ECE428: Distributed Systems

An Exercise in Complicated Rust Code

Pradyun Narkadamilli

Contents

1 System Model

- How do you define a distributed system?
  - *Broadly*: Independent components or elements connected by a network, which communicate by passing messages across said network
  - These components usually have a *common goal*, and achieve this while appearing as a coherent system
    - Ex: Many servers appearing as unified Google Drive service
- Algorithms must be formulated differently when dealing with distributed systems
  - Typical algorithms are formulated sequentially, they need to be *striped* across different systems, where steps may occur concurrently between different systemes
  - Full sequencing of an algorithm even after being striped renders the "distributed" nature of the algorithm completely irrelevant - you might as well kill yourself
- There are three main "aspects" when designing a distributed system
  - Processes communicate with each other to coordinate, and the time to do so may not be fixed
  - Different process *may not be synchronized* (different clocks)
  - Failure may happen in a process *or* on a communication channel

1.1 Communication

- Processes communicate via network sockets, even if on the same host
- Consider the model of a **communication channel** - what properties to consider?
  - *Latency* is the delay it takes for the message to *go across the channel*
    - Any queueing delays over the network are considered in latency
  - *Bandwidth* is the total amount of information that can be sent per unit time
    - Sometimes called channel bandwidth, for obvious reasons
- Total time to send data is a composite of bandwidth and latency
  - Approximation: $T = \frac{size}{B} + L$

1.2 Modeling Synchronicity

- Different computers will have different internal clocks, so they may have different frequencies and phase
- Two big models for a distributed system
  - **Synchronous Systems**
    - Known upper/lower bounds on time per step of process

    * Bounded message passing delays and clock drift
  - **Asynchronous Systems**
    * No bounds on any of the 3 factors above
    * Most real systems fall under this model, but it is *possible* to construct a synchronous system

- Time-critical systems (like a supercomputer) *can* be made to be synchronous, but it's expensive

1.3 Component Failures

- **Omission**: Process/channel fails to perform an action that was expected
  - Crashed process, runtime errors, etc. etc.
- **Arbitrary Failures**: Any type of error, best summarized as "something breaking" or "buggy code"
- **Timing Failures**: Timing guarantees are not met in a synchronous system
- Something to consider is how to orchestrate *failure detection*
  - Two rudimentary options to check if remote is alive:
    1. Periodically "ping" system, check for an `ack`
       * A ping timeout can be set to $2 \times$ (max network delay) in a synchronous system
       * Asynchronous timeouts may need to be dynamic or obscenely high
         · Can be set experimentally by measuring the max RTT at runtime, and waiting for some multiple of it
    2. Design the remote to periodically send a "heartbeat"
       * Synchronous timeout is max net delay minus min net delay
       * Asynchronous timeout is some factor of observed delay
- Any failure detection mechanism must guarantee two properties to be *correct*
  - *Completeness* means that every failed process is detected eventually
  - *Accuracy* means that there are no false positives
  - Both check-alive methods above are considered correct in a *synchronous system*
    * In an asynchronous system, timeouts can potentially be aggresive, violating accuracy
- It is *impossible* to achieve both *completeness* and *accuracy* in an **asynchronous** system
- Worst-case failure detection times
  - Ping-Ack: $T + \Delta_1 - \Delta$ (Δ is time taken for last ping from p to reach q)
  - Heartbeat: $T + \Delta_2 + \Delta$ (Δ is time taken for last heartbeat from q to reach p)
  - Note that Ping-Ack requires two messages per detection, heartbeat has one

- **Two Generals Problem**
  - Both are thinking when to attack - they both want to *attack at the same time*
  - Messages must be passed to coordinate attack time, can potentially be dropped
  - Both sides should continually send messages *until* the other side confirms a receipt
  - Receipt will terminate the ingress message stream
    * There is a shitfuck case where the ack gets killed *and* all ingress messages get dropped, so there is a false assumption of the receiver of communication success
    * We ignore the shitfuck case, this might be okay in practice

1.3.1 Fancy Heartbeating

- Want to extend our idea of heartbeating to a system with many machines
- *Centralized Heartbeating*: All heartbeats sent to a single source, but failure rate is high if the central server fails
- *Ring Heartbeating*: All heartbeats are sent to the next server in a ring.
  - Single machine failure is fine (next machine times out) but if multiple machines fail then you won't know how many
  - Additional overhead for when you need to repair the ring
- *All-to-All Heartbeats*: Everyone keeps track of all other machines. Perfect state capture, but lots of bandwidth

2 Time

- Clocks are useful (not signals, like literal system clocks)
  - Can check how long it took for a request to go somewhere
  - Timestamps can be used to order events
  - Can timestamp actions as well for logging purposes
- In a distributed systems, the clocks (wall clocks) need to be synchronized for accurate timestamping
  - Different computers will have a clock *skew*, relative difference
  - There will also be a *drift rate*, which is rate at which a clock skews from a perfect reference
- In synchronous systems, drift rate is bounded, but skew is technically unbounded
- Two big types of clocks
  - Quartz crystal clocks, which drift $\approx 10^{-6} \frac{s}{s}$
    * Skew is $\approx$ 30 minutes after 60 years

  - Atomic clocks have a drift rate of $\approx 10^{-13} \frac{s}{s}$
    - Skew is about 0.18 ms after 60 years, used for standardized clocks (U.S naval observatory)

- Clocks can be synchronizes either *internally* or *externally*
  - **External synchronization** is done with an authoritative clock for *accurate timestamps*, skew bounded to D
  - **Internal synchronization** is done within a system for relatively matched timestamps
  - External synchronization imputes internal synchronization if done completely, with a bound of $2D$

- How to estimate propagation delay for synchronization in a synchronous system?
  - Client sends packet to other server asking for its current time, packet is received after some comm delay
  - Since comm delay is bounded in a synchronous system, add $\frac{\min + \max}{2}$ to packet time
    - Provably minimizes worst case skew to $\frac{\max - \min}{2}$

- Need more complex algorithms for synchronizing two asynchronous systems

- *Cristian Algorithm*
  - When a packet is sent, client will measure the RTT for *this packet*
  - Offset received packet time by $\frac{RTT}{2}$

- *Berkeley Algorithm*
  - Only works with *internal* synchronization
  - Assume hub-spoke model (single server, many clients)
  - Server periodically polls clients to see what their clock time is
  - Server uses the Cristian Algorithm to estimate the time per client
  - Average local time is used as the new "reference" time
  - Send an offset to all clients of how much they should adjust their clock
    - Offsets are not invalidated by network delays, but a reference time would be

- *Network Time Protocol*
  - Single primary UTC synchronization clock
  - Internet devices are organized in to levels of a tree called "strata"
  - Each strata synchronizes with a server in a higher strata
  - Authentication mechanisms are used to ensure security
  - Clocks can be synchronized via a server multicasting timestamps on LAN (low accuracy) or a procedure-call (Cristian algorithm)
  - *Symmetric mode* is used to synchronize the lower-strata servers

- *NTP Symmetric Mode*: A&B exchange messages, record the TX/RX timestamps
  - A/B will exchange their local timestamps, and compute their relative offsets
- Asynchronous Synchronization Options (Recap)
  - Cristian Algorithm: Client-server synchronization
  - Berkeley Algorithm: Internal synchronization between clients
  - NTP: Internet-wide hierarchical synchronization

3 Consensus

- **Consensus**: Each process proposes a value, and all processes must *agree* on one of these values
  - Some criteria should be used on how to
  - You can consider total ordering a form of consensus
- We can formalize this concept into the *consensus problem*
  - You have N processes
  - Each process begins an undecided state, proposes a value v_i
  - At some point the process will set a decision d_i, and enters a "decided" state
- *Consensus algorithms* solve the above problem with a couple of requirements
  - **Termination**: All processes must eventually set the decision
  - **Agreement**: All correct processes should make the same decision
  - **Integrity**: If all correct processes propose the same value, all correct processes should use that for decision
- **Round-Based Algorithms**
  - Processes are synchronized and operate in "rounds"
  - A "round" completes in $\epsilon + T$ time
    - e.g the start/end time of a round in two processes is off by a max of ϵ
  - Algorithm runs in $f + 1$ rounds, where f is the number of node failures
  - Communication channels are assumed to be reliable (TCP-esque)
  - Every round, multicast the *new* values you receive between rounds
  - When rounds end, you can now use a decision criterion on all the received values
- Consensus **cannot** be solved in asynchronous sytems - you either violate safety or liveess
  - We can create an algorithm that fulfills *some* of the criterion

3.1 Paxos

- Popular algorithm for asynchronous conensus, under relaxed conditions
  - May not terminate, but it fulfills safety
- A process will possess some of 3 roles
  - *Proposers*: propose values to *Acceptors*
  - *Acceptors*: Accept values proposed under some conditions, tell the world
    - An acceptor can't just accept *everything*
    - Once a value is accepted, any higher ID acceptances need to have same value
  - *Learners*: Learns whichever value the *majority* accepted
    - Majority includes **crashed processes**, which can recover
- Everyone is a learner, but some processes are proposers or acceptors
- The algorithm is **dual-phase**
  - *Phase I*
    - Proposer selects a proposal number, and sends a `prepare` w/ it to at least a majority of acceptors
    - After this point, acceptors cannot respond/accept any lower-ID `prepare`
    - The `prepare` will receive an `OK` response saying it agrees, and if there was a prior acceptance
  - *Phase II*
    - Proposers will send an `accept` request to the acceptors if it receives `OK` from a majority of acceptors
    - If proposer does not get enough `OKs`, it will wait then re-propose with a higher proposal number
    - Response needs to be sent to proposer with `OK` after `accept`
  - Once a majority is hit with a value in Phase II, it is decided
    - A `decided` message is sent to everyone at this point
  - Every time an acceptance is made, the acceptor sends the value/proposal to *distinguished learners*
    - The aristocrats will determine if a decision is reached, and inform the other learners
    - Use multiple aristocrats for better failure handling - if all of them fail, the re-proposal mechanism fires
- Paxos functions best with a single leader-proposer who is *also* the distinguished learner
  - This can lead to livelock because of failures in leader election, but it's still "safe"

3.2 Raft

- Practical systems will need to decide on a value ordering for a *log*
  - You could run Paxos multiple times per entry, but this gets fucked in performance and complexity
- **Raft** is a consensus algorithm guaranteeing servers execute same commands in the same order
  - We keep a consensus module to guarant log replication
  - If a majority of servers are online, system will continue to make progress
  - Tries to be designed as an *understandable* system instead of Paxos black-magic
- Raft uses a *leader*-based algorithm
  - Simplifies normal operation and decomposes the problem into clear roles
  - Improves efficiency compared to leaderless decentralized algos
- Raft - surprisingly, has a simple checklist of function
  1. Elect a leader, and re-elect on crashes
  2. Neutralize the old leaders
  3. Log Replication
  4. Guarant post-election safety and consistency
- Raft communicates with **RPCs** - *remote procedure calls*
  - A server can call a function/procedure on a different process
- **Raft Leader Elections**
  - A server is always a *leader*, a *follower*, or a *candidate*
    - Leader handles all client interaction and log replication - 1 at a time
    - Follower is passive, will respond to incoming requests
    - Candidates are used to elect new leaders
  - Time is divded into ”terms” - each has an ”election” followed by normal operation
    - If there is a split vote, then the term has no leader, and normal operation isn't hit
  - Servers all start up as followers - they must receive heartbeats from a leader to remain so
    - If there is an election timeout *without* a heartbeat, then election starts
    - Follower promotes to a *candidate* then starts the election
  - Raft election terminates when a server receives a vote from server majority *or* if leader hits them with the ”shut the fuck up”
    - If election times out without a victor, term is incremented and a new election begins
  - Elections must be *safe* (only one winner) and *live* (someone eventually wins)

        * We can guarant the former but not the latter

- **Raft Leader Neutering**
  - Terms are used to detect stale leaders/candidates - RPCs must be tagged with sender's term
  - If the sender term is older than receiver, the RPC is *rejected*, and the sender reverts to a follower
  - If the *receiver* term is older, then it reverts to a follower then processes the RPC
  - This way if a stale leader tries to do anything, it's either neutered or slapped into reality

3.3 Blockchain

- Good example of *distributed* consensus
  - Bitcoin is still a currency - all transactions need an agreed order, and can't be double-done
- Transactions are grouped into a "block"
  - Block is added to a *chain* by the block leader
  - Every node picks a random number, winner is the one whose hash is less than a threshold
  - Hash of previous log/messages used as a *seed* for the proof of work hash
- Solutions on the blockchain should be quickly *verified* but not quickly *found*
- To avoid "tie" coditions, nodes will always work on the *longest chain* that they receive
  - A transaction is "committed" when 6 more are linked afterwards (for Bitcoin)
  - The longest chain represents "majority" usually, since the most proof of work was put into it
- Doing the proof of work of bitcoin is incentivized via a *mining reward* - this is why crypto-mining was so popular.
  - T is adjusted every now and then so that one block is added every 10 minutes
  - Small T will slow transactions, big T will waste *lots* of effort during chain splits

4 Timestamps and Ordering

- Synchronized clocks are useful because we can determine some global event ordering
  - Good for debug, reconciling updates/recency, and generating recovery timestamps of some sort
- Consider some system with n processes
  - Each process has its own state

  - State changes on rx/tx a message or some computation

- Events between processes need to be ordered
  - Ex: *Happened-Before Relationship (HB)* is denoted $a \rightarrow b$ for a before b
  - HB is a transitive relation, and supports the notion of "perspective"
    - $a \rightarrow_k b$ means that this ordering is from the perspective of process k
    - Perceived ordering can set constraints on the larger global ordering

- **Lamport's Logical Clock**
  - Each process keeps a local clock L_i, initialized to 0
  - Clock incremented before timestamping an event
  - Clock is sent *with* a message - on RX message, max of local and RX clock is taken
    - RX event happens *after* clock prioritization
  - $L(a) < L(b) \not\Rightarrow a \rightarrow b$ notably - $a||b$ possible

- **Vector Clocks**
  - Each process maintains a vector of clocks per process, all initialized to 0
  - Personal clock is incremented before timestamping, vector is sent w/ message
  - On message RX, the clock for *that process* is max'd w/ the received clock
    - Receive event is still timestamped w/ *our* clock

- We create a notion of comparison for different vector clocks
  - $A = B \equiv A[i] = B[i] \forall i$
  - $A \leq B \equiv A[i] \leq B[i] \forall i$
  - $A < B \equiv A \leq B \wedge \exists j, A[j] < B[j]$

- If $A < B \Rightarrow a \rightarrow b$
  - If $A = B$ or none of the ordering relations hold, then $a||b$

- *Timestamp Tradeoffs*
  - Physical timestamp imputes absolute ordering but requires strict clock sync
  - Lamport's timestamps will sometimes conflate a causal and concurrent relation
  - Vector timestamps capture all causal relations but require big messages

5 State and Snapshotting

- Before we talked about *processes* having state
  - *Channels* can also have state (like message channels)
  - They need to know pending messages - this channel state can be computed from proc states
- Global snapshotting is useful for checkpointing, finding unreferenced objects, deadlock detection, and debug
  - Encompasses state of each process and channel in system at a certain time - hard to capture though
  - If snapshots are done relative to physical time, then clock synchronization becomes an obvious problem
- Can use some terminology to solidify our discussion of state
  - history(p_i) or h_i is all the events of process in a list
  - prefix history(p_i^k) or h_i^k gives first k events for process
  - s_i^k gives the state of a process after k events
- Slightly different terminology when dealing with global state (when do we pull state for each process?)
  - A **cut** refers to taking prefix history of each process up to a point c_i (differs per process)
  - The *frontier* of a cut refers to the set of most recent event per process before the cut
  - The global state of a cut S is pretty obviously derived
- A cut is called consistent iff
$$\forall e \in C(f \rightarrow e \equiv f \in C)$$
  - Effectively, the cause of each event in the cut should *also* be in the cut
  - It's okay if a causal event hasn't fully propagated yet when the cut is made
- A global state can be consistent iff it corresponds to a consistent cut
- **Chandy-Lamport Algorithm**: records a global snapshot with consistent state, identifies a consistent cut to do so
  - Corresponding state at each process is recorded once cut is identified
  - Models any two processes as having *two* one-way FIFO channels
    - messages all arrive in one piece, without duplication
    - assume no channel failures or process failures either
  - Any process can initiate the algorithm, but its effects are relevant to every process
- *CLA steps*

  - Initiator should record their own state and creates a *marker message* - it sends this to all orhter processes
  - When a process receives the marker *for the first time*, it records its state, then sends the marker to all *other* processes
  - Processes will keep recording messages they receive on other channels *until* they receive the marker from it
  - The algorithm terminates when every process receives a marker on *every one of their channels*

- *Liveness* is the guarantee that something "good" will eventually happen
  - This is satisfied if any possible continuance of execution will hit a state that is live at some point

- *Safety* is the guarantee that nothing "bad" will every happen
  - This is satisfied if every possible continuance of execution will never hit a state that is unsafe

- *Stable global predicates* refer to the idea that once a system is live, it is live *forever* afterwards
  - Every reachable state must be alive

6 Multicast

- There are 3 big communication modes
  - *Unicast* is a 1-to-1 process communication scheme
    * Best effort, makes "intact" guarantees but not reliability guarantees
    * Guarants inorder delivery
  - *Broadcast* refers to a 1-to-all scheme
  - *Multicast* refers to a 1-to-*group* scheme - this group is more granular than broadcast
    * Ideally, we desire *reliability* and *ordered* transmission

- For an application, we consider the multicast protocol to have two interface functions
  - `multicast(g, m)` sends a message to a process group `g`, where `g` includes the current process
  - `deliver(m)` receives a message from the multicast protocol to the current application

- *Basic Multicast* or *B-Multicast* is straightforward
  - Simply use unicast while iterating over each of the processes in the group
  - We consider `B-deliver` to be identical to a unicast `receive`

- *Reliable Multicast* or *R-Multicast* expands on this implementation

  - We wish to achieve some reliability conditions
    * *Integrity*: a correct process will deliver a message at most *once*
    * *Validity*: if a correct process multicasts, it will eventually self-deliver the message
    * *Agreement*: if a correct process delivers a message `m`, then all other processes in the sent group will eventually deliver it (all-or-nothing)
  - R-multicast acts as a user of the B-multicast interface to achieve reliability
    * On message delivery, if it is a new message, we take note
    * We then re-multicast that message via B-multicast to the rest of the group

- There are 3 popular multicast orderings
  - *FIFO*: A single sender's messages are delivered inorder w.r.t each other at all receivers
  - *Causal Order*: multicasts sending causally related events will be delivered in an order obeying the relation
    * Guarantees FIFO ordering
  - *Total Order*: All processes receive all multicasts *in the same order* (the messages have a strict order)
    * Does not pay attention to order of multicast sending, and has no relation to causal ordering
    * May require delay of delivery at some processes

- To FIFO-order multicast, we implement a layer in between the B-multicast and application interfaces
  - Each receiver maintains a per-sender sequence number - messages are FIFO'd if there is a jump
  - Sequence number incremented as contiguous messages get popped off the FIFO
  - To make this reliable just use R-multicast instead of B-multicast, hook FIFOing on R-deliver

- Similar approach to guarant Total Ordering on multicast, need a sequence number
  - Can either use a centralized sequencer or a decentralized mechanism (ISIS)

- *Centralized Sequencer*
  - Process elected to act as leader/sequencer
  - Multicast messages sent to group *and* sequencer- this will maintain a global sequence number S
  - When the sequencer receives message, it re-multicasts with a sequence number attached

- *ISIS*
  - Sender multicasts to the group - receivers will reply with a *proposed* priority
    * Proposal must be larger than all observed/proposed priorities thus far

  - Message is stored in a priority queue - agreed or proposed, whichever is known
  - Sender will take maximum across group and re-multicast message ID with the agreed prio
  - Receivers can use this multicast to finalize the message priority, deliver when eligible
- *Causal Ordering* is most similar to FIFO's implementation
  - You still have a vector of per-sender sequence numbers, but you send "vector timestamps"
  - Check if this is $\leq$ compared to all other most vector timestamps from senders, deliver if so
- Important to consider more efficient mechanism when *sending* messages across the network
  - Looped unicast propagates a message to the same network node multiple times
- Can instead do *tree-based multicast* - construct an MST of network nodes, send unicast along tree
  - If we construct a tree w/ the routers, we can do an *IP-based multicast*
  - If a node fails, then we have to consider the overhead of tree construction
- We can also use the *gossip* approach
  - Probabilistically send message to a couple of nodes, and do the same when you receive one
  - No guarantees on complete network propagation, but it's good enough for many applications

7 Mutual Exclusion

- Usual 391 crap on mutually exclusive access and critical sections
  - On a single system, we can use semaphores - this is a shared var though
  - Need to somehow support the idea of mutexes in a *distributed* system
- Any algorithm for mutual exclusion must make 3 guarantees
  - *Safety*: Only one process should execute on shared state at a time
  - *Liveness*: Every request access to the shared state is eventually guaranteed
  - *Ordering*: requests are granted in the order they're made
- **Central Server Algorithm**
  - Elect a leader server - keeps a queue of waiting requests from processes attempting CS access
  - Keeps a token that, when sent, allows holder to make access to the shared state
  - Processes can `enter` to request a token and `exit` to award the token back to server

  - May not necessarily guarantee ordering, which is fine, that one is optional
  - Does not require a lot of bandwidth, relatively good client delay, and tolerable synchronization delay
  - Leader can bottleneck this algorithm, and acts as a single failure point

- **Ring-Based Algorithm**
  - N processes organized in a ring - each process can send a message to next one in ring
  - 1 token is being passed - if you are not using it, pass it along immediately
  - Still does not guarantee ordering, and bandwidth is higher (has idle bandwidth)
  - Potentially better delay if there is high contention on the shared state
  - Still $O(n)$ for client/synchronization delay

- **Ricart-Agrawala Algorithm**
  - *Does not use a token-based approach* - depends on causality and multicast
  - Lower CS waiting time compared to the ring-based algorithm
  - On enter, set state to wanted, and multicast a timestamped `request`
  - Once all other processes respond with a `reply` of valid timestamp, you are active
  - Buffer any requests while waiting or held, reply to them all in order once done w/ CS
    - When holding, reply to requests w/ smaller vector timestamp than our "want" to avoid deadlock

- RA Algo guarantees *all 3* of our desired qualities in a mutex algorithm!
  - High bandwidth, but O(1) client and synchro delays!

- **Maekawa's Algorithm**
  - Improves on RA by only requiring replies from *some* processes - specifically a "voting set"
    - Each process is in its own voting set
    - Voting set formed by making a square array w/ processes
    - The row/column of curr proc in the array above is voting set
    - Each voter gives permission to *one* process at any given time
      - Process needs permission from its *entire* voting set
    - On enter, process multicasts a request and waits for all other voters to reply
    - On exit, a `release` is multicasted to the voter set
    - When a `request` is received, send `reply` if no other requests active, else buffer it
      - On `release`, send a `reply` to the first valid request in buffer
    - Guarants safety, but can have liveness and ordering issues (potential deadlock)

8 Leader Election

- Many algorithms refer to a central or "leader" process/server
  - How to elect or decide which server this is?
  - How to replace the leader in failure cases?
- *Election algorithm* will elect a non-faulty leader and ensure consensus on this leader
- **Election Algorithm Criteria**
  - Any process can initiate election, but at most one election at a time
  - If multiple processes initiate an election, the elections together should lead one leader
  - Election process shuld be *symmetric* regardless of the initiator
  - Election algorithms must always elect a *valid* leader for **safety**
  - Election algorithms must always elect *a* leader and terminate the run for **liveness**
- **Ring Election Algorithm**
  - Send election message to next process in the ring
  - On election message receive, send another one to the next in ring
    - Elect yourself if attribute greater, else elect the predecessor
  - Message propagates around the ring until it reaches the "leader"
    - If received message elects *you*, then the ring has terminated
  - Only send election message *once* to avoid the "tie" case
  - Up to $O(n^2)$ messages, but $O(n)$ turnaround guaranteed
- **Bully Algorithm**
  - Send `election` message to processes with higher priority/process id
  - If `disagree` received, stand down, and wait for `coordinator` message
    - If timeout, you are fucked! Start another `election`
  - If `election` received, respond with `disagree`, start your own election
  - If timeout after `election`, you are the leader! Send `coordinator` downwards

9 Transactions and Concurrency

- A **transaction** is a series of operations executed by a client on a server (or servers)
  - Generally separates into read/write - overwriting or getting state
- Transactions have some properties by design
  - We want *atomicity* - happens fully, or not at all (is an "unbreakable" unit)
    - Transactions will *commit* to confirm tentative updates, or *abort* to rollback

  - We want to be *consistent* with any required rules
    - Check validity of tentative values at commit time, abort if anything violated
  - Multiple transactions should be *isolated*, with no unintended cross-talk
    - Can either execute all transactions serially or have concurrency w/ serial equivalence
    - Can have *pessimistic* concurrency control w/ locks or *optimistic* concurrency control w/ a commit-time check for serial equivalence
  - We want our values to be *durable* after a crash for persistence
    - Need some sort of permanent storage and replication across servers
  - Generally abbreviated to **ACID**

9.1 Locking for Isolation

- We can either have *global* locks or *R/W* locks - we prefer the latter for better throughput
  - Acquire reader to read, writer to write – simple enough
- For any transaction, locks should all be acquired in phase 1, and all released in phase 2
  - Can lead to a *deadlock* if phase 1 allocates a reader lock and then a writer lock in two processes ast the same time
  - Generally – deadlock happens if 3 conditions hold (this is an iff)
    1. Some objects accessed in exclusive lock modes
    2. Transactions holding locks are *not* preempted
    3. There is some circular wait cycle in the wait-for-graph
       - What you like to think about as "cross-dependencies"
- Deadlock can be avoided!
  - Lock all objects *atomically* in the beginning - you will get everything in one shot, or wait a while
  - You can timeout the transaction if lock cannot be acquired - though this can cause inefficiencies
  - Check wait-for graph periodically for cycles (and therefore, deadlocks)
    - Can abort transaction(s) in this cycle if found to break loop

9.2 Opimistic Concurrency Control

- Optimal concurrency, since many things can just run in parallel
  - You can think of this like a fire & forget scheme
  - Used by many big companies' apps and KV stores
  - Better than pessimistic *when conflicts are rare*
- *First Cut Approach*

  - Write and read objects freely, but check for equivalence at commit
  - If a transaction is aborted because of inconsistency, rollback state
  - On rollback, track down any error propagated transactions, nuke those too
    * We call these *cascading aborts* – these are recursive, and screw with perf

- *Timestamped Ordering*
  - Each transaction gets a timestamp ID
  - Check to make sure that (on write) data isn't clobbered before lower timestamp access, read isn't clobbered by higher timestamp
  - Abort if any rules are violated (no shit, sherlock)
  - Check slides for actual rules idfk

9.3 Distributed Transactions

- Standard ACID challenges are still there, but A/I are now *way* harder
  - You need atomicity and isolation between multiple servers!
- Atomicity reduces down to the *consensus* problem (also called atomic commit)
  - Need to ensure that all servers commit T, or *no* servers commit T
  - We can have a coordinator server that initiates the transaction – can be separate, or an object-haver
  - Can have multiple coordinators handling multiple transactions
- *One-Phase Commit*: client relays commit/abort to coordinator. Coordinator tells everyone else
  - Server w/ object can't really tell coordinator whether it can abort (consistency can fail)
  - Server can crash before receiving the commit, with some updates still in its memory
  - Big issue – *no reciprocity* in communication
- *Two-Phase Commit*
  - Coordinator sends transaction to other servers, they all reply `Y/N`
  - If coordinator sees that **all** other servers can commit da ting, tell everyone to commit
  - If timeout or animous, tell everyone to abort (prevents OOM)
  - For crash safety, save anything tentative into storage *before* initial reply
  - Coordinator logs all decisions and tx/rx on disk – can recover log state
  - Server can poll coordinator to check for its failure – it will block until the coord receives
- *Isolation* falls into two points of responsibility
  - Each server applies concurrency control to its objects

  - Together, servers must guarant serial equivalence via per-server timestamped ordering
  - Aborts from the timestamped ordering will be relayed to coordinator, per Two-Phase Commit
  - Locking will be handled locally per-server
  - Each server reports its wait-for relations to the coordinator, which constructs a global graph
- So far we've assumed a central coordinator, which causes scalability issues failure centralization
- Deadlocks can be detected via *edge chasing*
  - Probe messages are forwaded to servers in the edges of the wait-for graph
  - If a server who probed receives the message back, then there is a cycle!
  - By using local wait-for relationships, globally the forwarding creates a coherent wait-for graph
- *Edge Chasing Phases*
  - *Initiation* is when a server starts detection because it knows that a transaction is waiting on another
    * Detection initiated by sending probe messages w/ the wait-for relationship to other servers
  - In *detection* servers will receive probes and determine if a deadlock is occurring (who am I waiting for) or if someone else should receive the probe
  - In *resolution* the cycle is detected, and 1+ transactions are *aborted* to free contested lcoks
- Susceptible to phantom deadlocks (false positives) because "wait-for edges" may have *disappeared*
  - Cycle can be detected by using stale edges to close the cycle, leading to spurious aborts
- Transactions can be *sharded* (distributed) or *replicated* across servers
  - A combination of both even!
  - Sharding improves load-balancing/scalability, replication improves fault-tolerance and availability
- Node failures are *common* – think about the Google story from JDean!
  - Good to *replicate* data so that you can be resilient to failures
  - All replicas should be consistent, and the client should not think replicas are different objects
  - Updates can be propagated to the replica via *Active Replication* or some *Passive Replication*

    - Active updates all identically, passive has some "leader" replica
    - Both require some state machine, with multiple copies to account for replicas

- *One-Copy Serializability*: A concurrent transaction execution on a *replicated* database is OCS if equivalent to a serial execution of transactions over a single logical copy of the database
  - TL;DR: Correctness condition is *serial equivalence*
- We use two-level operation (2PC per object) and Paxos/Raft among replicas
  - Consensus needed to agree on lock acquisition and operations, or when committing transactions
- High-level: 2PC is used between replica *groups*, with Paxos/Raft used *per-group*
  - Coordinator leader sends Prepare message to leader of each group
  - Paxos used per-group to commit `prepare` to group logs (Paxos prepare-response-accept cycle)
  - On commit of `prepare`, can respond to the coordinator leader, which then uses Paxos in its group to commit the decision
  - Coordinator leader will now send a 2PC `commit` to the leader of each replica group
  - Replica group leaders use Paxos to process `commit` message, send back `commit ok` afterwards

10 Key-Value Stores

- Databases aren't great for modern usecases
  - Yes, you have like MySQL and shit w/ structured tables
  - Bad for random R/W accesses, it's unstructured, and rare `JOIN` cmds
- By contrast, KV Stores are like a dict - `get/put` model
  - Called *NoSQL Data Stores* sometimes

10.1 Distributed Hash Tables (DHT)

- *Chord* is an early popular algorithm for this shit
  - It can load-balance, is decentralized, is scalable, is always available, and is *flexible* in naming keys
- **Chord Hashing**
  - Uses IP and Port through SHA to generate a bitstring, truncates it to some 2^m bits
  - This bitstring is a "peer ID", which maps to a point on a logical circle
  - Use same ID'ing algorithm on K-V pairs – store at the key's *successor node* (inclusive)

- For lookups in Chord, we have two obvious options
  - All-to-All connections (shitty routing tables)
  - Ring succession (simple routing, but long transmission times)
  - We make a third option (which is actually used) – **fingering**
- Each Chord node keeps *finger table* w/ m entries
  - An entry i will be successor$(n + 2^i)$ for node n
  - Do some funny binary number addition bullshet to find an arb node
  - Always do the *lowest jump* that you can guarantee *won't overshoot the element*
  - Informally – *lowest finger entry that doesn't go past element*
    - If k is in the range between node and $next(k)$, just pull the next node directly
  - Funny binary number addition bullshet means lookup is $O(\log(n))$
- This all sounds well and good, but what to do for failures?
  - Need to update our fingering
  - Lookups might fail/timeout before the tables are fixed
  - Want to have a system *robust* to failures while we re-finger
- Simple solution (ish) - keep some r successor entries (can do multi-lookup)
  - Also replicate data at these r entries at all time for fault tolerance
  - Need to update successors, fingers, and keys whenever nodes fail, leave, or join!
- We maintain two invariants to keep protocol correctness even under high churn
  - Each node n correctly maintains its *direct successor* `next(n)`
  - The node `successor(k)` is responsible for key k
- *Stabilization Protocol*
  - When n joins, initialize its ring successor `next(n)` and notify them
  - If notified by a node, and your current "previous" node is *unitialized* or the pinger is between your *current* predecessor and yourself, then `prev` becomes the piner
- Each node will periodically run stabilization to determine correct node progression (`prev(next(n))`), update `next` accordingly, then ping new next node
  - Each node n also periodically updates a random finger
- Given failure detectors, we keep knowledge of r ring successors, and the predecessor of a failed node can update its ring successor
- Note that *lookups can fail while Chord stabilizes*, but the failures are *transient* and not permanent
  - Application can re-attempt, and once Chord stabilizes it will be serviced

10.2 Cloud KV Stores

10.2.1 Map/Reduce

- Need funny programming models that inherently have…
  - Fault tolerance
  - Replication and consensus
  - Cluster scheduling
  - **Map/Reduce provides this**
- **Map/Reduce** is a LISP-inspired programming model
  - Handles the creation of `map` and `reduce` via *application frameworks*
  - The application frameworks will handle cluster management via *resource managers*
  - The user supplies a `map` function to generate a list of K/V pairs from a single K/V
  - This intermediate pair gets passed into a `reduce` function, which will generate a K/V *result*
  - The set of K/V results is considered the *output* of a Map/Reduce
- You can *chain* multiple Map/Reduce pairs to create more powerful tasks
  - Can specify how many maps and reduces you have, without changing the programs
  - Inherently masks the underlying complexity for parallelism
  - Reduces should only start after *all* Maps are done
- *Map/Reduce Execution*
  1. Parallelize Map
     - Can add ”partition” values to the Map to stripe across nodes
  2. Transfer Map to Reduce w/ the shuffling
     - There should be barrier synchro to wait until it’s all ready
  3. Parallelize Reduce
  4. Implement storage for the Map I/O and Reduce I/O
     - Overall I/O is on distributed FS, but intermediate I/O is local FS

10.2.2 Job Scheduling

- Many things to do – when to do them? What order?
  - Ideally, we schedule to have good throughput, high resource utilization, and fairness
- *Some basic-ass scheduling algorithms*
  - FIFO or FCFS: lower tail completion time
  - Shortest Remaining Processing Time (SRPT) or SJF: lower average completion time

        * Generally optimal, but hard to know runtime prior to completion
        * Is technically a subcase of *priority scheduling*
    - Round Robin: better task fairness

- *Elasticity*: a job queue can exceed its resource limits if more are free/idle
- Scheduling gets harder when different jobs have *varying* multi-resource requirements
    - How to be "fair" if time is not the only variant resource?
    - UC Berkeley proposed *Dominant Resource Fairness (DRF)*
- **DRF**: For any given job, the %age of the *dominant* resource that each job (cluster-wide) is *identical* for all jobs
    - *Examples of Resources*: CPU, RAM, Network, Disk Bandwidth
    - May not always equalize if a job's demand is met and it does not need more shit, or if equations lead to weird task counts

10.2.3 Large-Scale Implementation

- Centers on analyzing *Cassandra*
- Design requirements of a large Distributed KV Store?
    - Low *total cost of operation* (TCO)
    - Fewer system administrators and incremental scalability (add more machines or more powerful machines easily)
    - Should be *fast* (high throughput, low latency)
    - Avoid single point of failure (multi-node replication)
- Overall: **high performance**, **low cost**, **scalable**
- **CAP Theorem**
    - *Consistency* means that all reads return the latest written value by *any* client
    - The system should be *available* - respond to any request on a non-failing node quickly
    - Should be *partition-tolerant* in the present of network partitions
        * *Network Partition* means that two nodes effectively cannot speak to each other
    - We can only guarantee a maximum **2** out of the 3 above properties
- If partition-tolerant, we must either be *consistent* by centralizing requests or accepting multi-node inconsistency
    - Consider data replicated across two nodes for this example
- The NoSQL explosion began as a result of the CAP tradeoff
    - CP: HBase, HyperTable, BigTable, Spanner

  - PA: **Cassandra**, RIAK, DynamoDB, Voldemort
  - CA: Non-replicated RDBMS like SQL (to some degree)

- *Cassandra* is an open-source distributed KV stores for uni and multi datacenter applications
  - Cassandra uses a *ring-based* distributed hash table (DHT), but no *finger tables* or *routing tables*
  - Each datacenter's servers are modeled as a single ring for Cassandra

- A *partitioner* is used to map keys to various servers via a hash, as well as determining primary replicas
  - Can either use *Chord* partioning or *ByteOrderedPartioner* (assign ranges of keys to servers)
  - *ByteOrderedPartitioner* is useful for range queries, for example a set of books in a certain code range

- We can also have a choice in *replication strategy*
  - *SimpleStrat*: First replica placed based on partioner, replicas then clockwise in relation to primary
  - *NetworkTopologyStrategy*: Useful for multi-DC deployments, w/ 2-3 replicas per datacenter
    * On each datacenter, you replica according to partitioner, then go clockwise until you find a different rack

- Cassandra writes are designed to be *lock-free* and *fast* (reads/disk seeks unneeded!)
  - Client sends write to 1 coordinator in the cluster (coordinator can be per-key, per-client, or per-query)
  - Coordinator usese partitioner to query all replica nodes, waits for X responses to return an `ack`
    * Can choose X to be any one, a majority, all of them, etc… depends on how consistent we need our system!

- *Hinted Handoff*: Coordinator can buffer writes for a while if a replica (or all of them) are down
  - Writes are logged in disk commit log to have failure-recovery
  - *Memtable* is an in-memory representation of KV pairs (effectively a cache searched by key, is write-back)
    * Changes are made to *memtable*, not directly to disk
  - When memtable is full/old, flush it to the disk into a *sorted string table* (SSTable) – K/V pairs sorted by key
  - SSTables will have some auxiliary lookup mechanism (like a bloom filter) to make it faster

- Data updates will acumulate over many SSTables over time – these need to all be compacted
  - *Compaction* process must run periodically per-server
- Deletions are not done right away, we instead write a *tombstone*
  - This is detected during compaction, which will trigger deletion on the underlying table
- Reads are done by contacting X same-rack replicas, prioritizing faster-response replicas when querying
  - When X respond, then the *newest-timestamped* value is returned to the client
  - Coordinator will also fetch values from off-rack replica, and will do a background consistency-check
  - *Read repair* is triggered if consistency check fails, bringing all replicas up to date *eventually*
- During a read, any given replica will check the Memtables and then the SSTables
  - If a row is split between SSTables pre-compaction, then the iterative nature will make it slower than a write
- Cassandra will elect a per-DC coordinator to coordinate data between datacenters
  - Election is *Zookeper*, a variant of the Bully algorithm
- X parameter is based on the *consistency* spectru, where relaxed consistency requirements will lead to faster R/W access times
  - Cassandra offers *eventual consistency* – if a key stop is written, then all replicas will *eventually* converge
- Cassandra segments into *consistency levels* based on the X value, client can pick its consistency level
  - *Any*: any server can be used for R/W, with coordinator caching the write and replying quickly
  - *All*: All replicas are contacted for an R/W, which is slow but consistent
  - *One*: At least one replica must be contacted, is faster than *All* but is not failure-tolerant
  - *Quorum*: Requires a quorum across all replicas in *all DCs*
    - Typically quorum is a *majority* – we require that any two quorums intersect, so you are always guaranteed to see up-to-date data
    - Faster than *All* but you still have strong consistency guarantees
    - This is what Cassandra usually uses!
- Quorum varies between *reads* and *writes*
  - **Read**
    - Client specifies R, which is at most the total number of key replicas

        * Coordinator waits for R replicas to respond before sending result
        * Coordinator checks for consistency on the other $N - R$ replicas and will read-repair if needed
    - **Write**
        * Client specifies W which is at most the total number of replicas
        * Client writes new value to all replicas, returns when it hears from all
        * This is the default strategy
- Consistency is met when $W + R > N$ and $W > \frac{N}{2}$
    - Guarantees that the R/W quorums intersect somewhere, and that two conflicting writes don't happen simultaneously
    - Can choose the W,Rbased on balance between consistency, access types, and performance!
- Cassandra clients can also pick quorums between *all replicas*, *per-DC quorum*, and quorum *only in the coordinator's DC*
    - For obvious reasons, the third is the fastest. The 2nd lets you have hierarchical replies, effectively
- Cassandra's eventual consistency guarantee means that client may receive stale data for a couple requests, but eventually system will be internally coherent
    - This works especially well when systems have low write traffic compared to reads
- Switching gears, leader election is something we need state for
    - Who is currently in our cluster (e.g who can become the new leader if a server fails?)
    - Need to update some internal membership list as servers join/leave/fail
    - Membership list is gossipped between intra-cluster nodes – nodes marked as fail if a heartbeated list is too old
- Modern KV stores promise different properties than Relational Database Management Systems (RDBMS)
    - RDBMS provide ACID, KV stores provide *Basically Available Soft-state Eventual Consistency* (BASE)
    - Modern stores prefer availability for performance over consistency
    - RDBMS like MySQL will have 300ms/350ms for R/W, while Cassandra has 15ms/0.12ms R/W