributed search and coordination services round out our basic toolkit for building applications. Streaming systems are applied for continuous, low-latency processing of datasets, and tend to look more like frameworks than databases. Their dual, distributed queues, focus on the *messages* rather than the *transformations*.

A Pattern Language

- General recommendations for building distributed systems
- Hard-won experience
- Repeating what other experts tell me
  - Over beers
- Hearsay
- Oversimplifications
- Cargo-culting
- Stuff I just made up
- YMMV

Don't distribute

- Rule 1: don't distribute where you don't have to
- Local systems have reliable primitives. Locks. Threads. Queues. Txns.
  - When you move to a distributed system, you have to build from ground up.
- Is this thing small enough to fit on one node?
  - "I have a big data problem"
  - Softlayer will rent you a box with 3TB of ram for $5000/mo.
  - Supermicro will sell a 6TB box for ~$115,000 total.
- Modern computers are FAST.
  - Production JVM HTTP services I've known have pushed 50K requests/sec
  - Parsing JSON events, journaling to disk, pushing to S3
  - Protocol buffers over TCP: 10 million events/sec
  - 10-100 event batches/message, in-memory processing
- Can this service tolerate a single node's guarantees?
- Could we just stand up another one if it breaks?
- Could manual intervention take the place of the distributed algorithm?

Use an existing distributed system

- If we have to distribute, can we push the work onto some other software?
- What about a distributed database or log?
- Can we pay Amazon to do this for us?
- Conversely, what are the care and feeding costs?
- How much do you have to learn to use/operate that distributed system?

Never fail

- Buy really expensive hardware
- Make changes to software and hardware in a controlled fashion
- Dry-run deployments against staging environments
- Possible to build very reliable networks and machines
- At the cost of moving slower, buying more expensive HW, finding talent
- HW/network failure still *happens*, but sufficiently rare => low priority

Accept failure

- Distributed systems aren't just characterized by *latency*, but by *recurrent, partial failure*
- Can we accept this failure and move on with our lives?
- What's our SLA anyway?
- Can we recover by hand?

- Can we pay someone to fix it?
- Could insurance cover the damage?
- Could we just call the customer and apologize?
- Sounds silly, but may be much cheaper
- We can never prevent 100% of system failures
- Consciously choosing to recover *above* the level of the system
- This is how financial companies and retailers do it!

Backups

- Backups are essentially sequential consistency, BUT you lose a window of ops.
- When done correctly
  - Some backup programs don't snapshot state, which leads to FS or DB
    corruption
  - Broken fkey relationships, missing files, etc…
- Allow you to recover in a matter of minutes to days
- But more than fault recovery, they allow you to step back in time
  - Useful for recovering from logical faults
  - Distributed DB did its job correctly, but you told it to delete key data

Redundancy

- OK, so failure is less of an option
- Want to *reduce the probability of failure*
- Have the same state and same computation take place on several nodes
- I'm not a huge believer in active-spare
  - Spare might have cold caches, broken disks, old versions, etc
  - Spares tend to fail when becoming active
  - Active-active wherever possible
  - Predictability over efficiency
- Also not a huge fan of only having 2 copies
  - Node failure probabilities just too high
  - OK for not-important data
  - I generally want three copies of data
  - For important stuff, 4 or 5
  - For Paxos and other majority-quorum systems, odd numbers: 3, 5, 7 common
- Common DR strategy: Paxos across 5 nodes; 3 or 4 in primary DC
  - Ops can complete as soon as the local nodes ack; low latencies
  - Resilient to single-node failure (though latencies will spike)
  - But you still have a sequentially consistent backup in the other DC
  - So in the event you lose an entire DC, all's not lost

  - See Camille Fournier's talks on ZK deployment
- Redundancy improves availability so long as failures are uncorrelated
- Failures are not uncorrelated
  - Disks from the same batch failing at the same time
  - Same-rack nodes failing when the top-of-rack switch blows
  - Same-DC nodes failing when the UPS blows
  - See entire EC2 AZ failures
  - Running the same bad computation on every node will break every node
  - Expensive queries
  - Riak list-keys
  - Cassandra doomstones
  - Cascading failures
  - Thundering-herd
  - TCP incast

Sharding

- The problem is too big
- Break the problem into parts small enough to fit on a node
- Not too small: small parts => high overhead
- Not too big: need to rebalance work units gradually from node to node
- Somewhere around 10-100 work units/node is ideal, IMO
- Ideal: work units of equal size
- Beware hotspots
- Beware changing workloads with time
- Know your bounds in advance
- How big can a single part get before overwhelming a node?
- How do we enforce that limit *before* it sinks a node in prod?
  - Then sinks all the other nodes, one by one, as the system rebalances
- Allocating shards to nodes
- Often built in to DB
- Good candidate for ZK, Etcd, and so on
- See Boundary's Ordasity

Independent domains

- Sharding is a specific case of a more general pattern: avoiding coordination
- Keep as much independent as possible
  - Improves fault tolerance
  - Improves performance
  - Reduces complexity
- Sharding for scalability
- Avoiding coordination via CRDTs
- Flake IDs: generate globally unique identifiers locally

- Partial availability: users can still use some parts of the system
- Processing a queue: more consumers reduces the impact of expensive events

ID structure

- Things in our world have to have unique identifiers
- At scale, ID structure can make or break you
- Consider your access patterns
  - Scans
  - Sorts
  - Shards
- Sequential IDs require coordination: can you avoid them?
  - Flake IDs: *mostly* time-ordered identifiers, zero-coordination
  - See http://yellerapp.com/posts/2015-02-09-flake-ids.html
- For *shardability*, can your ID map directly to a shard?
- SaaS app: object ID can also encode customer ID
- Twitter: tweet ID can encode user ID

Immutable values

- Data that never changes is trivial to store
- Never requires coordination
- Cheap replication and recovery
- Minimal repacking on disk
- Useful for Cassandra, Riak, any LSM-tree DB.
- Or for logs like Kafka!
- Easy to reason about: either present or it's not
- Eliminates all kinds of transactional headaches
- Extremely cachable
- Extremely high availability and durability, tunable write latency
- Low read latencies: can respond from closest replica
- Especially valuable for geographic distribution
- Requires garbage collection!
- But there are good ways to do this

Mutable identities

- Pointers to immutable values
- Pointers are small! Only metadata!
- Can fit huge numbers of pointers on a small DB
- Good candidate for consensus services or relational DBs
- And typically, not many pointers in the system
- Your entire DB could be represented by a single pointer

- Datomic only has ~5 identities
- Strongly consistent operations over identities can be *backed* by immutable HA storage
- Take advantage of AP storage latencies and scale
- Take advantage of strong consistency over small datasets provided by consensus systems
- Write availability limited by identity store
  - But, reads eminently cachable if you only need sequential consistency
  - Can be even cheaper if you only need serializability
- See Rich Hickey's talks on Datomic architecture
- See Pat Helland's 2013 RICON West keynote on Salesforce's storage

Confluence

- Systems which are order-independent are easier to construct and reason about
- Also helps us avoid coordination
- CRDTs are confluent, which means we can apply updates without waiting
- Immutable values are trivially confluent: once present, fixed
- Streaming systems can leverage confluence as well:
- Buffer events, and compute+flush when you know you've seen everything
- Emit partial results so you can take action now, e.g. for monitoring
- When full data is available, merge with + or max
- Bank ledgers are (mostly) confluent: txn order doesn't affect balance
  - But when you need to enforce a minimum balance, no longer confluent
  - Combine with a sealing event (e.g. the day's end) to recover confluence
- See Aiken, Widom, & Hellerstein 1992, "Behavior of Database Production Rules"

Backpressure

- Services which talk to each other are usually connected by *queues*
- Service and queue capacity is finite
- How do you handle it when a downstream service is unable to handle load?

1. Consume resources and explode
2. Shed load. Start dropping requests.
3. Reject requests. Ignore the work and tell clients it failed.
4. Apply backpressure to clients, asking them to slow down.

- 2-4 allow the system to catch up and recover
- But backpressure reduces the volume of work that has to be retried
- Backpressure defers choice to producers: compositional
- Clients of load-shedding systems are locked into load-shedding

  - They have no way to tell that the system is hosed
- Clients of backpressure systems can apply backpressure to *their clients*
  - Or shed load, if they choose
- If you're making an asynchronous system, *always* include backpressure
  - Your users will thank you later
- Fundamentally: *bounding resources*
- Request timeouts (bounded time)
- Exponential backoffs (bounded use)
- Bounded queues
- Bounded concurrency
- See Zach Tellman, "Everything Will Flow"

Services for domain models

- The problem is composed of interacting logical pieces
- Pieces have distinct code, performance, storage needs
- Monolithic applications are essentially *multitenant* systems
  - Multitenancy is tough
  - But its often okay to run multiple logical "services" in the same process
- Divide your system into logical services for discrete parts of the domain model
- OO approach: each *noun* is a service
  - User service
  - Video service
  - Index service
- Functional approach: each *verb* is a service
  - Auth service
  - Search service
  - Dispatch/routing service
- Most big systems I know of use a hybrid
  - Services for nouns is a good way to enforce *datatype invariants*
  - Services for verbs is a good way to enforce *transformation invariants*
  - So have a basic User service, which is used *by* an Auth service
- Where you draw the line… well that's tricky
  - Services come with overhead: have as few as possible
  - Consider work units
  - Separate services which need to scale independently
  - Colocate services with tight dependencies and tight latency budgets
  - Colocate services which use complementary resources (e.g. disk and CPU)
  - By hand: Run memcache on rendering nodes
  - Newer shops: Google Borg, Mesos, Kubernetes

Structure Follows Social Spaces

- Production software is a fundamentally social artifact
- Natural alignment: a team or person owns a specific service
- Jo Freeman, "The Tyranny of Structurelessness"
  - Responsibility and power should be explicit
  - Rotate people through roles to prevent fiefdoms
  - Promotes information sharing
  - But don't rotate too often
  - Ramp-up costs in software are very high
- As the team grows, its mission and thinking will formalize
- So too will services and their boundaries
- Gradually accruing body of assumptions about service relation to the world
- Punctuated by rewrites to respond to changing external pressures
- Tushman & Romanelli, 1985: Organizational Evolution
- Services can be libraries
- Initially, *all* your services should be libraries
- Perfectly OK to depend on a user library in multiple services
- Libraries with well-defined boundaries are easy to extract into services later
- Social structure governs the library/service boundary
- With few users of a library, or tightly-coordinated users, changes are easy
- But across many teams, users have varying priorities and must be convinced
- Why should users do work to upgrade to a new library version?
- Services *force* coordination through a defined API deprecation lifecycle
  - You can also enforce this with libraries through code review & tooling
- Services enable centralized control
- Your performance improvements affect everyone instantly
- Gradually shift to a new on-disk format or backing database
- Instrument use of the service in one place
- Harder to do these things with libraries
- Services have costs
- The failure-complexity and latency overhead of a network call
- Tangled food web of service dependencies
- Hard to statically analyze codepaths
- You thought library API versioning was hard
- Additional instrumentation/deployment
- Services can use good client libraries
- That library might be "Open a socket" or an HTTP client
  - Leverage HTTP headers!
  - Accept headers for versioning
  - Lots of support for caching and proxying
  - Haproxy is an excellent router for both HTTP and TCP services
- Eventually, library might include mock IO

  - Service team is responsible for testing that the service provides an API
  - When the API is known to be stable, every client can *assume* it works
  - Removes the need for network calls in test suites
  - Dramatic reduction in test runtime and dev environment complexity

Review

When possible, try to use a single node instead of a distributed system. Accept that some failures are unavoidable: SLAs and apologies can be cost-effective. To handle catastrophic failure, we use backups. To improve reliability, we introduce redundancy. To scale to large problems, we divide the problem into shards. Immutable values are easy to store and cache, and can be referenced by mutable identities, allowing us to build strongly consistent systems at large scale. As software grows, different components must scale independently, and we break out libraries into distinct services. Service structure goes hand-in-hand with teams.

Production Concerns

- More than design considerations
- Proofs are important, but real systems do IO

Distributed systems are supported by your culture

- Understanding a distributed system in production requires close cooperation
  of people with many roles
- Development
- QA
- Operations
- Empathy matters
- Developers have to care about production
- Ops has to care about implementation
- Good communication enables faster diagnosis

Test everything

- Type systems are great for preventing logical errors
- Which reduces your testing burden
- However, they are *not* great at predicting or controlling runtime performance
- So, you need a solid test suite

- Ideally, you want a *slider* for rigorousness
- Quick example-based tests that run in a few seconds
- More thorough property-based tests that can run overnight
- Be able to simulate an entire cluster in-process
- Control concurrent interleavings with simulated networks
- Automated hardware faults
- Testing distributed systems is much, much harder than testing local ones
- Huge swath of failure modes you've never even heard of
- Combinatorial state spaces
- Bugs can manifest only for small/large/intermediate time/space/concurrency

"It's Slow"

- Jeff Hodges: The worst bug you'll ever hear is "it's slow"
- Happens all the time, really difficult to localize
- Because the system is distributed, have to profile multiple nodes
  - Not many profilers are built for this
  - Sigelman et al, 2010: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure
  - Zipkin
  - Big tooling investment
- Profilers are good at finding CPU problems
  - But high latency is often a sign of IO, not CPU
  - Disk latency
  - Network latency
  - GC latency
  - Queue latency
- Try to localize problem using application-level metrics
  - Then dig in to process and OS performance

Instrument everything

- Slowness (and outright errors) in prod stem from the interactions *between* systems
- Why? Because your thorough test suite probably verified that the single system was mostly correct
- So we need a way to understand what the system is doing in prod
  - In relation to its dependencies
  - Which can, in turn, drive new tests
- In a way, good monitoring is like continuous testing
- But not a replacement: these are distinct domains
- Both provide assurance that your changes are OK
- Want high-frequency monitoring
  - Production behaviors can take place on 1ms scales

  - TCP incast
  - ~1ms resolution *ideally*
  - Ops response time, in the limit, scales linearly with observation latency
  - ~1 second end to end latency
  - Ideally, millisecond latencies, maybe ms resolution too
  - Usually cost-prohibitive; back off to 1s or 10s
  - Sometimes you can tolerate 60s
- And for capacity planning, hourly/daily seasonality is more useful
- Instrumentation should be tightly coupled to the app
  - Measure only what matters
  - Responding to requests is important
  - Node CPU doesn't matter as much
  - Key metrics for most systems
  - Apdex: successful response WITHIN latency SLA
  - Latency profiles: 0, 0.5, 0.95, 0.99, 1
    * Percentiles, not means
    * BTW you can't take the mean of percentiles either
  - Overall throughput
  - Queue statistics
  - Subjective experience of other systems latency/throughput
    * The DB might think it's healthy, but clients could see it as slow
    * Combinatorial explosion–best to use this when drilling into a failure
  - You probably have to write this instrumentation yourself
  - Invest in a metrics library
- Out-of-the-box monitoring usually doesn't measure what really matters: your
  app's behavior
  - But it can be really useful in tracking down causes of problems
  - Host metrics like CPU, disk, etc
  - Where your app does something common (e.g. rails apps) tools like New
    Relic work well
- Superpower: distributed tracing infra (Zipkin, Dapper, etc)
  - Significant time investment

Logging

- Logging is less useful at scale
- Problems may not be localized to one node
  - As requests touch more services, must trace through many logfiles
  - Invest in log collection infrastructure
  - ELK, Splunk, etc
- Unstructured information is harder to aggregate

  - Log structured events

Shadow traffic

- Load tests are only useful insofar as the simulated load matches the actual load
- Consider dumping production traffic
- Awesome: kill a process with SIGUSR1, it dumps five minutes of request load
- Awesome: tcpdump/tcpreplay harnesses for requests
- Awesome: shadowing live prod traffic to your staging/QA nodes

Versioning

- Protocol versioning is, as far as I know, a wide-open problem
- Do include a version tag with all messages
- Do include compatibility logic
- Inform clients when their request can't be honored
  - And instrument this so you know which systems have to be upgraded

Rollouts

- Rollouts are often how you fix problems
- Spend the time to get automated, reliable deploys
- Amplifies everything else you do
- Have nodes smoothly cycle through to prevent traffic interruption
  - This implies you'll have multiple versions of your software running at
    once
  - Versioning rears its ugly head
- Inform load balancer that they're going out of rotation
- Coordinate to prevent cascading failures
- Roll out only to a fraction of load or fraction of users
- Gradually ramp up number of users on the new software
- Either revert or roll forward when you see errors
- Consider shadowing traffic in prod and comparing old/new versions
  - Good way to determine if new code is faster & correct

Feature flags

- We want incremental rollouts of a changeset after a deploy
- Introduce features one by one to watch their impact on metrics
- Gradually shift load from one database to another

- Disable features when rollout goes wrong
- We want to obtain partial availability when some services are degraded
- Disable expensive features to speed recovery during a failure
- Use a highly available coordination service to decide which codepaths to enable, or how often to take them
- This service should have minimal dependencies
  - Don't use the primary DB
- When things go wrong, you can *tune* the system's behavior
- When coordination service is down, fail *safe*!

Oh no, queues

- Every queue is a place for things to go horribly, horribly wrong
- No node has unbounded memory. Your queues *must* be bounded
- But how big? Nobody knows
- Instrument your queues in prod to find out
- Queues exist to smooth out fluctuations in load
- Improves throughput at expense of latency
- If your load is higher than capacity, no queue will save you
  - Shed load or apply backpressure when queues become full
  - Instrument this
  - When load-shedding occurs, alarm bells should ring
  - Backpressure is visible as upstream latency
- Instrument queue depths
  - High depths is a clue that you need to add node capacity
  - End to end queue latency should be smaller than fluctuation timescales
  - Raising the queue size can be tempting, but is a vicious cycle
- All of this is HARD. I don't have good answers for you
  - Ask Jeff Hodges why it's hard: see his RICON West 2013 talk
  - See Zach Tellman - Everything Will Flow

Review

Running distributed systems requires cooperation between developers, QA, and operations engineers. Static analysis, and a test suite including example- and property-based tests, can help ensure program correctness, but understanding production behavior requires comprehensive instrumentation and alerting. Mature
distributed systems teams often invest in tooling: traffic shadowing, versioning, incremental deploys, and feature flags. Finally, queues require special care.

Further reading

Online

- Mixu has a delightful book on distributed systems with incredible detail. http://book.mixu.net/distsys/
- Jeff Hodges has some excellent, production-focused advice. https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/
- The Fallacies of Distributed Computing is a classic text on mistaken assumptions we make designing distributed systems. http://www.rgoarchitects.com/Files/fallacies.pdf
- Christopher Meiklejohn has a list of key papers in distributed systems. http://christophermeiklejohn.com/distributed/systems/2013/07/12/readings-in-distributed-systems.html

Trees

- Nancy Lynch's "Distributed Algorithms" is a comprehensive overview of the field from a more theoretical perspective