Distributed Systems

Personal notes based on lecture material and assigned papers from Princeton's first offering of COS 418: Distributed Systems, taught by Mike Freedman and Kyle Jamieson.

Table of Contents

Fundamentals

MapReduce

[MapReduce paper link]

Operation
- User program MR library divides input into M input files/tasks
- Master dispatches tasks to M map workers
    - KV pairs in input file passed to user-defined Map function
    - Intermediate KV pairs buffered in memory
    - Buffered KV pairs written to local disk
        - Partitioned into R regions by partitioning function
    - Master notified of location of buffered pairs
- R reduce workers read from M input files
    - Uses RPCs to read from local disk of map workers
    - Intermediate KV pairs grouped by/sorted by key
    - For each unique key, intermediate values passed to Reduce function
    - Output of Reduce function appended to final output fil

Programming interface
- map(k1, v1) -> list(k2, v2)
    - Map function applied to (k, v) pair, producing list of intermediate pairs
    - E.g. word count

```
map(string key, string value)
        for each w in value
                emit (w, "1");
```

- reduce(k2, list(v2)) -> list(v2)
    - Reduce function applied to (k, list(v)) pair; merges values into possibly smaller list
    - E.g. word count

```
reduce(string key, Iterator values)
        int result = 0
        for each v in value
                result += ParseInt(v)
```

emit AsString(result)

Optimizations

- Combining function [partial aggregation]
  - combine(list<key, value>) -> list<k,v>
  - Partial aggregation performed on mapper node, before data sent over network
    - E.g. <the, 1>, <the, 1>, <the, 1> becomes <the, 3>
    - Requirement: reduce() should be commutative, associative
  - Output of combiner function written to intermediate file sent to reduce task
- Partitioning function
  - partition(key, int) -> int
  - Users specify number of reduce tasks/output files desired
  - Users can provide a special partitioning function, in lieu of default
    - Default: hashing, e.g. hash(key) mod R

Fault-tolerance

- Master pings each worker periodically
  - If no response received in certain amount of time, worker marked as failed
- Map worker failure
  - Both in-progress/completed tasks marked as idle (eligible for re-scheduling)
  - Why completed outputs discarded?
    - Stored on local disk of failed machine, inaccessible
- Reduce worker failure
  - In-progress tasks marked as idle
  - Why not completed tasks?
    - Written to global file system
- Map task reassignment
  - Scenario: map task originally assigned to A is re-assigned to worker B
  - All workers executing reduce tasks are notified of re-execution
    - Any reduce tasks that hasn't read data from A will read from B
- Resilient to large-scale worker failures
  - Master simply re-executes work on available machines
- Master failure
  - Possible solution
    - Master can write periodic checkpoints of master data structures
    - If master dies, new copy can be started from last checkpointed state
  - In practice: computation aborted if master fails
    - Master failures deemed unlikely
- Failure semantics
  - Guarantee: distributed implementation produces same output as non-faulting sequential execution of entire program
  - In-progress tasks write output to private, temporary files
    - Reduce task produces 1 such file

        - Map task produces R such files (one per reduce task)
    - Task completion
        - Map worker sends message to master with name of R temporary files
        - Reduce worker atomically renames temp output file to final output file

Backup tasks

- Straggler - machine that takes unusually long time to complete a task
- Causes
    - Machine with bad disk that slows its read performance
    - Other tasks scheduled on same machine
- Mitigation
    - Master schedules backup executions on in-progress tasks (race to complete)
    - Task marked as completed when either primary, backup execution completes

Quiz questions

- MapReduce improves performance by computing partial aggregates on each node when possible, before the reducer phase. [T]
- Spark also relies on saving data to disk between each stage in order to achieve fault tolerance. [F]
- MapReduce is fault-tolerant because reducers write data to local disk as soon as mappers send it to them, without waiting to collect all data and compute the reduce function. [F]
- Stragglers in MapReduce are map or reduce tasks that run slower than others, and are solely caused by slow nodes. [F]

Network File Systems

Key idea of distributed file systems: make a remote file system look local

Stateless Network File System (NFS)

- Implemented as Remote Procedure Calls (RPCs)
- Client maintains offset into file
    - read(fh, offset, buf, n)
    - write(fh, offset, buf, n)
- File handles (fh)
    - Provided by server to client
        - Retrieved by call to fh = lookup("path", flags)
    - Includes all info needed to identify file/object on server
        - volume ID, inode #, generation #
    - Concurrency and versioning
        - Implemented via generation #s

        - Clients interact with "their" version of a file
- Challenge: achieving read-write coherence (linearizability)

Caching options

- Read-ahead - pre-fetch blocks before they are needed
- Write-through - send all writes to server
- Write-behind - buffer writes locally, send as batch
- Consistency challenges
    - How to propagate writes by one client to caches of other clients
        - Solutions: callbacks
    - How to deal with concurrent writes
        - Solutions: locking, last-writer-wins rule

Maintaining per-client state

- Pros
    - Requests can be smaller (don't need to contain offsets, etc.)
    - Processing requests is easier
    - Easier to offer cache coherence, concurrency control (e.g. by locking files)
- Cons
    - Limits scalability
    - Fault-tolerance on state required for correctness
        - Crash recovery is more difficult
    - Open/close interface needed

Kinds of state

- Hard state
    - Needs to be maintained for correctness
    - Can be written to disk, cold remote backup
- Soft state
    - Performance optimization
    - Can be lost at will, but recovery may impact availability (liveness)

Network File System (NFS) history

- Stateless protocol
- Data read from server, cached in NFS client
- NFSv2 was write-through (i.e. synchronous)
- NFSv3 was write-behind
    - Writes delayed until close, fsync received from application

Consistency tradeoffs

- Write-to-read semantics (strong consistency)
    - Goal: ensure reads see data from most recent write
    - Implications: can't cache, need server-side state

    - Verdict: too expensive
- Close-to-open "session" semantics (weaker consistency)
    - Goal: ensure ordering between application close() and open()
        - If B opens after A closes, B should see A's writes
    - If two clients open at the same time, no guarantees
        - Last writer wins rule for conflicting writes

NFS cache consistency

- Challenge: potentially concurrent writers
- Cache validation
    - Clients get file's last modification time from server, via getattr(fh)
    - After that, clients poll every 3-60 seconds
        - If server's last modification time has changed...
            - Clients flush dirty blocks and invalidate cache
    - Reading a block
        - Validate: currentTime - lastValidationTime < threshold
        - If valid, data served from cache
        - Otherwise, data refreshed from server

Problems

- "Mixed reads" across version
    - A reads blocks 1-10 from file, B replaces blocks 1-20, A keeps reading 11-20
- Synchronized clocks assumption
    - Not correct - need logical clocks
- Writes specified by offset
    - Concurrent writes can change offset - need OT, CRDTs

Locks

- Client requests lock over a file / byte range
- Client performs writes, then unlocks
- Lock release
    - Well-behaved clients comply (automatically releases when done)
    - Server forcibly reclaims, if necessary
- Problem: what if client crashes
    - Solution: keep-alive timer maintained; lock recovered on timeout
- Problem: what if client is alive, but network route failed
    - Implication: client thinks it still has lock, but server reassigns (split-brain)

Leases

- Clients obtain lease on file for reads/writes
    - Has pre-defined expiration time (unlike a lock)
    - Has file version number (for cache coherence)
- Read lease - allows client to cache clean data

    - Guarantee - no other client is modifying file (no write lease issued)
- Write lease - allows safe delayed writes
    - Client can modify locally, make batched writes to server
    - Guarantee - no other client has file cached (no read, write lease issued)
- Conflicts - server sends eviction notices
    - Evicted write lease - owner must write back
    - Evicted read lease - owner must flush/disable caching
    - Client sends acknowledgement when complete

Recovery with leases

- Before lease expires, client must renew lease
- If client fails while holding lease
    - Server waits until expiration, then reclaims
- If client fails during eviction process
    - Server waits until eviction timeout, then reclaims
- If server fails with leases outstanding, on recovery
    - Server waits lease period + clock skew before issuing new leases
    - Server absorbs renewal requests and/or writes for evicted leases

Andrew File System

- Key design goal: scalability
    - Many servers, 10,000s of users
- Workload observations
    - Reads much more common than writes
    - Concurrent writes are rare
- Files, not blocks, as interface unit
    - Whole-file serving - entire files and directories served
    - Whole-file caching - clients cache files to local disk
- Consistency model
    - Close-to-open consistency
        - No mixed writes - precluded by whole-file caching, whole-file overwrites
        - Update visibility - callbacks used to invalidate caches
    - Crashes and partitions
        - Client invalidates cache if and only if
            - Recovering from failure
            - Regular liveness check (heartbeat) to server fails
        - Server assumes cache invalidated if
            - Callbacks fail and heartbeat period is exceeded

Quiz questions

- All versions of NFS server maintain state. [F]
- If a read lease is given to a client then no other client can read the file. [F]
- If a read lease is given to a client then no other client can *modify* the file. [T]

- If a write lease is given to a client then no other client has cached the file. [T]
- If a write lease is given to a client then no other client can modify the file locally. [T]

Network Communication and RPCs

RPC goals
- Enable programmer to write single-threaded, client-server code
- Make communication appear like local procedure call
    - Callee pushes arguments onto stack
    - Callee reads arguments from stack

RPC issues
- Heterogeneity
    - Server needs to expose an interface to the client
- Failure
    - Messages could get dropped
    - Client, server, or network could fail
- Performance
    - Much slower than normal procedure call (especially if network involved)

Heterogeneity
- Problem: differences in data representation
    - Data types represented differently, byte ordering (endianness), floating point representation, data alignment requirements
- Problem: differences in programming support
    - Some programming languages naturally support RPCs, others don't
- Solution: interface description language
    - Goal: pass procedure parameters/return values in machine-independent way
    - Programmer writes interface description in IDL
        - Defines API for procedure calls: names, parameters/return types
    - Programmer runs IDL compiler, which generates
        - Code to marshall native data types into machine-ind. byte streams

RPC steps
- Client process calls client stub function (in RPC library) with RPC arguments
- Client stub packages parameter into a network message, forwards to client OS
- Client OS sends a network message to server
- Server OS receives message, sends up to server stub
- Server stub unpacks parameters, calls server function
    - Dispatcher - receives client’s RPC request, identifies appropriate method to call

    - Skeleton - unmarshalls parameters to server-native types, calls local server procedure, marshals server response, sends response to dispatcher
- Server function runs, returns a value
- Steps repeat in other direction to return value to client

Server stub components
- Dispatcher
    - Receives client's RPC request
    - Identifies appropriate server-side method to invoke
- Skeleton
    - Executes local server procedure

Failure scenarios
- Packets may be dropped, due to normal packet loss/network failure
- Server may crash and reboot
- Network or server might be slow

At-least-once scheme
- Client re-sends request, after timeout, if no ACK received
- Returns error if no response after several retries
- Problem: operations can be executed multiple times or out of order
- When okay
    - Read-only operations, with no side-effects
    - Application can handle duplication and reordering

At-most-once scheme
- Client includes unique transaction id (xid) with each RPC request
    - Same xid used for retransmitted requests
- Ensuring transaction id uniqueness
    - Combine unique client id (e.g. IP address) with timestamp
    - Combine unique client id with sequence number
        - Log sequence number to persistent storage for crash recovery
    - Use a big random number
- Discarding server state
    - Problem: seen and old arrays grow without bound
    - Observation: once client has gotten ACK for xid, it will never be re-sent
    - Client includes "seen all replies <= X" with every RPC
        - Newer message subsume older ones
- Concurrent requests
    - Problem: how to handle duplicate request, while original is still executing
    - Solution: add pending flag when RPC starts executing
        - If pending flag set, server waits for procedure to finish, or ignores
- Server failure

    - Problem: server may crash and restart, losing in-memory state
    - Solution: server writes old[], seen[] arrays to disk

RPCs in Go

- Go's net/rpc has at-most-once semantics
- Characteristics
    - Server's TCP receiver filters out duplicate messages
    - RPC code doesn't retry automatically
- Returns an error if no reply received
    - After a TCP timeout
    - If server didn't see request
    - If server processed request, but server/net failed before reply came back

Exactly-once scheme

- Combines retransmission (at-least-once) and duplicate filtering (at-most-once)
- Surviving client crashes
    - Client records pending RPCs on disk
    - Allows client to replay RPCs on restart with same xid
- Surviving server crashes
    - Server logs results of completed RPCs to disk (old array)
    - Allows server to return response without re-executing operation
- Can't achieve in general
    - Server could crash between logging an operation and executing it
    - Can't tell which happened, though can make window very small

Quiz questions

- The primary goal of RPCs is to improve a distributed application's performance. [F]
- A client can safely re-issue an at-least-once RPC that has timed out if the operation is not idempotent. [F]
- RPCs behave identically to local function calls from the perspective of the calling application. [F]
- In an at-most-once RPC scheme that uses a randomly-chosen integer for its transaction id (xid), the server may execute the operation more than once. [F]
- In an "at-most-once" RPC scheme whose xid is a tuple of a local monotonically-increasing counter combined with the client's (assumed unique) IP address, the server may execute the operation more than once. [F]

Time Synchronization and Logical Clocks

Time synchronization challenges

- Internet is asynchronous - arbitrary message delays

- Internet is best-effort - messages don't always arrive

Synchronization to time server
- Client issues RPC to server with accurate time to obtain time
- Problem: network latency - message contains outdated server answer

Cristian's algorithm
- Steps
  - Client sends request packet, timestamped with local clock T_1
  - Server timestamps receipt of request T_2 with its local clock
  - Server sends response packet, timestamped with local clock T_3
  - Client locally timestamps receipt of server's response with T_4
- Offset calculation
  - Client computes RTT = $\delta = (T_4 - T_1) - (T_3 - T_2)$
  - Client sets clock to $T_3 + \frac{1}{2}\delta$

Berkeley algorithm
- Distributed algorithm for timekeeping
- Motivation: single time server can fail, blocking timekeeping
- Assumptions
  - All machines have equally-accurate local clocks
- Steps
  - Master machine polls L other machines via Cristian's algorithm
  - Computes average time across all servers
  - Sends all servers difference between average and received time

Network Time Protocol (NTP)
- Goal: allow clients to accurately synchronize to UTC, despite message delays
- Guarantees
  - Reliable service - survives lengthy losses of connectivity
  - Accurate service - leverages heterogeneous accuracy in clocks
- System structure
  - Servers (time sources) arranged in layers
  - Stratum 0: high-precision time sources (e.g. atomic clocks, radio time receivers)
  - Stratum 1: NTP servers directly connected to Stratum 0
  - Stratum 2: NTP servers that synchronize with Stratum 1
  - Stratum 3: NTP servers that synchronize with Stratum 2
  - Users synchronize with Stratum 3 servers
- Server selection
  - Messages between NTP client and server exchanged in request-response pairs
    - Cristian's algorithm used
  - For i^{th} message exchange with particular server, client computes
    - Clock offset $T_3 + \frac{1}{2}\delta - T_4$

        - Round trip time RTT = δ
    - Client chooses server with minimum dispersion in RTT
    - Client adjusts its clock by offset θ_0 corresponding to minimum RTT
- Update scheme
    - Don't want time to run backwards
    - Change update rate for clock
        - Prevents inconsistent local timestamps

Clock synchronization takeaways

- NTP, Berkeley clock synchronization
    - Rely on timestamps to estimate network delays
    - Clocks not exactly synchronized
- Often inadequate for distributed systems
    - Need to reason about order of events
    - May need precision on order of ns

Logical time

- Key idea: only ordering ("happens-before" relation) of events matters
- When does event **a** "happen before" event **b**
    - If **a** and **b** are observed at a single process and **a** occurs before **b**
    - If **b** is the receive event for a message whose send event is **a**
        - Messages are assumed to take finite time to send
    - Via transitivity - if **a** happens before **c** and **c** happens before **b**
- If neither **a**→**b** nor **b**→**a**, then **a** and **b** are concurrent (**a** || **b**)

Lamport clocks

- Goal: clock time C(**a**) such that if **a**→**b** then C(**a**) < C(**b**)
- Each process P_i maintains local clock C_i
- Before executing an event
    - Increment local clock $C_i \leftarrow C_i + 1$
    - Set event time C(**a**) $\leftarrow C_i$
    - Add local timestamp to message C(**m**) $\leftarrow C_i$
- On receiving a message **m**
    - Set local clock C_j, receive event time $\leftarrow 1 + \max\{C_j, C(\mathbf{m})\}$
- Break ties by appending process number to each event
    - Process P_i timestamps event **e** with $C_i(\mathbf{e}).i$
    - Result: C(a).i < C(b).j when either:
        - C(a) < C(b)
        - C(a) = C(b) and i < j
- Implication
    - No two events have the same Lamport timestamp
    - Can definite total ordering $C(\mathbf{a}_1) < C(\mathbf{a}_2) < \ldots < C(\mathbf{a}_m)$ on all events
- Issue

    - By construction, **a**→**b** implies C(**a**) < C(**b**)
    - Can't use Lamport timestamps to infer causal relationships between events
        - C(**a**) < C(**b**) does not imply **a**→**b** (possibly **a || b**)

Totally-ordered multicast

- Motivation: multi-site database replication
    - Client sends query to nearest copy
    - Client sends update to both copies
- Client sends update to one replica
    - Update contains Lamport timestamp C(x)
    - Replica places event in local queue, sorted by increasing C(x)
- Steps
    - On receiving an event from a client
        - Broadcast it to all others
    - On receiving or processing an event
        - Add it to local queue
    - For received events at head of queue
        - Broadcast an ACK message to every process (including self)
    - On receiving an acknowledgement
        - Mark corresponding event as ACKed (for process) in queue
    - For any event at head of queue
        - Remove and process if everyone has ACKed
- Ensures
    - Events executed in same order at all replicas (in order of increasing C(x))
    - Events only executed if ACKed (received, at head of queue) by all replicas

Vector clocks

- Each event **e** is labelled with vector V(**e**) = [c_1, c_2, c_3, …]
    - c_i is the count of events in process i that causally precede **e**
- Update rules
    - For a local event at process i, increment local entry c_i
    - If process j receives message with vector [d_1, d_2, d_3, …]
        - Set each entry c_k = max{c_k, d_k}
        - Increment c_j
- Comparing vector clocks
    - Define V(**a**) = V(**b**) if $a_k = b_k$ for all k
    - Define V(**a**) < V(**b**) if $a_k \leq b_k$ for all k and V(**a**) ≠ V(**b**)
    - Otherwise say **a** || **b**
- Implications
    - If V(**a**) < V(**b**) then **a**→**b**
        - Chain of causal events connects a and b
    - If neither V(**a**) < V(**b**), nor V(**b**) < V(**a**), then **a** || **b**

Causally-ordered bulletin board

- Want: no user sees reply before corresponding original message post
- How to achieve
    - Process event handler delivers message only if:
        - All causally preceding messages have been delivered
    - Example: reply receive event has VC = (1, 1, 0)
        - Not delivered until (1, 0, 0) received

[Lamport's original paper]

Quiz questions

- Most clock synchronization protocols only need to synchronize servers once, when they begin operation. [F]
- One cannot determine the one-way latency of a packet from one server to another. [T]
- In the NTP algorithm, a server's clock time may be set backwards. [F]

Fault-Tolerance

Primary Backup

Primary-backup goals

- Replicate and separate servers
- Goal 1 - fault-tolerance - provide a highly reliable service
    - Despite server and network failures
- Goal 2 - abstraction - have servers behave like single, more reliable server

State machine replication

- Any server is a state machine
    - Holds set of (key, value) pairs
    - Operations transition between states
- All-or-nothing atomicity
    - Operation execute on all replicas, or none at all
- Key assumption: operations are deterministic
    - Relaxed in VMWare FT protocol

Challenges

- Network and server failures
- Network partitions

Primary-backup approach

- Clients send all operations (get, put) to current primary
- Primary orders clients' operations
    - Should be only one primary at a time
- Steps
    - Primary logs operation locally
    - Primary sends operation to backup and waits for ACK
        - Backup performs op and just adds it to the log
    - Primary performs operation and ACKs to client

View server

- Decides who is primary and who is backup
    - Clients and servers depend on this decision
- Challenge: only want one primary at a time
- Assumption: view server never fails

Monitoring server liveness

- Each replica periodically pings view server
    - View server declares replica dead if it missed N pings in a row
    - Replica considered alive after a single ping
- Can an alive replica be declared dead by view server?
    - Yes - in the case of a network failure/partition

Agreeing on current view

- Any number of servers can ping view server
- Want everyone to agree on view number
    - View number included in RPCs between parties
- Okay to have a view with a primary, but no backup

Transitioning between views

- Ensuring a primary has up-to-date state
    - Only a previous backup is promoted (not previously-idle server)
    - Set liveness detection timeout > state transfer time
- View server: checking that a backup is up-to-date
    - View server sends view-change message to all servers
    - Primary must ACK new view once backup is up-to-date
        - View server stays with current view until ACK
        - Even if primary has or appears to have failed

Split-brain situation

- Server(s) become disconnected from the view server, resulting in disagreement about who is the primary and who is the backup
- Can result in improper handling of client requests

State transfer protocol

- Question: How does a new backup (in view i) get the current state?
    - Primary (from view i-1) transfers state
- Operation log
    - Entire operation log transferred
- Snapshot
    - Every op falls either before or after a state transfer
    - If op before transfer, transfer must reflect op
    - If op after transfer, primary forwards op to backup after transfer concludes

Rules

- View i's primary must have been a primary/backup in view i-1
- Non-backup must reject forwarded requests
    - Backup only accepts forwarded requests if view number matches its view
- Non-primary must reject direct client requests
- Every operation must be before or after state transfer

Primary-backup summary

- Idea: first step in making stateful replicas fault-tolerant
- Allows replicas to provide continuous service despite net, machine failures
- Finds repeated application in practical systems

[VMWare FT protocol paper link]

Basic VMWare vSphere Fault Tolerance (VM-FT) protocol design

- Replication - backup VM executes same operations are primary VM, with small time lag
    - Includes non-deterministic operations
    - Outputs of backup VM are dropped
- Logging channel - all events received by primary sent to backup VM via logging channel
    - Includes input events
    - Includes results of non-deterministic instructions (e.g. timestamp counter read)
    - Includes output events (must be ACKed by backup)
- Shared disk - primary and backup VM's virtual disks run on shared storage
- Output requirement - if the backup VM takes over after a failure of the primary (failover), it must execute in a way that is consistent with the outputs sent out by the primary VM
    - Output rule - primary VM cannot send an output to the external world, until the backup VM has received and acknowledged the log entry for the output event

Exactly-once semantics

- Backup cannot know if primary crashed before/after sending last output (without 2PC)
- Possibilities
    - Incoming packet to primary could be lost (primary fails just after receiving)
    - Outgoing packet could be duplicated (backup doesn't know if already sent)

- Handling
    - Network infrastructure (TCP) handles lost, duplicate packets

Detecting/responding to failure

- If backup VM fails, primary VM "goes live"
    - Leaves recording mode (stops sending entries on logging channel)
    - Starts executing as normal VM (executes without waiting for ACKs)
- If primary VM fails, backup VM is promoted to a primary, then goes live
    - Must consume log entries that have been received/acknowledged
    - Stops replaying mode, starts executing as normal VM
- Failure detection
    - Primary and backup each run UDP heartbeating, monitor logging traffic/ACKs
    - Failure declared if heartbeating or logging traffic stops for longer than a timeout
- Split-brain issue (avoiding two primaries)
    - Logging channel may break - both servers think other has failed
    - Shared storage between virtual disks used to ensure only one VM goes live
    - Atomic test-and-set operation executed on shared storage
        - If succeeds, VM allowed to go live
        - If fails, VM halts itself (assumption: other VM already went live)
        - If cannot access, VM waits

Quiz questions (VM-FT protocol)

- Failover may result in the loss of a network packet the FT VM would have received from the network. [T]
- Failover may result in the duplication of a network packet the FT VM receives from the network. [F]
- Failover may result in the duplication of a network packet the FT VM sends to the network. [T]
- FT protocol delays any output operation until the backup acknowledges it. [F]

Two-Phase Commit

Goal: fault tolerance - building reliable systems from unreliable components

Safety and liveness

- Safety - bad things don't happen; no stopped or deadlocked states; no error states
    - Mutual exclusion - two processes don't enter a critical section at the same time
    - Bounded overtaking - a process can't overtake another process into a critical section more than once

    - Important in: banking transactions
- Liveness - good things happen eventually
    - Starvation freedom - process 1 can eventually enter critical section, as long as process 2 terminates
    - Eventual consistency - if a value in an application doesn't change, two servers will eventually agree on its value
    - Important in: social networking sites

Two-phase commit

- Goal: general purpose, distributed agreement on some action, with failures
- Example: transfer money from A to B
    - Debit at A, credit at B, tell client "okay"
    - Require: both banks make transfer, or neither
- Participants
    - Client, transaction coordinator (TC), servers (A, B)
- Steps
    - Client -> TC: "start"
    - TC -> A, B: "prepare"
    - A,B -> TC: "yes" or "no"
    - TC -> A, B: "commit" / "abort"
        - TC sends commit if both say yes
        - TC sends abort if either say no
    - TC -> C: "okay" (commit) or "failed" (abort)
    - A, B commit on receipt of commit message
- Reasoning
    - Neither can commit unless both agree to commit
    - Performance - timeouts, reboot
- Timeouts
    - TC waits for "yes"/"no" from A and B
        - No commit messages sent, so can safely abort on timeout
    - A and B wait for "commit"/"abort" from TC
        - If sent no, can safely abort
        - If sent yes, pings other servers
- Server termination protocol
    - Scenario: server B (voted "yes") is waiting for commit/abort from TC
    - B -> A: "status"
        - No reply from A - no decision, B waits for TC
        - Server A received commit/abort from TC - agree with TC decision
        - Server A hasn't voted/voted no - both abort
        - Server A voted yes - both must wait for TC
            - TC could have decided either to commit or abort
    - Safety/liveness
        - Safety - guaranteed correctness for some timeout situations

        - Liveness - A and B could block, due to failure of TC or network to TC
- Crash and reboot
    - Scenarios
        - TC crashes just after sending "commit"
        - A or B crash just after sending "yes"
    - Write-ahead log used to record "commit"/"yes" decisions to disk
- Recovery protocol
    - If everyone rebooted and is reachable
        - TC checks for commit record on disk and resends action
    - TC: If no **commit** record found on disk, abort
        - No "commit" message was sent, so safe to abort
    - A,B: If no **yes** record found on disk, abort
        - Didn't vote "yes", so TC couldn't have committed
    - A,B: If **yes** record found on disk, execute server termination protocol
        - Potentially blocking
- Properties
    - Safety - all hosts that decide reach the same decision
    - Liveness - if no failures and all say "yes", then commit
        - But if failures, 2PC can block
- Evaluation
    - Doesn't tolerate faults well - has to wait for servers to reboot
- Failure scenarios
    - Participant fails before sending response
        - TC times out and aborts
    - Participant fails after sending yes vote
        - TC decides to commit
        - Failed participant commits on restart
    - Participant votes yes, but vote is lost enroute to TC
        - TC times out and aborts
        - Participants pings TC about decision, learns about abort
    - TC fails before sending prepare
        - On restart, TC sends out prepare messages again
    - TC fails after sending prepare
        - On restart, TC sends out prepare messages again
    - TC fails after receiving votes
        - On restart, TC sends out prepare messages again
            - Participants vote again
    - TC fails after sending decision
        - Decision is logged to disk
        - On restart, participants that didn't receive decision ping TC
            - TC resends decision
    - TC fails before making decision
        - If both voted yes, participants can decide to commit

Quiz questions

- Two-phase commit blocks unless the transaction coordinator is alive and reachable. [T]
- Two-phase commit cannot complete a new transaction if a server being coordinated fails. [T]

FLP Impossibility and Paxos

Consensus properties

- Termination - all non-faulty processes eventually decide on a value
- Agreement - all processes that decide do so on the same value
- Validity - value that has been decided must have been proposed by some process

System model

- Network model
    - Synchronous (time-bounded) or asynchronous (arbitrary delay)
    - Reliable (in-order, intact message delivery) or unreliable communication
    - Unicast or multicast communication
- Node failures
    - Fail-stop (correct/dead) or Byzantine (arbitrary)

FLP result

- System model
    - Asynchronous network
    - Reliable communication
    - Unicast communication
    - Fail-stop failures
- No deterministic, 1-crash-robust consensus algorithm exists for asynchronous model
- Holds even for
    - Weak consensus (only some process needs to decide, not all)
    - Two states (0 and 1)
- Proof approach
    - Consider initial states of k processes in system (0 or 1) and system decision
    - Sort state tuples by decision
        - Some tuple of states must have a "neighbor" with different decision
            - Neighbor: tuple with exactly one process with different state
        - If differing process has failed, nothing can distinguish two state tuples
            - Bi-valent states - both decisions possible
- Adapting to multiple failures
    - Wait until process recovers before handling new failures
- Circumvention methods

    - Probabilistically
    - Randomization
    - Partial synchrony (failure detectors)

Paxos

- Safety
    - Agreement - only a single value is chosen
    - Validity - only a proposed value can be chosen
    - Only chosen values are learned by processes
- Liveness
    - Termination - some value eventually chosen if fewer than half processes fail
    - If a value is chosen, a process eventually learns it
- Process roles
    - Proposers - propose values
    - Acceptors - accept values, which are chosen if the majority accepts
    - Learners - learn outcome (chosen value)
- Proposals
    - Tuple of (proposal #, value) = (n, v)
        - Proposal # strictly increasing
        - Globally unique: lower-order bits set to proposer's ID
    - Acceptors accept multiple proposals
- Algorithm pseudocode (from MIT 6.824)
- Proposers
    - proposer(v)
        - Choose proposal number n > n_p
        - Send prepare(n) to all servers including self
        - If prepare_ok(n, n_a, v_a) received from majority
            - If no n_a returned, choose v' = v
            - Otherwise, choose v' = v_a corresponding to highest n_a
            - Send accept(n, v') to all
                - If accept_ok(n) received from majority
                    - Send decided(v') to all
- Acceptors
    - State
        - n_p - highest prepare seen
        - n_a, v_a - high accepted proposal number/value
    - prepare(n) handler
        - If n > n_p # accept if higher than any seen
            - Set n_p = n
            - If no prior proposal accepted
                - Reply prepare_ok(n, _, _)
            - Otherwise
                - Reply prepare_ok(n, n_a, v_a)

    - accept(n, v) handler
        - If n >= n_p
            - n_p = n_a = n
            - v_a = v
            - Notifies learners of accepted value
            - Reply accept_ok(n)
- Learners
    - Approach 1
        - Each acceptor notifies all learners
        - More expensive
    - Approach 2
        - Elect "distinguished learner"
        - Acceptors notify distinguished learner, which informs others
        - Failure-prone (single point of failure)
- Comparison to 2PC
    - 2PC can be viewed as a special case of Paxos
        - TC is the only proposer
        - Nodes are all acceptors
        - All acceptors must accept to choose value
    - Improvement: much less likely to block
        - Doesn't hinge on single proposer (TC) being alive
        - But still can block!
- Key idea
    - If proposal with value v decided (accepted by majority of acceptors), then any higher-numbered proposal issued by any proposer has value v
        - prepare_ok for higher proposal will return accepted value
        - Proposer will echo accepted value in its accept(n, v) request
- Liveness issue
    - If accept requests are continually rejected, Paxos may not reach consensus
- Paxos with leader election
    - Overview
        - Each process plays all three roles
        - First proposer whose proposal is accepted deemed leader
            - If elected proposer can communicate with a majority, protocol guarantees liveness
        - Paxos can tolerate failures of f nodes, f < N/2
    - Operation
        - Leader election used to decide transaction coordinator
        - Problem: could be subject to split-brain issues
        - Solution: leases
            - New leaders wait before leases of old (possibly partitioned) leaders expire before accepting any new operations

Quiz questions (Paxos)

- During the prepare phase, servers find out about the chosen value. [F]
- Paxos with leader election can have more than one leader at any one time. [F]
- Different acceptors can accept different values. [T]
- Is Paxos guaranteed to reach consensus? [F]
- Leases can prevent split-brain issues. [T]

Paxos FAQs

- How does Paxos sidestep the FLP result?
    - Paxos doesn't guarantee liveness, so it achieves a kind of conditional consensus that is not precluded by FLP
- How does multi-Paxos elect a leader?
    - First proposer whose proposal is accepted is deemed leader
- How does multi-Paxos solve the liveness issue of normal Paxos?
    - All subsequent proposals must come from the leader, so multi-Paxos eliminates races between proposers
- Why must an higher-numbered accepted proposal echo an accepted value? While it is true that the new proposer is guaranteed to receive the accepted value from at least 1 node (due to quorum intersection), why couldn't it also receive another value corresponding to a higher accepted proposal number from some other node?
    - Can apply argument to the supposedly received higher accepted proposal number and the accepted value. Their quorums must have intersected in 1 node, and so on, until accepted value must have been replayed.

[Lamport's Paxos Made Simple paper link]

Replicated State Machines and Raft

State machine replication

- Any server is a state machine
    - Operations cause transitions between states
- Need: distributed, all-or-nothing atomicity for operations

Problems with primary-backup

- Network partition - backup cannot be promoted, view server needed
- View server failure - need distributed "view" determination
- Availability - don't want to block client until operation executed on *all* servers
    - Declare op to be stable if written on majority of backups
    - Can expect success, as replicas are identical

View changes on failure

- Backups monitor primary
- If backup thinks primary failed, initiates view change (leader election)
    - Safety argument
        - View change requires agreement of f+1 nodes out of 2f+1
        - Op committed once written to f+1 nodes
        - At least one node sees both the write and the new view
            - f+1 and f+1 nodes overlap in 1 node

Basic Replicated State Machine (RSM) approach

- Run consensus protocol to elect a leader
- Run 2PC to replicate operations from the leader
- Have replicas excute ops once committed

Benefits of having a leader

- Decomposition - normal operation vs. leader changes (easier to reason about)
- Eliminates source of conflicts
- More efficient than leaderless approaches (e.g. Paxos)
- Isolates non-determinism

[Raft paper link]

Server states

- Leader, follower, candidate
- Normal operation: 1 leader, N-1 followers

Liveness validation

- Servers start as followers
- Leaders send heartbeats (empty AppendEntries RPCs) to maintain authority
- If election timeout expires, follower assumes leader crashed and starts new election

Terms (epochs)

- Terms begin with election, continue with normal operation under a leader
- Each server maintains current term value
- Key role: identify obsolete information

Elections

- Start - increment current term, change to candidate, vote for self
- RequestVote RPCs sent to all other servers
- Election retried until
    - Yes votes received from majority of servers
    - RPC received from valid leader
    - No one wins election
- Properties

    - Safety - at most one winner allowed per term
        - Each server votes only once per term (decision persisted on disk)
        - Two different candidates can't get majorities in the same term
    - Liveness - some candidate must eventually win
        - Each candidate chooses timeouts randomly in [T, 2T]
        - If T >> network RTT, with high probability, one will win election

Log structure

- Log entry = <index, term, command>
- Log stored on stable storage (disk) to survive crashes
- Entry committed (durable/stable) if known to be stored on majority of servers
    - Will eventually be executed by state machines

Normal operation

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once entry is committed
    - Leader executes command (on its state machine)
    - Leader sends result to client
    - Leader notifies followers of commitment in later AppendEntries
    - Followers execute committed commands
- Crashed/slow followers?
    - Leader retries RPCs until they succeed
- Performance optimal in common case
    - One successful RPC sent to any majority of servers

Log operation

- Coherence
    - If long entries on different servers have same index, term
        - Command stored is identical
        - Log identical in all preceding entries
    - If given entry is committed, all preceding entries committed
- Consistency check
    - AppendEntries carries <index, term> of entry preceding new ones
    - Follower must contain matching entry; otherwise, RPC rejected
    - Implements induction step, ensuring coherency

Leader changes

- New leader's log is truth
    - Will eventually make follower's logs identical to its own
- Old leaders have have left entries partially replicated
    - Multiple crashes can leave many extraneous log entries

Safety requirement

- Goal: once log entry applied to a state machine, no other state machine must apply different value for that log entry
- Raft safety property - if leader has decided a log entry is committed, entry will be present in the logs of all future leaders
- Claim: Raft safety property ensures safety requirement
    - Leaders never overwrite entries in their logs
    - Only entries in leaders' log can be committed
    - Entries must be committed before being applied to state machines
- Ensuring committed entries show up in future leaders' logs
    - Restrictions on commitment (replicated on majority, from current term)
    - Restrictions on election (majority vote, most complete log)

Picking the best leader

- Leader's log should be at least as up-to-date as voter's
- Implication: leader has most complete log among election majority

Committing entries

- Case 1 - leader decides entry in current term is committed
    - Leader for next term must also contain that entry
- Case 2 - leader trying to finish committing entry from earlier
    - Not safely committed, unless entry from current term also on majority

Challenge - log inconsistencies

- Leader changes can result in log inconsistencies
- New leader must make follower logs consistent with its own
    - Delete extraneous entries
    - Fill in missing entries
- Leader keeps nextIndex for each follower
    - Index of next entry to send to that follower
    - Initialized to 1 + leader's last index
    - If AppendEntries consistency check fails
        - Decrement nextIndex and try again

Neutralizing old leaders

- Leader temporarily disconnected
    - Other servers elect new leader
    - Old leader is reconnected
    - Old leader attempts to commit log entries
- Term numbers used to detect stale leaders, candidates
    - Every RPC contains term of sender
    - Sender's term < receiver's term

        - Receiver rejects RPC
    - Receiver's term < sender's term
        - Receiver reverts to follower, updates term, process RPC
- Election updates terms of majority of servers
    - Deposed server cannot commit new log entries

Client protocol

- Client sends commands to leader
    - If leader unknown, can contact any server, which redirects to leader
- Leader responds after it has logged, committed, and executed the command
- If request times out (e.g. due to leader crashes)
    - Client reissues command to new leader (after possible redirect)
- Exactly-once semantics
    - Leader can execute command, then crash before responding to client
    - Solution: client embeds unique id in each command
        - Client id is included in log entry
    - Before accepting request, leader checks if id contained in log

Configuration changes

- View configuration: {leader, {members}, settings}
- Consensus must support changes to configuration
    - Replace failed machine
    - Change degree of replication
- Cannot switch directly from one config to another
    - Conflicting majorities could arise

Two-phase approach (via joint consensus)

- Joint consensus in intermediate phase
    - Need majority of both old, new configurations for elections, commitment
    - Any server from either configuration can serve as leader
        - If leader not in C_{new}, must step down once C_{new} committed
- Configuration change - log entry
    - Applied immediately on receipt (committed or not)
- Once joint consensus committed, can begin replicating log entries for new config

Viewstamped replication

- Primary copy method to support highly-available distributed systems
- Comparison to Raft
    - Strong leader
        - Log entries only flow from leader to other servers
        - Leader selected from limited set, so no catch up required
    - Leader election
        - Randomized timers used to initiate elections

    - Membership changes
        - New joint consensus approach used, with overlapping majorities
        - Cluster can operate normally during configuration changes

Byzantine Fault Tolerance

Byzantine faults

- Node/components fails arbitrarily
    - Performs incorrect computation
    - Gives conflicting information to different parts of system
    - Colludes with other failed nodes

Fail-stop failures

- Examples
    - Node crashes
    - Network breaks or partitions
- Consensus protocols
    - State machine replication with N=2f+1 replicas can tolerate f failures

Using Paxos for BFT

- Can't rely on primary to assign sequence numbers
    - Could assign same sequence number to different requests
- Can't use Paxos for view change
    - Intersection of two majority quorums (f+1 nodes each) may be bad node
    - Bad node can tell different quorums different things
        - Accepts value val1 proposed by node in N0
        - Proposes value val2 to node in N1 for acceptance
            - Doesn't follow prepare_ok rule to replay accepted values

Client certification

- Clients sign input data before storing it
- Clients verify signatures on data retrieved from service
- Problems
    - Replay attack - Byzantine nodes can replay stale, signed data in their response
    - Inefficient - clients have to perform computations, sign data

[Practical BFT paper link]

Practical BFT

- 3f+1 replicas used to survive f failures (proven to be minimal)
- Requires three phases

- Provides state machine replication

Correctness argument
- Operations are deterministic
- Replicas start in same state
- If replicas execute same requests in same order, will produce identical results

Client failures - non-problem
- Clients **can't** cause internal inconsistencies in the data in the servers
    - State machine replication property
- Clients **can** write bogus data to the system
    - System should authenticate clients, separate data just like any other datastore

Client operation
- Send requests to primary replica
- Wait for f+1 identical replies
    - Note: replies may be deceptive
        - Replica could return the correct answer, but do otherwise locally
    - Ensures >= 1 reply actually from non-faulty replica

Replica operation
- Carry out protocol that ensures:
    - Replies from honest replicas are correct
    - Enough replicas process each request to ensure that
        - Non-faulty replicas process same requests in the same order

Primary-backup protocol
- Group runs in a view
- View number designates primary replica
    - Primary is the node whose id (modulo view #) = 1

Ordering requests
- Primary picks request ordering
    - Note: primary might be a liar
- Backups ensure primary behaves correctly
    - Check and certify correct ordering
    - Trigger view changes to replace faulty primary

Byzantine quorums
- A collection of >= 2f+1 replicas
    - An op's quorum overlaps in f+1 nodes with the next op's quorum
        - Why? Total 3f+1 replicas
    - Implication: overlap must contain >= 1 non-faulty replica

- Quorum certificate
    - Collection of 2f+1 signed, identical messages from Byzantine quorum

Keys

- Symmetric-key cryptography used to encrypt messages
    - Bootstrapped using public-private keypairs owned by each client, replica
- Each client, replica owns separate keys for
    - Sending messages
    - Receiving messages

Ordering requests

- Primary chooses request's sequence number (n)
    - Sequence number determines order of execution
- Backups locally verify when they've seen <= 1 client request for seq number n

Operation

- Pre-prepare phase (check primary's message)
    - Primary receives signed request m from client
    - Primary chooses a sequence number seq(m) = n for request
    - Primary sends message (“pre-prepare”, view, n, Digest(m)) to all backups
        - Primary could be sending different messages to all backups
    - If ≤1 client request seen with seq number n, backup starts prepare phase
- Prepare phase (collect prepared certificate)
    - Backup i multicasts (“prepare”, v, n, D(m), i) to all other replicas
    - A replica accepts a prepare message if
        - Signatures are correct
        - View number v matches replica’s current view number
        - Sequence number n is valid (between global h and H)
    - Backup i waits for accepts from 2f other replicas
    - Predicate prepared(m, v, n, i) is deemed true iff backup i has logged:
        - Request m
        - Pre-prepare for m in view v with seq number n
        - Prepare accepts from 2f other backups that match the pre-prepare
    - Guarantee: non-faulty replicas agree on total order for requests within a view
        - Specifically: If prepared(m, v, n, i) is true, prepared(m’, v, n, j) is false for any non-faulty replica j (including i = j)
        - Why? Need 2f+1 to accept prepare for prepared to be true. If two different replicas each achieved 2f+1, their quorums must intersect in f+1 nodes, which must include 1 non-faulty replica.
        - That 1 replica must have accepted two conflicting prepares (or pre-prepares if it is the primary for v), i.e., two prepares with same v and seq number n and different D(m). Not possible if that replica is not faulty.
- Commit phase (collect committed certificate)

    - Replica i multicasts (“commit”, v, n, D(m), i) to other replicas
        - After prepared(m, v, n, i) is true
    - A replica accepts a commit message if
        - Signatures are correct
        - View number v matches replica’s current view number
        - Sequence number n is valid (between global h and H)
    - Predicate committed(m, v, n) is true iff
        - prepared(m, v, n, i) is true for all i in a set of f+1 non-faulty replicas
    - Predicate committed_local(m, v, n, i) is true iff
        - prepared(m, v, n, i) is true and i has accepted 2f+1 commits*
            - Commits that match pre-prepare for m (i.e. v, n, D(m) are same)
    - Invariant: if commited_local(m, v, n, i) is true for some non-faulty i, then committed(m, v, n) is true
        - Ensures that non-faulty replicas agree on sequence numbers of request that commit locally, even if they commit in different views at each replica
        - Ensures that any request that commits locally at a non-faulty replica will commit at f+1 or more non-faulty replicas eventually
- Execution
    - Replicas execute operation once request is committed
        - Waits until committed_local(m,v,n,i) is true
        - Waits until i's state machine reflects sequential execution of all requests with lower seq numbers (safety property)
    - Replicas send reply directly back to client
    - Replicas discard requests with lower timestamps than that of last reply sent to client to guarantee exactly-once semantics

Byzantine primary

- Assume primary sends two distinct requests m and m' for same seq number n
- Claim: backups won't prepare if primary lies
    - Argument: two quorums that accept a prepare will intersect in one non-faulty node, which won't have accepted both m and m'

View change

- If replica suspects primary is faulty, it requests a view change
    - Sends view change request to all replicas
    - Replicas ACK view change request
- New primary collects quorum (2f+1 nodes) of responses
    - f+1 non-faulty nodes for commit + 2f+1 nodes for view change guarantees committed operations persist into new view (intersection of 1)
    - Sends new-view message with this certificate
- Considerations
    - Need committed operations to survive into next view
    - Need to preserve liveness

        - Don't want rapid view changes if primary is actually okay
        - Don't want malicious replicas trying to propose bogus view changes

Garbage collection

- Log will grow without bound, if all messages and certificates are stored
- Need: protocol to shrink the log when it gets too big
    - Can't discard messages, certificates on commit
        - Need them for view change
    - Replicas have to agree to shrink the log
- Checkpoints with proofs
    - Messages must be kept in log until associated request has been provably executed by f+1 non-faulty replicas
    - Proofs generate periodically, e.g. when seq num % 100 = 0
        - State generated by proof request - checkpoint
        - Checkpoint with proof - stable checkpoint

Proactive recovery

- Existing guarantee: good services, provided no more than f failures over system lifetime
- Want: correct service provided no more than f failures in small time window (e.g. 10 min)
    - Problem: cannot recognize faulty replicas
    - Proactive recovery: recover replica to good state whether faulty or not

Recovery protocol sketch

- Components
    - Watchdog timer
    - Secure co-processor
        - Stores node's private key (of private-public keypair)
    - Read-only memory
- Restart node periodically
    - Saves its state (timed operation)
    - Reboot, reload code from read-only memory
    - Discard all secret keys (prevent impersonation)
    - Establish new secret keys and state

Performance

- Byzantine Fault-Tolerant File System (BFS) runs atop BFT
- Performance relative to NFS - BFS is 15% slower

Practical limitations

- Protection achieved only when at most f nodes fail
- Needs more messages, rounds than conventional state machine replication
- Does not prevent many classes of attack
    - Turn a machine into a botnet node

    - Steal SSNs from servers

Quiz questions

- PBFT requires the use of secret-key cryptographic operations [T]
- Clients conclude an operation has succeeded once the primary replica responds to it [F]
- If at most f nodes can be malicious, once a replica has seen an operation written at f+1 other replicas, it knows that the operation has been safely committed [F]
- Replica set can provide correct state machine replication even when one replica fails [T]
- Replica set can provide correct state machine replication even when two replicas fail, the second failure occurring more than 10 minutes after the first replica has recovered [T]
- The replica set can implement an arbitrary deterministic program correctly even when the current primary fails [T]

Scalability, Consistency, and Transactions

Distributed Hash Tables and Chord

Peer-to-peer advantages

- Better load balancing
- Easier deployment
- No single point of failure
- Harder to attack

BitTorrent

- User clicks on download link
    - Gets torrent file with file chunk hashes
    - Gets IP address of tracker - used to locate peers who have file
- After downloaded, user informs tracker it has file
- User's BT client serves file to others

Centralized lookup (Napster)

- Central database must hold O(n) state (n = number of keys)
- Central database is a single point of failure

Flooded queries (Gnutella)

- No single point of failure
- Lookup requires O(n) messages (n = number of peers)

Routed DHT queries (Chord) - goals

- Robust system

- Reasonable amount of state/node
- Reasonable number of hops

Distributed Hash Table API
- lookup(key) -> IP address (Chord lookup service)
- send-RPC(IP address, put, key, data)
- send-RPC(IP address, get, key) -> data

System design
- Distributed applications performs get(key), put(key, data)s on
- Distributed hash table (DHash) performs lookup(key) on
- Lookup service (Chord)

DHT design challenges
- Decentralized - no central authority
- Scalable - low network traffic overhead
- Efficient - find items quickly (low latency)
- Dynamic - allow nodes to leave/join system

Chord paper link

Chord lookup algorithm properties
- Scalable - O(log N) state held per node, N number of keys
- Efficient - O(log N) messages per lookup, N number of servers
- Robust - can survive massive failures

Chord identifiers
- Key identifier - SHA-1(key)
- Node identifier - SHA-1(IP address)

Consistent hashing
- Circular, 7-bit ID space
- Key with key-id k stored at node with next-higher id

Simple lookup algorithm
- If succ < key-id, call succ.Lookup(key-id)
- Otherwise, return succ

Finger tables
- Traversing successor list takes O(n) time in number of nodes
- Finger table allows log N-time lookups
    - Finger i points to successor of $n + 2^i$
- Lookup algorithm

        - Find highest n in finger table s.t. n < key-id
        - If n found, call n.Lookup(key-id)
        - If not found, return succ
- Performance
    - Reduces lookups to O(log N) hops

Node joining

- Find node successor - lookup(node-id)
- Set successor pointer on new node
- Transfer keys from successor to new node
- Tell predecessor to update view of successor
    - Via notify, stabilize messages
- Update finger tables in background

Node failures

- Incorrect lookups - if successors have failed, key gets routed to wrong node
    - Gets routed to next entry in finger table, which could be too high

Handling failures - successor lists

- Each node stores list of r immediate successors
- In case of failures, find first live successor
- List length
    - List of size r = O(log N) makes probability entire list dead 1/N
- Lookup algorithm
    - Find highest n in finger table AND successor list s.t. n < key-id
    - If n exists
        - Call n.Lookup(key-id)
        - If call fails
            - Remove n from finger table/successor list
            - Return self.Lookup(key-id)
    - Else, return succ

DHash DHTs

- Builds key-value storage system on top of Chord
- Replicates blocks for availability
    - K replicas stored at k successors after block on Chord ring
    - Replicas easy to find if successor fails - use successor list
    - Hashed node IDs ensure independent failure
- Cache blocks for load balancing
- Authenticates block contents
    - Types of DHash blocks
        - Content-hash - key = SHA-1(data)
        - Public key - key = public key, data signed by private key

        - Example: Chord file system
            - Root-block signed with public key
                - Signature stored at end of block
            - All nodes contain hash pointers (H(data)) to successors
                - E.g. directory blocks, inode blocks

Evaluation

- Quick lookup in large systems
    - Average messages per lookup O(log N) in number of nodes
- Low variation in lookup costs
    - Tail-end latencies are reasonable
- Robust despite massive failures
    - Until >40% failures, failed lookups remain low

Why don't all services use P2P

- Performance - peer-to-peer communication has high latency, limited bandwidth
- Reliability - user computers less reliable than managed servers
- Trust - cannot trust peers to behave correctly
    - Secure DHT routing is hard, unsolved problem

DHT takeaways

- Consistent hashing - elegant way to divide workload across machines
- Replication - offers high availability, recovery after failure
- Incremental scalability - as nodes are added, capacity increases
- Self-management - minimal configuration required

Eventual Consistency and Bayou

Availability vs. consistency

- NFS and 2PC have single points of failure
    - Not available under failures
- Distributed consensus algorithms allow view-change to elect primary
    - Strong consistency model
    - Strong reachability requirements

Eventual consistency

- If no new updates to an object, all accesses should return last updated value
- Advantages
    - Fast read/write of local copy
    - Disconnect operation possible

[Bayou paper link]

Bayou - Weakly Connected Replicated Storage System

- Meeting room calendar application
- Calendar entry = (room, time, set of participants)
- Want everyone to (eventually) see same set of entries
    - Users may otherwise 1) double-book rooms, 2) avoid an empty room

Swapping complete database

- Expends lots of network bandwidth
- No conflict handling (i.e. if two concurrent meetings)

Conflict resolution goals

- Automatic - algorithm/protocol should exist for resolving conflicts
- Application-specific - user-specified update functions; identical conflict resolution

Problem - need universal agreement

- Update functions can be applied to DB replicas separately
    - Could result in inconsistent calendar state

Insight - total ordering of updates

- Each node maintains a write log
    - Contains ordered list of updates for node
- Requirement: each node holds same updates, applies in same order
- Requirement: updates are a deterministic function of database contents

Write log

- Timestamp - (local timestamp T, originating node ID)
    - Update a < b if 1) a.T < b.T or 2) a.T = b.T and a.ID < b.ID
- Example
    - (701, A) - A asks for meeting M1 at 10 AM, else 11 AM
    - (770, B) - B asks for meeting M2 at 10 AM, else 11 AM
    - Pre-sync database state
        - A puts M1 at 10 AM
        - B puts M2 at 10 AM
    - Correct eventual outcome
        - Timestamp order execution - M1 at 10 AM, M2 at 11 AM
    - A and B sync
        - A can run B's update function
        - B has to undo its own operation - A's operation has precedence
    - Solution
        - B "rolls back" database and re-runs both operations in correct order

- Takeaway - log holds truth; DB is just an optimization
- Not externally consistent
    - B could have earlier request by wall-clock time
    - Due to clock skew, A's meeting has lower timestamp, is prioritized
- Causality
    - Does not respect causality out of the box
        - (701, A) - A asks for meeting M1 at 10 AM, else 11 AM
        - (700, B) - B asks for deletion of update (701, A)
            - B's local clock is slow, update gets earlier timestamp
        - Delete update is ordered before corresponding add update
    - Solution - use Lamport timestamps
        - If node observes E1 then generates E2, TS(E1) < TS(E2)
            - Note: TS(E1) < TS(E2) does not imply E1 before E2
        - T = max(T_{max} + 1, wall-clock time) used to generate TS
            - T_{max} = highest TS seen from any node, including self
        - Previous example - B sets delete update timestamp to (702, B)
            - Causally-related events get correctly ordered timestamps

Committing writes

- Problem - using timestamps for write ordering arbitrarily constrains order
    - Writes from past could appear
    - Entries in log remain tentative forever - must store log forever
- Strawman - call update (k, p_i) stable if all nodes have seen all updates TS < k
    - Problem - disconnected server could prevent writes from stabilizing
        - Many writes could be rolled back on re-connect
- Criteria for committing writes
    - For log entry X to be committed, all servers must agree:
        - On the total order of all previously committed writes
        - That X is the next in the total order
        - That all uncommitted entries are "after" X
- Procedure for committing writes
    - Primary commit scheme
        - One designated node (primary) commits updates
    - Primary marks each received write with permanent CSN
        - Writes with CSN are committed
        - Complete timestamp = (CSN, local TS, node-id)
    - Nodes exchange CSNs when they sync with each other
        - CSNs define total order for committed writes
- Displaying committed writes
    - Entire log up to newly committed write must be committed
    - Why? Node may not know about earlier committed write
        - Writes propagated between node in CSN order

Tentative writes

- Two nodes may disagree on meaning of tentative (uncommitted) writes
    - Even if the nodes have synced with each other
- CSNs from primary replica used to resolve disagreements
    - Primary replica may re-order updates
    - Final commit order may differ from tentative order

Trimming the log

- On receiving new CSNs, nodes can discard previously committed log entries
    - Why: CSNs are received in order
- Database reflects all writes up to highest CSN

Choosing CSNs (commit order)

- Primary's write order must preserve causal order of writes at each node
    - Uses local timestamps to guarantee this
- Does NOT need to preserve order among different nodes' writes

Syncing with trimmed logs

- Assume node has discarded all committed writes from log
    - Writes maintained in "stable" DB
- Invariant: a node cannot receive a write that conflicts with stable DB
    - New write would have to lower CSN than a discarded CSN
    - But already saw all writes with lower CSNs
        - If write appears again, can discard it
- Syncing with node X
    - If X's highest CSN less than mine
        - X can use my stable DB as starting point
        - X can discard all his CSN log entries
        - X applies tentative writes to new (my) database
    - If X's highest CSN greater than mine
        - X can ignore my DB
- Tentative updates
    - Nodes communicate highest local TS for every other node ("version vector")
        - E.g. A -> B [X:40, Y:20]
        - E.g. B -> A [X:30, Y:20]

Cluster membership changes

- Problem: servers A and B sync, A contains node Z in version vector, B doesn't
    - How to know if Z is new (B is behind) or Z retired (A is behind)
- New server
    - Server Z joins by contacting some server X
    - X issues new server identifier for Z - (T_z, X)
        - T_z is X's logical clock as of when Z joined

    - X issues "new server Z" update <-, T_z, X> to other servers
- Server retirement
    - Z has id (20, X)
    - Z sends update <-,..., Z> "retiring"
    - Servers delete Z from their VV if they see the retirement update
    - A syncs to B
        - A's VV has log entry from Z: (-, 25, (20, X))
        - B's VV has no Z entry
    - Case 1: B is behind
        - B's VV entry for X is [X: 10]
        - Implication: B didn't see X's "new server" Z update
            - Z is new
    - Case 2: B is ahead
        - B's VV entry for X is [X: 30]
        - Implication: B once new about Z, but then saw retirement update
            - Z is retired

Bayou complexity

- Update function log, version vectors, tentative operations
- Cause 1: peer-to-peer sync
    - Want both: disconnected operation, ad-hoc connectivity
- Cause 2: human-focused applications
    - Can't just sync through a central server
    - Need to display both committed, tentative writes

Bayou take-away ideas

- Update functions - allow for automatic, application-driven conflict resolution
- Ordered update log - source of real truth, not DB
- Lamport logical clocks - used to achieve causal consistency

Key-Value Storage and Amazon Dynamo

Horizontal vs. vertical scaling

- Horizontal scaling is chaotic
    - Probability of any failure = $1 - (1-p)^n$
        - p - probability one machine fails in a given period
        - n - number of machines
    - For 50k machines, 99.99966% machine success probability
        - 16% of the time data center experiences failures
    - For 100k machines
        - 30% of the time data center experiences failures

Scaling out - partition and place

- Partition management
    - How to recover from node failure
    - How to deal with changes in system size
        - Nodes joining/leaving
- Data placement
    - On which node to place a partition
- Centralized - cluster manager
- Decentralized - deterministic hashing and algorithms

Data partitioning approaches

- Modulo hashing
    - Given k servers and object id X, place X on server i = hash(X) mod k
    - Problem: if server joins/fails, k changes
        - Have to rehash all objects
- Consistent hashing
    - Assign n tokens to random points on mod 2^k circle
        - Hash key size = k
    - Hash object to random circle position
    - Put object in closest, clockwise bucket
        - successor(key) -> bucket
    - Desired features
        - Balance - no bucket has "too many" objects
        - Smoothness - addition/removal of token minimizes object movements
    - Load balancing problem
        - Each node owns $1/n^{th}$ of ID space in expectation
        - If node fails, successor takes over bucket
            - Smoothness - only localized shift
            - Load balance - successor now owns $2/n^{th}$ of key space
        - Solution: virtual nodes
            - Each physical node maintains v > 1 tokens
                - Each token corresponds to a virtual node
                - Each virtual node has a successor in the ring
            - Each virtual node owns expected $1/(vn)^{th}$ of ID space
            - On physical node failure
                - 1/n of key space distributed among v successor nodes
                - Each physical node picks up at most 1/vn of key space
                    - Assumption v < n, so collisions unlikely
                    - Owns 1/n + 1/vn = (v+1)/vn of key space

[Dynamo paper link]

Dynamo context

- Chord/DHash - intended for wide-area, peer-to-peer systems
- Central challenges - low-latency key lookup with small forwarding state per node
- Techniques
    - Consistent hashing - used to map keys to nodes
    - Replication (at successors) - availability under failures

Dynamo system interface

- Key-value store
    - get(k) and put(k, v)
    - Keys and values opaque to Dynamo
- get(key) -> value, context
    - Returns one value, or multiple conflicting values
    - Context describes version(s) of value(s)
- put(key, context, value) -> "OK"
    - Context indicates which versions this version will supercede/merge

Dynamo's techniques

- Consistent hashing - data placement on nodes
- Sloppy quorums - used to ensure fast reads/writes (availability)
- Vector clocks - used to track versions, resolve versioning conflicts
    - Goal: eventual consistency
    - Prioritize success/low latency of writes over reads
    - Prioritize availability over consistency
- Merkle trees - used to synchronize replicas

Data replication

- Coordinator - immediate successor for a node
- Preference list - list of nodes on which to try replicating a (key, value) pair
    - Starts with N successors determined for a key
    - Size > N to account for node failures
    - Skips tokens to ensure distinct physical nodes
        - Failures affect all objects on a physical node
- Wide-area replication
    - Goal: data should survive failure of entire data center
    - Preference list contains nodes from more than one data center
        - Caveat: don't require replication to remote site before returning

Gossip and lookup

- Gossip: once per second, each node contacts a randomly chosen other node
    - Exchange list of known nodes (including virtual node IDs)
- Each node learns which others handle all key ranges
- Result: all nodes can send directly to any key's coordinator (zero-hop DHT)

    - Reduces variability in response times (i.e. high tail-end latency seen in Chord)

Network partitions

- Traditional consensus approach
    - If three replicas are partitioned into two replicas and one replica (master)
        - Two replica partition - no server can write (no master)
        - One replica partition - no server can write (no majority)
    - Favors consistency over availability when partitions
- Eventual consistency approach
    - Client informed of write completion after some replication
        - Rest of replication continues in background
    - Allows writes in both partitions, but risks
        - Returning stale data
            - Subsequent read goes to outdated replica
        - Write conflicts when partition heals

Sloppy quorums

- First attempt: store put() values on first N live nodes on preference list
- To speed up get()/put()
    - Return success for put() when W < N replicas have completed write
    - Return success for get() when R < N replicas have completed read
- Situation: coordinator doesn't receive W replies when replicating a put()
    - Hinted handoff - coordinator tries next successors in pref list (beyond first N)
        - Indicates intended replica node to backup node
        - Backup periodically tries to forward to intended replica node
- Situation: coordinator doesn't receive R replies when processing a get()
    - Read presumably fails
- Freshness of reads
    - Do sloppy quorums guarantee a get() sees all prior put()s?
    - Example: N = 3, R = W = 2
    - With no node failures, yes
        - Two writers saw each put()
        - Two readers responded to each get()
        - Write and read quorums must overlap (since 2 + 2 > 3)
    - With node failures, no
        - Two nodes in preference list go down
            - put() must be replicated outside of preference list
        - Two nodes in preference list come back up
            - get() occurs before they receive prior put() (via HH)
- Conflicts
    - Example: N = 3, R = W = 2; nodes A, B, C
    - 1st put(k, …) completes on A and B
    - 2nd put(k, …) completes on B and C

    - get(k) completes on A and C
        - A and C return conflicting results
        - Dynamo returns both results
            - User sees union of two shopping carts
            - User sees resurrection of deleted item

Version vectors (vector clocks)

- List of (coordinator node, counter) pairs
- Stored with each key-value pair
- Idea: track ancestor-descendant relationships for data versions stored under key k
    - Rule: if vector clocks compare v1 < v2, first is ancestor of second
- On a put(), Dynamo increments the VV counter for the coordinator node
- On a get(), Dynamo returns the VV for the value(s) returned in the context
    - Users must supply context to put()s that modify same key
- Resolutions
    - Auto-resolving - if v2 > v1, v1 is ancestor and can be dropped
    - App-resolving - if v2 || v1, client performs reconciliation
- Trimming
    - Dynamo stores time of modification with each VV entry
    - When VV > 10 nodes long
        - VV drops timestamp of node that least recently processed the key
    - Deleting entry could raise need for application resolution
        - v2 previously descendant of v1, but now concurrent
- Concurrent writes
    - Two clients see same initial version via get()
    - Both clients send a different put() to same coordinator
    - Want: creation of two versions with conflicting VVs
    - Solution: coordinator detects problem via put() context
    - Actual: not indicated in paper

Durability threats

- Need way to ensure each key-value pair is replicated N times
    - Hinted handoff node could crash before data replicated to node in preference list
- Mechanism: replica synchronization
    - Nodes nearby on ring periodically gossip
        - Compare (k, v) pairs they hold
        - Copy any missing keys held by other
    - Goal: compare and copy efficiently
    - Achieved via Merkle trees
        - One Merkle tree for each virtual node key range
        - Hierarchically summarize key-value pairs held by a node
            - Leaf node - hash of one key's value
            - Internal node - hash of concatenation of children

        - Hashes compared
            - If match, values in subtrees match - prune subtrees
            - If don't match
                - If interior node, compare children
                - If leaf, exchange values
        - Process continued down the tree, starting at root

Dynamo parameters

N	R	W	
3	2	2	Good durability, good R/W latency
3	3	1	Slow reads, fast writes (weak durability)
3	1	3	Fast reads, slow writes (strong durability)
3	3	3	Slow reads, slow writes (but more likely that reads are fresh)
3	1	1	Read/write quorums don't overlap - reads may not reflect writes

Partitioning and placement - evolution

- Strategy 1 - Chord approach
    - New nodes steal key ranges from new successor
    - Requires: expensive copies, recalculation of Merkle trees
- Strategy 2 - Fixed-size partitions, random token placement
- Strategy 3 - Fixed-size partitions, equal tokens per partition

Take-away ideas

- Consistent hashing - broadly useful for replication, even in non-P2P context
- Extreme emphasis on availability, low latency - at the cost of some inconsistency
- Eventual consistency model - let writes/reads return quickly, even with partitions/failures
- Version vectors - allow some conflicts to be resolved automatically

Strong Consistency and CAP Theorem

Spectrum of consistency

- Strong consistency (2PC/Consensus) - Paxos, Raft
- Sequential consistency
- Causal consistency
- Eventual consistency - Dynamo

Components

- Fault-tolerance/durability - operations aren't lost, once committed
- Event ordering - ordering between (visible) operations is consistent across system

Strong consistency (linearizability)

- Happens-before relationship guaranteed
    - Reads should reflect most recent write
    - Subsequent reads should return same value, until next write
- Linearizability
    - All servers execute operations in some identical, sequential order
    - Global ordering preserves each client's own local ordering
        - If event A occurred before event B on same process, A must appear before B in global ordering
    - Global ordering consistent with global time-stamps
        - If ts_op1(x) < ts_op2(y), then OP1(x) precedes OP2(x) in sequence
- Implications
    - Once write completes, all later reads (by wall-clock start time) should return value of that write, or value of later write
    - Once read returns a particular value, subsequent reads should return same value, or value of later write

Sequential consistency

- Drop third condition from linearizability
    - All servers execute operations in some identical, sequential order
    - Global ordering preserves each client's own local ordering
- Operations need not be ordered w.r.t. real-time, but all servers must see the same order

Tradeoffs

- CAP theorem - in the face of network partitions, can have either consistency (linearizability) or availability, but not both
- Low-latency vs. consistency
    - 2PC - write on N, read from 1
    - Raft - write on N/2 + 1, read from N/2 + 1
    - General - |W| + |R| > N

Chain replication

- Writes go to head, which is responsible for ordering
    - Writes are propagated sequentially through replicas to tail
- Reads go to tail, which orders them with respect to committed writes
- Chain replication for read-heavy loads (CRAQ)
    - Replicas maintain multiple versions of objects while "dirty" (uncommitted)
        - Once replicated on tail (committed), notification sent up the chain
    - Read to "dirty" replica triggers check with tail
        - Tail returns committed version number of object
    - Allows reads to any replica in chain, not just tail (supports read-heavy loads)

Causal Consistency and COPS

Linearizability Sequential **Causal** Eventual

Causal consistency

- Writes that are potentially causally related must be seen by all machines in same order
- Concurrent writes may be seen in a different order on different machines

Causal vs. sequential consistency

- All writes must be seen in some, identical order by all machines (sequential)
- Concurrent writes can be seen in different orders on different machines (causal)

Causal consistency and state-machine replication

- Lazy replication (as compared to in linearizable systems)
- Trades off consistency for low-latency

COPS - Scalable Causal Consistency for Wide-Area Storage

- COPS paper link
- New consistency model - causal consistency with convergent conflict handling (causal+)
    - Achieves availability and low latency with slightly stronger consistency guarantees than commonly accepted
- COPS - key-value store that delivers causal+ consistency across wide-area
    - Specifically: casual+ consistency between individual items in data store, even if causal dependences spread across many machines in local datacenter
    - Cost: similar overhead to prior log-exchange based systems (e.g. Bayou)
- COPS-GT - introduces get transactions that provide consistent view of multiple keys
    - Requires: no locks, no blocking, at most two rounds of intra-datacenter requests
- Guarantees
    - Causal consistency - causally connected writes always seen in correct order
    - Convergent conflict handling
        - Replicas never permanently diverge
        - Conflicting updates to same key handled identically at all sites
    - Implication: clients only see progressively newer versions of keys
- Previous causal systems
    - State updated via exchange of logs
    - Causal relationships implicitly embedded in log
    - Log becomes bottleneck (limits scalability, doesn't allow cross-server causality)
- Implementation
    - Local COPS cluster is a linearizable (strongly consistent) key-value store
        - Keyspace partitioned into N linearizable partitions, each of which resides on single node/single chain of nodes

        - Composability of linearizability - if partitions are linearizable, system as a whole will be linearizable (linearizability is a guarantee on single objects)
    - Extensions to key-value store
        - Each key-value pair is assigned metadata
            - COPS - version number
            - COPS-GT - dependent keys and their versions
        - Three exported operations: get_by_version, put_after, dep_check
        - (COPS-GT) Old versions of key-value pairs
    - Key-value store features
        - Keys partitioned via consistent hashing
        - Keys replicated across small number of nodes via chain replication
        - Keys associated with one primary node in each cluster
            - Equivalent nodes - set of primary nodes for a key across clusters
    - Writes, overview
        - After a write completes locally, primary node places it in a replication queue, from which it is sent to key's equivalent nodes
        - Remote equivalent nodes wait until value's dependencies are satisfied in local cluster before locally committing the value
    - Writes, details
        - Client calls put(key, val, ctx_id)
            - Library computes complete set of dependencies deps
            - Identifies some tuples as "nearest" dependencies
        - Library calls (bool, vers) ← put_after(key, val, [deps], nearest, vers=0)
            - Ensures that val is only committed to cluster after all dependencies have been written
            - [deps] - argument only needed in COPS-GT (must be stored)
        - Primary node assigns unique version number to each update
            - Lamport timestamp (higher order) + unique node id (lower order)
        - After write commits locally...
        - Primary node asynchronously sends out write to all equivalent nodes
            - put_after(key, val, [deps], nearest, vers)
        - On receipt of put_after, node checks if nearest dependencies are satisfied
            - bool ← dep_check(key, version) sent out to local nodes responsible for dependencies
                - If dependency has been written, responds true immediately
                - If not, blocks until version has been written
            - If all checks succeed, node commits written value
    - Reads
        - Client calls get library function
        - Library issues read to primary node for key
            - (value, version, deps) ← get_by_version(key, version=LATEST)
        - On receiving response, library returns value to client
            - Client library adds (key, value, [deps]) tuple to client context

- Performance
    - Performance of COPS-GT matches that of COPS for low per-client write rates
    - High per-client write rates result in 1000s of dependencies
        - Performance of COPS degrades slightly, but very markedly for COPS-GT

Consistency Recap

Linearizability (strong consistency)

1. All servers execute operations in identical, sequential order
2. Operation ordering is consistent with each client's local ordering
3. Operation ordering is consistent with assigned global timestamps

Sequential consistency

- Conditions (1) and (2) from linearizability, but not (3)

Causal consistency

- Writes that are causally related must be seen by all machines in same order
- Concurrent writes can be seen in different orders on different machines

Eventual consistency

- Writes are eventually applied* in some total order
    - Reads might not see preceding writes in total order
- If no new updates made to a given object, all reads will eventually return last value

Concurrency Control, Locking, and Recovery

Transactions

- Could consist of multiple data accesses or updates (more than 1 object)
- Must commit or abort as a single atomic unit
    - Commit - all updates permanent, visible to other transactions
    - Abort - database restored to pre-transaction state

Properties of transactions

- Atomicity - either all constituent operations complete, or none do
- Consistency - transaction in isolation preserves integrity constraints on data
- Isolation - transaction's behavior not impacted by other concurrent transactions
- Durability - transaction's effects survive failure of volatile and non-volatile storage

Challenges

- Speed requirements - want transactions to be fast, can't fsync() to disk for each result
- Atomicity/durability - want to handle arbitrary crashes, even non-volatile storage (hard disks, SSDs) use write buffers in volatile memory

Techniques
- Write-ahead logging and checkpointing (recovery) - atomicity and durability
- Locking and snapshot isolation (concurrency control) - isolation
- Application logic - consistency

System design
- Buffer manager moves pages (smallest unit of storage that can be atomically persisted) from buffer pool (volatile memory) to disk (non-volatile storage)
- Force / no-force policy
    - Forcing all of a transaction's writes to disk before transaction commits (yes/no)
    - Force (problem): slower disk writes block transaction from committing
    - No-force: need redo operation (replay committed transaction's writes on disks)
- Steal / no-steal policy
    - Allowing uncommitted writes to overwrite committed values on disk (yes/no)
    - No-steal (problem): buffer manager loses write scheduling flexibility
    - Steal: need undo operation (remove effects of uncommitted operation from disk)

Undo/redo implementation
- Log - maintain sequential file that stores info about transactions, system state
    - Resides in separate partition or disk in non-volatile storage
- Log records
    - Entry for each update, commit, abort operation
    - Contains:
        - Log sequence number (LSN) - monotonically increasing
        - New value (after image) - used for redo
        - Old value (before image) - used for undo

Write-ahead logging (WAL)
- Required to support no-force/steal policy
- Force all log records pertaining to an updated page into log before writing to page itself
- Transaction not considered committed until all its log records forced into the log
    - Including: commit record, i.e. force_log_entry(commit)
- Concern
    - Commit log record size > page size (write won't be atomic)
    - Solution: also write checksum of entire log entry

Isolation and concurrency control
- Isolation (before-after atomicity) - transaction appears to happen either completely before or completely after another transaction

- Schedule - ordering of operations performed by set of transactions
    - Serial execution - all operations of one transaction performed first
    - Concurrent execution - operations of transactions interleaved
- Inconsistent retrieval problem
    - Concurrent execution result different from any serial execution result

Scheduling

- Two operations from different transactions conflict if:
    - They read and write to the same data item (RW)
    - They write and write to the same data item (WW)
- Schedules are equivalent if:
    - They contain the same transactions and operations
    - They order all conflicting operations of non-aborting transactions in the same way
- Conflict serializability
    - Schedule is conflict serializable if it is equivalent to some serial schedule
    - Specifically: non-confliction operations can be reordered to get a serial schedule
- Precedence graph
    - Way to test for serializability
    - Each transaction t represented by a node t
    - Edge from s to t exists if some action of s precedes/conflicts with some action of t
    - Property: schedule is conflict serializable iff its precedence graph is acyclic

Ensuring serializability

- Locking
    - Transactions request shared (RR) or exclusive (W) locks
    - Maintained by transaction manager - responsible for granting/denying
- Big global lock approach
    - Acquire the lock when transaction starts, release when transaction ends
    - Problem: serial schedule, but sacrifices performance
- Independent object locks
    - Doesn't preclude non-serializable interleaving of operations!
- Two-phase locking
    - Once a transaction has released a lock, not allowed to obtain any other locks
    - Sufficient but not necessary - precludes some interleaved, serializable schedules
    - Deadlock possible - can be detected by central controller, transactions aborted
    - Phantom problem - if DB has fancier ops than KV-store, non-serializable schedules possible
        - T1: begin_tx; update employee (set salary = 1.1 * salary) where dept = "CS"; commit_tx
        - T2: insert into employee ("carol", "CS")
        - Valid conflict serializable interleaving could yield different results than any serial execution

Serializability vs. linearizability

- Linearizability - guarantee about single operations on single objects
    - Once write completes, all later reads should reflect that write
- Serializability - guarantee about transactions over one or more objects
    - A transaction schedule is serializable if it is equivalent to some serial schedule
    - Doesn't impose real-time constraints
- More info: http://www.bailis.org/blog/linearizability-versus-serializability/

ARIES - Algorithm for Recovery and Isolation Exploiting Semantics

- Recovery algorithm used in commercial databases such as IBM DB2 and SQL Server
- Key ideas
    - Refinement of WAL (with steal/no-force policy)
    - Repeating history after restart due to a crash (redo)
    - Logging every change, including undo operations *during crash recovery*
        - Helps for repeated crashes/restarts
- Non-volatile storage data structures
    - Transaction log, composed of log records with:
        - Log sequence number (LSN)
        - prevLSN - pointer to previous log record for same transaction
    - Page metadata
        - pageLSN - uniquely identifies log record for last update applied to a page
- In-memory data structures
    - Transaction table (T-table) - one entry per transaction
        - Transaction identifier
        - Transaction status - running, committed, or aborted
        - lastLSN - LSN of most recent log record written by transaction
    - Dirty page table - one entry per page
        - Page identifier
        - recoveryLSN - LSN of log record for earliest change to page not on disk
- Transaction commit
    - When commit begins, write commit log record to log
    - Write all log records associated with transaction to log
    - Write end log record to log (actual commit point)
- Checkpoint transaction
    - Separate transaction that occurs while other transactions are running
        - Does not flush dirty pages to disk
        - Does tell how much to fix on crash
    - Write "begin checkpoint" to log
    - Write 1) current transaction table and 2) dirty page table (volatile DS) to log
    - Write "end checkpoint" to log
    - Force log to non-volatile storage
    - Store "begin checkpoint" LSN -> master record
- Crash recovery

- Phase 1 (analysis)
  - Initialize transaction table and dirty page table to checkpointed values
  - Read log forward from checkpoint, updating tables
    - Remove transactions from T-table if end entry encountered
    - For other log entries, add to or update T-table as necessary
- Phase 2 (redo)
  - Scan log entries forward in time, starting at firstLSN
    - firstLSN - earliest recoveryLSN in checkpointed page table
    - Reapply actions, updating pageLSN
  - At end, DB state should match state recorded by log at time of crash
- Phase 3 (undo)
  - Scan log entries backward from end (two passes?)
    - For updates
      - Write compensation log record (CLR) to log
        - Set UndoNextLSN <- prevLSN for update
      - Undo the update's operation
    - For CLRs
      - If UndoNextLSN = null, write end record
        - Undo for transaction is done
      - Else skip to UndoNextLSN for continued processing
        - Turned the undo into a redo (done in Phase 2)

Concurrency Control II (OCC, MVCC)

Optimistic concurrency control

- Process transaction as if it will succeed
  - Check for serializability only at commit time
  - If fails, abort transaction
- Performance
  - Higher performance than locking-based approaches when few conflicts
  - Lower performance than locking-based approaches when many conflicts
- Three phase approach - modify, validate, commit
  - Modify - reads values of committed data items; updates only local copies
  - Validate - two checks: 1) initial consistency, 2) no conflicting concurrency
  - Commit - if validates, transaction updates applied to DB; otherwise, tx restarted
- Validation phase
  - Consider transaction 1. For all other txns N committed or in validation phase:
    - N completes commit before 1 starts modify (no overlap)
    - N completes commit before 1 starts commit
      - ReadSet 1 and WriteSet N are disjoint (initial consistency)
    - N completes modify (commits could overlap)

            - ReadSet 1 and WriteSet N are disjoint (initial consistency)
            - WriteSet 1 and WriteSet N are disjoint (no conflicting concurrency)
        - To validate transaction 1, check that (A), (B), or (C) hold (in that order)
        - If none hold, validation fails - abort transaction 1
- Comparison to 2PL
    - 2PL + OCC = strict serialization
    - 2PL - pessimistically gets all the locks first
    - OCC - optimistically creates copies, RW validation performed before commit

Snapshot isolation

- All reads in transaction will see a consistent snapshot of database
- Transaction only commits if no updates it proposes conflict with any concurrent updates made since snapshot
- Weaker isolation guarantees than serializability, but often better performance

Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp
- Allocate correct version to reads
- Unlike in 2PL/OCC, reads never rejected
    - Replicas maintain old object versions, so can always return old values
- Transaction split into "read set" and "write set"
    - All reads execute as if one "snapshot"
    - All writes execute as if one later "snapshot"
- Timestamps
    - Transactions assigned timestamps, which are transferred to objects read/written
    - Each object version O_V assigned read and write TS
        - ReadTS - largest timestamp of transaction that reads O_V
        - WriteTS - timestamp of transaction that wrote O_V
- Executing transaction T
    - Reads
        - # Determine last version written to before read snapshot time
          Find O_V corresponding to max WriteTS(O_V) s.t. WriteTS(O_V) <= TS(T)
        - Set ReadTS(O_V) = max(TS(T), ReadTS(O_V))
        - Return O_V to T
    - Writes
        - # Determine last version written to before read snapshot time
          Find O_V corresponding to max WriteTS(O_V) s.t. WriteTS(O_V) <= TS(T)
        - # Abort if another T' exists and has read O after T
          If ReadTS(O_V) > TS(T)
            - Abort and roll-back T
        - Else
            - Create new version O_W
            - Set ReadTS(O_w) = WriteTS(O_W) = TS(T)

Distributed transactions

- Environment: data partitioned over multiple servers
- Problem: 2PL doesn't adapt
- Achieving serializability
    - Centralized transaction manager used
    - Assign global timestamp to transaction
        - Choose some time between lock acquisition and commit
        - Use distributed consensus protocol (Paxos, Raft) to pick timestamp
- Strawman - single Lamport clock, consensus per transaction group
    - Motivating observation: linearizability composes
    - Problem: doesn't solve concurrent, non-overlapping transaction problem

Google Spanner

[Spanner paper link]

Setup

- Dozens of zones (datacenters)
- Each zone contains 100-1000s of servers
- Each server contains 100-1000s of partitions (tablets)
- Every tablet is replicated for fault-tolerance (e.g. 5x)
    - Tablets are replicated via Paxos with leader election
    - Paxos groups can stretch across datacenters

Software stack

- Each server responsible for 100-1000 tablets
    - Tablets hold (key, timestamp) -> value mappings
- Servers implement Paxos state machines on top of each tablet
    - Each state machine stores metadata and log in corresponding tablet
- Operation
    - Reads access state directly from tablet at any up-to-date replica
    - Writes initiate Paxos protocol at leader
- Lock table
    - Maintained at every leader replica to implement concurrency control
    - Contains state for two-phase locking
        - Maps ranges of keys to lock states
    - Designed for long-lived transactions (optimistic concurrency control NA)
    - Only consulted by operations that require synchronization
        - E.g. transactional reads
- Transaction manager

    - Maintained at every leader replica to support distributed transactions
        - Transaction manager replica - participant leader
        - Other replicas in group - participant slave
    - If transaction involves more than one Paxos group
        - Groups' leaders coordinate to perform two-phase commit
        - One participant group chosen as coordinator
            - Participant leader of that group - coordinator leader
            - Participant slaves of that group - coordinator slaves

TrueTime concept

- Global wall-clock time with bounded uncertainty
    - Event e_{now} is invoked at tt = TT.now()
    - Guarantee: tt.earliest <= $t_{abs}(e_{now})$ <= tt.latest
    - Total spread: 2ϵ, where ϵ is the average error bound
- Choosing a timestamp s
    - After commit request arrives at leader, run e_1 = TT.now()
        - Pick s > e_1.latest
        - Ensures s occurs "after" e_1
    - Wait to release locks until TT.now().earliest > s

True-time architecture

- Each data center contains a set of (1+) timemaster machines
    - Majority contain GPS receivers (GPS timemasters)
    - Some contain atomic clocks (atomic-clock timemasters)
- Each machine contains a timeslave daemon
- Client (daemon) polls a number of masters to synchronize itself
    - Error bound ϵ defined as
        - ϵ = reference ϵ (1 ms) + worst-case local-clock drift (200 μs/sec)
    - Daemon poll interval set to 30 sec
        - ϵ = 1 + (.2 * 30) = 1-7ms
        - Average error bound ϵ = 4ms

Transactions supported

- Read-write transactions
    - Uses two-phase locking
    - Includes standalone writes
- Read-only transactions
    - Lock-free, via snapshot isolation
    - Executes at system-chosen timestamp
    - Can execute at any "sufficiently up-to-date" replica
- Snapshot reads
    - Client can specify timestamp or upper bound on timestamp staleness
    - Can execute at any "sufficiently up-to-date" replica

Paxos leader leases

- Timed leases used to make leadership long-lived (default: 10 seconds)
- To obtain lease, leader requests timed lease-votes
    - Replica extends vote implicitly on successful write
    - If quorum of votes received, leader knows it has lease
- Leader request lease-vote extensions if they are near expiration
- Leader's lease interval
    - Start: when leader has discovered it has quorum of lease votes
    - End: when leader no longer has quorum of lease votes
- Lease disjointedness invariant
    - For each Paxos group, all leader lease intervals are disjoint
- Abdication
    - Leaders can abdicate by releasing slaves from lease votes
    - Abdication condition: TT.after(s_{max}) must be true
        - s_{max} - maximum timestamp used by a leader

Read-write transactions - timestamps

- Can be assigned any timestamp between:
    - Time of last lock acquisition
    - Time of first lock release
- Transactions assigned the timestamp of corresponding Paxos commit write
- Timestamp monotonicity invariant
    - Timestamps assigned to Paxos writes in monotonically increasing order
    - Simple with single leader replica
    - Across leaders - invariant dependent on lease disjointedness
        - Leaders assign timestamps within their leader lease interval
- External consistency invariant
    - If start of T_2 occurs after commit of T_1, commit ts(T_2) > commit ts(T_1)
        - $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \rightarrow s_1 < s_2$
    - Enforced via two rules
        - Start - write T_i assigned timestamp s_i > TT.now().latest
        - Commit wait - end commit only when TT.now().earliest > s_i

Serving reads

- Replica can satisfy read at timestamp t if t <= t_{safe}
- Replicas track safe time value t_{safe}
    - Maximum timestamp at which replica is up-to-date
    - Define $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$
        - t_{safe}^{Paxos} - timestamp of highest-applied Paxos write

Read-only transactions - timestamps

- Given global timestamp, can be implemented lock-free

- Approach: snapshot isolation
    - Step 1: choose timestamp s_{read} = TT.now.latest()
        - Using TT.now.latest() preserves external consistency
    - Step 2: execute snapshot read at s_{read} at any sufficiently up-to-date replica

Read-write transactions - implementation

- Client performs and buffers writes locally
- Client issues reads to leader replica of tablet group
    - Leader acquires read locks, returns most recent data
- Client begins two-phase commit after completing all reads, buffering all writes
    - Client chooses coordinator group
    - Client sends commit message to each group leader
        - Includes: identity of coordinator, buffered writes
- On receiving commit message
    - Non-coordinator leaders (prepare)
        - Acquire write locks
        - Choose prepare ts > previously assigned ts (monotonicity)
        - Log prepare record through Paxos
        - Notify coordinator of prepare timestamp
    - Coordinator leader (commit)
        - Acquires write locks
        - Chooses commit timestamp s after hearing from other leaders
            - s >= all prepare timestamps (t_{safe} condition)
            - s > TT.now.latest() at time commit message received (start)
            - s > previously assigned ts (monotonicity)
        - Logs commit record through Paxos
        - Waits until TT.after(s) == true (commit wait)
            - Expected wait at least 2ϵ
        - Sends commit timestamp to replicas, other leaders, client
- All leaders log transaction outcome through Paxos
- All participants apply writes at commit timestamp, release locks

Read-only transactions - implementation

- Scope expression required
    - Set of keys that will be read by entire transaction
- If scope values served by single Paxos group
    - Client issues transaction to group's leader
    - Leader assigns s_{read}, executes read
        - Can set s_{read} = LastTS() - timestamp of last committed write
            - Satisfies external consistency
- If scope values served by multiple Paxos groups
    - Possible solution: run round of negotiation
    - Client executes reads at s_{read} = TT.now().latest

        - Can be executed at any sufficiently up-to-date replica

Boutique Topics

Conflict Resolution (OT), Crypto, and Untrusted Cloud Services (SPORC)

Concurrent writes can conflict

- Encountered in many settings
    - Peer-to-peer (Bayou)
    - Multi-master: single cluster (Dynamo), wide-area (COPS)
- Potential solutions
    - "Last writer wins"
        - Related: ignored outdated writes (Thomas write rule)
    - Application-specific merge/update (Bayou, Dynamo)

Approach: incremental updates

- Instead of specifying new values, specify diffs (updates)
    - E.g. Encode $10 transaction from Alice -> Bob as: Alice -$10 and Bob +$10
    - E.g. Encode word insertion in a shared doc as: insert(string, position)
- Update operations must be commutative
    - Commute: Withdraw $10, Deposit $15
    - Don't commute: Insert("1500s", 166), Delete(1, 0)
        - Position changes based on whether delete comes first

Operational transformations (OT)

- Original state of system is S, ops a and b are to be performed concurrently on S
- Depending on order of application, second op is transformed
    - If a applied first, then final state: S * a * b'
        - b' = OT(b, {a})
    - If b applied first, then final state: S * b * a'
        - a' = OT(a, {b})

Cryptography

- Symmetric (secret key) crypto
    - Main challenge: how to distribute key to participants
    - Objectives
        - Confidentiality (encryption)
        - Message authentication and integrity (MAC)

    - Encryption
        - C = E(M, K)
    - Decryption
        - M = D(C, K)
- Public-key cryptography
    - Encryption API
        - ciphertext = encrypt(message, PK)
        - message = decrypt(ciphertext, sk)
    - Digital signatures API
        - signature = sign(message, sk)
        - isValid = verify(signature, message, PK)
- Cryptographic hash functions
    - Produces short, fixed-size number H(m) from message m of arbitrary length
    - One-way function
        - Efficient - easy to compute H(m)
        - Hiding property - given H(m), hard to find m
        - Random - want output to "look" random
    - Collision resistance
        - Strong resistance
            - Resistant to finding any collision (birthday paradox)
            - E.g. find any (m, m') s.t. H(m) = H(m')
        - Weak resistance
            - Resistant to finding specific collision
            - E.g. given m, find m' s.t. H(m) = H(m')
    - Example: checking validity of typed passwords
        - Don't want to store passwords themselves in a physical file
        - Instead: store salt, h = H(password || salt)
- Hash pointers
    - Hash of a data structure's contents
        - Can be used to both look it up and verify its integrity
- Self-certifying names
    - A file F named s.t. F_{name} = H(data)
    - Application: P2P file sharing software, e.g. Limewire
        - Participants can verify that H(downloaded) = F_{name}
    - Application: distributed file sharing, e.g. BitTorrent
        - Large file is split into smaller chunks (~256KB each)
        - Torrent file specifies name/hash of each chunk
        - Participants verify that H(downloaded) = C_{name}
        - Security assumption: torrent file is from trustworthy source
- Hash chains
    - Goal: tamper-evident log of data
    - If data changes, all subsequent hash pointers change

SPORC

- SPORC paper link
- Goals
  - Practical cloud apps - real-time collaboration (concurrent, low-latency editing of shared work), disconnected/offline operation, dynamic access control
  - Untrusted servers - can't read user data, can't tamper user data without risking detection, clients can recover from tampering
- SPORC server responsibilities
  - Storage
  - Ordering messages
- Problem 1 - keep clients' local copies consistent
  - Operational transformations (OT)
    - Allow clients to execute concurrent operations without locking
    - Enables conflict resolution
    - Supports offline access
- Problem 2 - dealing with a malicious server
  - Fork* consistency
    - Guarantee that if server is well-behaved, clients' committed histories are linearizable (one's committed history is a prefix of the other's)
    - Prevents misbehaving server from equivocating about order of operations, unless it is willing to fork clients into disjoint sets
    - SPORC supports automatic recovery from such forks via OT-based conflict resolution
- Features
  - Clients digitally sign all operations with their private key
  - Problem: misbehaving server could present different clients with divergent views of the history of operations
    - Solution - fork* consistency
    - Clients embed their view of their history into every operation they send
      - Clients maintain hash chain over their view of committed history
      - $h_i = H(h_{i-1} \,||\, H\,(op_i))$
        - h_i is the ith hash chain value (h_0 is some initial constant)
        - op_i is the ith operation in the history
      - When a client submits a new operation op_n, also submits h_n
        - Hash chain value can be verified by another client
    - If clients to whom the server has equivocated ever communicate, they will discover their server's misbehavior
    - Consequently, server must divide clients into disjoint groups, and not allow clients to communicate across group boundaries
  - Use of a central, untrusted server
    - Allows clients to verify whether their operation has been committed in a timely manner (without pinging other clients, which may be offline)
    - Defines global ordering on operations

- System state
    - Local state - compact representation of the client's current view of the document
    - Encrypted history - set of operations stored at/ordered by server
    - Committed history - official set of (plain-text) operations agreed upon by clients
        - Derived from server's encrypted history by applying appropriate OT to account for particular server ordering
        - Guaranteed to be fork* consistent
    - Pending queue - ordered list of client's local operations that have been applied to local state, but not yet committed
        - Guaranteed to be causally consistent
- Steps
    - Client application generates an operation, applies it to its local state, and then places it at the end of its pending queue
    - Client submits its oldest queued operation to the server, if none pending
        - Assigns it: client sequence number (clntSeqNo)
        - Includes in it: global sequence number of last committed operation (prevSeqNo) and corresponding hash chain value (prevHC)
        - Encrypts its payload and digitally signs it
    - Server adds submitted operation op to its encrypted history
        - Assigns it next available global sequence number (seqNo)
        - Forwards operation op to all clients participating in document
    - On receiving encrypted operation op, client
        - Verifies its signature V
        - Verifies that clntSeqNo, seqNo, and prevHC fields have expected values
            - Verifies that $H(h_{i-1} \,||\, H(op_i)) == prevHC$
        - If checks succeed, decrypts payload D for further processing
        - If checks fail, client concludes server is malicious
    - Client adds operation to end of its committed history, after transforming it
        - Transforms op past any operations committed since it was generated (i.e. operations with global seq numbers greater than op's prevSeqNo)
    - Client takes action based on received operation
        - If incoming operation is client's own, client dequeues oldest element in its pending queue (uncmt. version of same op) and prepares to send next op
        - Otherwise, client transforms op past all its pending operations, and transforms all those operations with respect to it
    - Client returns transformed version of incoming operation to application
    - Application applies op to its local state
- Recovering from a fork
    - Algorithm executed once two clients detect that they have been forked
    - Clients abandon malicious server and agree on new one
    - Clients roll back histories to last common point before fork
        - Common history uploaded to new server
    - Each client resubmits operations seen after fork

        - OT used to ensure that resubmitted operations are merged safely
        - If same op appears in both histories, inverse operation op^{-1} constructed
            - Transformed pass all operations following duplicate
            - Applied at end of sequence to cancel effect of duplicate
- Access control
    - Can't trust server to do it
    - Clients encode ACL changes as special ModifyUserOp meta-operations
        - Not encrypted, so non-malicious server can reject operations from users with insufficient privileges
        - Every client maintains own copy of ACL, and refuses to apply operations from unauthorized users (no need to trust server)
    - Payload of document operations encrypted under symmetric key known only to document's current members
        - Document creator generates random AES key, encrypts it under his own public key, and embeds it in DocumentCreate meta-op for future use
        - If user removed, AES key must be changed by an administrator
            - New key submitted as part of ModifyUserOp
    - Operations concurrent to ModifyUserOp removals can be safely ordered before
    - Concurrent ModifyUserOp submissions managed via barrier primitive
        - Invariant: every operation following a barrier operation with global seq number b must have a prevSeqNo >= b
        - Result: all future operations must acknowledge barrier operation
- Summary
    - Concurrent operations introduce conflicts in eventual-/causal-consistent systems
    - Solutions - OTs, conflict-free replicated data types (CRDTs)

Blockchains

Double-spending problem

- Traditional e-cash approach - verify with trusted third party (e.g. bank)
    - Bank maintains linearizable log of transactions
- Decentralized approach - make transaction log
    - Public
    - Append-only
    - Strongly consistent

Blockchain - append-only hash chain

- Creates tamper-evident log of transactions
    - A block cannot be modified because hash pointer to it will not check out
- Security based on collision-resistance of hash function
    - Difficult to find another block m' s.t. if h = hash(m), h = hash(m') as well

Consensus

- Byzantine fault-tolerant protocol required to achieve consensus on replicated log
- Based on membership
    - Assume independent failures
    - Necessitates strong notion of identity
- Sybil attack (2002)
    - Idea - one entity can create many "identities" in system
    - Typical defense - only allow 1 identity per IP address
    - Problem - IP addrs not difficult to acquire/spoof with botnets and cloud services
- Bitcoin's solution - consensus based on work
    - Count amount of CPU power expended toward new contribution to log
        - Use proof-of-work to cryptographically certify CPU work performed
- Chain length requires work
    - Generating new block requires "proof of work"
    - Correct nodes accept longest chain
    - Creation fork requires rate of malicious work >> rate of correct work
    - Implication: the older the block, the more "stable" it is

[Bitcoin whitepaper link]

Bitcoin proof of work

- Find nonce s.t. h(nonce || prev_hash || block data) < target
    - Target has some number of leading 0s
        - Recalculated every 10 minutes
    - Calibrated to desired level of difficulty
        - Goal: one block every 10 minutes
        - More total CPU power in system -> smaller target
- Incentivizing correct behavior
    - Correct behavior is to accept longest chain
        - Length determined by aggregate work, not number of blocks
    - Miners incentivized only to work on longest chain
- Randomized leader election
    - Nonce discovery doubles as leader election for 10-minute epoch
    - Leaders elected randomly
        - Probability of selection proportional to leader's % of global hash power
    - Leader decides which transactions comprise a block

Transaction commitment

- At time T, block header constructed
- Transactions delayed 10-20 min before appearing in block
    - Block header constructed at time T
    - Transactions in header received between [T - 10 min, T]

    - Block generated at time ~T + 10
- Transaction considered committed when 6 blocks deep
    - Takes ~1 hour for transaction to become committed

Transaction format

- To determine Alice's balance, scan entire blockchain for outputs directed to PK_Alice
- Inputs must be "fully spent" in a transaction
    - IOW: a transaction output can only appear in a single later input
    - Caveat: sender can transfer Bitcoins to another address it owns
    - Implication: a transaction's history takes O(1) space in the blockchain

Block construction

- Constant size block header
    - Composed of: prev hash, nonce, root hash (see below)
- Merkle tree of hashes
    - Leaves of tree - transactions
    - Nodes of tree - hash of left and right children
    - Root of tree - included in block header
- Verifying that a given transaction is included in a block
    - Requires obtaining logarithmic number of hashes up to the root
    - Hash values computed and verified at each level of tree, starting at bottom
- Can prune tree to reduce storage needs over time
    - Specifically: when a transaction's outputs are spent, it can be pruned

Scaling issues

- Bitcoin only allows 3-4 transactions to be committed/second
    - Limitation based on block size (1MB, 2000 txns) and commitment speed (10 min)
- Participation requires downloading, verifying full blockchain
- In comparison: Visa handles 2,000 transactions/second
    - Peak load: 47,000 transactions/second

Mining pools

- Mining involves a very low probability of a very large payout
    - High amount of risk involved
- Goal: amortize the risk of mining
    - Computational resources are pooled
    - Rewards are split as a fraction of work
- Verification
    - Want to ensure that entities claiming reward were indeed contributing to effort
    - Easier proofs-of-work ("shares") demonstrated to admins
        - Pool is split based on number of shares contributed
- Theft prevention
    - Want to prevent pool admin from absconding with block reward/mining fees

        - Block header (which includes mining reward transaction) is contributed by pool

Content Delivery Networks

DNS

- Uses
    - Hostname to IP address translation
        - Sometimes: IP address to hostname translation (reverse lookup)
    - Hostname aliasing
        - Use multiple hostnames (aliases) for single host (facebook.com, fb.com)
- Originally: flat namespace
    - Per-host file named /etc/hosts
        - Each line in file = IP address & DNS name
    - Problem - doesn't scale
        - Traffic implosion (lookups, updates)
        - Single point of failure
- Requirements
    - Want: scalability (decentralized maintenance), robustness, good performance
    - Don't need: strong consistency properties
- Hierarchical DNS namespace
    - Root servers
        - 13 root servers worldwide (A Verisign, C Cogent, etc...)
        - Each is a cluster of servers, replicated via IP anycast
    - Top-level domain (TLD) servers - e.g. com., gov., edu.
    - Authoritative DNS servers
        - An organization (e.g. Princeton)'s DNS servers
        - Maintained by organization itself or ISP
    - Local name servers - make queries, cache results for clients
        - Sends out queries to other name servers on behalf of client
        - Also called default or caching name servers
    - Resolvers - software running on clients
- Types of resource records
    - A (address) - name = hostname, value = IP address
    - NS (name server) - name = domain, value = hostname of authoritative for domain
    - CNAME - name = alias, value = canonical name
    - MX (mail exchange) - name = domain, value = name of mail server
- Types of queries
    - Recursive - nameserver responds with answer (e.g. A record) or error
        - Less burden on query initiator
        - More burden on nameserver
            - Most root and TLD servers won't answer

    - Iterative - nameserver can respond with a referral
        - More burden on query initiator
        - Less burden on name server
- Sample iterative query (via dig +norecurse)
    - Contains "Authority Section" with NS (referral) records
    - Contains "Additional Section" with glue records
        - IP addresses of DNS servers of referred identities
- Caching
    - All DNS servers cache responses to queries
    - Responses include time-to-live (TTL) field
        - Server deletes cached entry after TTL expires
    - Plays key role in CDN (e.g. Akamai) load balancing

DNS security

- Implications of subverting DNS
    - Redirect victim's web traffic to rogue servers (via wrong A/NS records)
    - Redirect victim's email to rogue email servers (via wrong MX record)
- Security issue 1
    - Local DNS server (run by coffee shop, contractor) returns incorrect response
    - Current: must trust, use HTTPS
    - Future solution: DNSSEC extension to DNS (signed DNS)
- Security issue 2 - cache poisoning
    - Add query request/response for foobar.com to DNS cache
        - Get user to lookup foobar.com via phishing attack, etc.
    - Replace correct IP address for google.com with IP address for foobar.com
    - Solution: Bailiwick checking - ignore extra records (e.g. in Additional Section) not in/under same DNS zone as question
    - Issue: Kaminsky poisoning - attack floods resolver with answers in bailiwick

HTTP

- Problem: stop-and-wait for objects
    - New TCP connection must be opened/closed for each object in request
    - Incurs TCP round-trip-time (RTT) delay for each request
- HTTP keepalive - let TCP connection persists across requests
    - Don't have to repeat handshake protocol/slow-start period
    - Requests/response for objects still sequential
        - HTTP response fully received before next HTTP GET dispatched
- Pipelining (HTTP 1.1)
    - Send GET requests in parallel (don't wait for responses)
    - Benefits
        - RTT amortized across multiple object requests
        - Reduces overhead of HTTP requests
            - Multiple requests are combined in a single TCP packet

Hosting

- Scenario: overloaded popular web site
- Solution: replicate site across multiple machines
- Question: how to direct clients to particular replicas
    - Solution 1: manual selection by clients
        - Each replica has its own site name
        - Some web pages list replicas, allow clients to choose link
    - Solution 2: load-balancer (single IP address; multiple machines)
        - Run multiple servers behind single IP address
        - Fix mapping for a particular TCP connection
        - Con: no geographical diversity
        - Con: need special handling for TCP connections
        - Con: does not reduce network traffic
    - Solution 3: DNS redirection (multiple IP addresses for single DNS name)
        - Single DNS name, but different IP address for each replica
        - Pro: no TCP connection issues
        - Con: round-robin selection may be less responsive
        - Con: does not reduce network traffic

Caching

- Overview
    - Instead of transferring same content over long distances, place closer to clients
- Benefits
    - Users get better response time
    - Network gets reduced load
- Reverse proxies
    - Goal: reduce server load
    - Cache data close to origin server
    - Clients communicate with proxy, think it is origin server
    - Does not work for dynamic content
- Forward proxies
    - Goal: reduce network traffic, latency
    - Cache data close to clients
    - Clients configured to send HTTP requests to proxy first
- Outstanding problems
    - Flash crowds overwhelm web servers, access link, backend DB
    - Rise of rich content - audio, video, photos
    - Diversity causes low cache hit rates (25-40%)

CDNs

- Core idea: proactive content replication
    - Content provider (e.g. CNN) pushes updates from origin server

    - CDN replicates content on servers throughout Internet
- Replication selection goals
    - Live server (availability)
    - Lowest load (to load balance across servers)
    - Closest (to reduce latency)
    - Best performance (throughput, latency, reliability)
- How Akamai uses DNS
    - User makes GET request to content provider (e.g. CNN.com)
    - Content provider returns base resource (index.html), links to auxiliary resources (e.g. images) located at cache URL (e.g. cache.cnn.com/foo.jpg)
    - User looks up cache URL
        - DNS TLD server makes referral to Akamai global DNS server
        - Akamai global DNS server makes referral to Akamai regional DNS server
        - Akamai regional DNS server makes referral to local Akamai cluster
    - User makes GET request for resource to local Akamai cluster
        - Local cluster retrieves resource from CNN (if not already cached)
        - Local cluster caches result and returns it to the user
- Mapping system
    - Equivalence classes of IP addresses
        - IP addresses grouped by experienced performance, inter-connectivity
    - IP classes mapped to preferred server clusters (high-level)
        - Mapping made based on performance, cluster health, etc.
        - Mapping updated roughly every minute
            - Akamai regional DNS uses short, 60-sec DNS TTLs
        - Adapting to cluster/network failure
            - Ping/traceroute used to detect, reroute
    - Client requests mapped to specific servers in cluster (low-level)
        - Akamai load balancer selects, e.g. to maximize cache hit rate
        - Adapting to server failure
            - Load balancer reroutes

Distributed Mesh Wireless Networks

[Roofnet paper link]

Roofnet design choices

- Volunteers host nodes at home
    - Pro: open participation, without central planning
    - Con: decentralized topology
- Omnidirectional antennas (vs. directional antennas)
    - Pro: ease of installation - don't need to choose neighbors

    - Con: links interfere, likely low quality
- Multi-hop routing (vs. single-hop hot spots)
    - Pro: improved coverage
    - Challenge: must build routing protocol
- Goal: high TCP throughput

Addressing

- IP address selection
    - High byte - private prefix
    - Low 3 bytes - low 24 bits of Ethernet address
- NAT between Roofnet and wired Ethernet
    - Private addresses (192.169.1/24) for wired hosts
        - Hosts can't connect to each other; only to Internet
    - Result: no address allocation coordination across boxes required

Internet gateways

- Gateways can reach Internet hosts
    - Job: translate between Roofnet and Internet IP address spaces
    - Gateway test: node sends DHCP request on Ethernet
- Roofnet nodes track gateway used for each open TCP connection
    - Best gateway could change; open connections will continue using existing GW
    - Gateways could fail, causing TCP connections through that GW to fail

Wired v. wireless links

- Wired links - fixed bit error rate
- Wireless links - bit error rate depends on interference, distance, etc.

Link loss rate/throughput observations

- Higher loss on low hop-count routes
    - E.g. 90% 1 hop route (A -> C) vs. 10% each on A -> B, B -> C
- Minimum-hop-count routes throughput-suboptimal

Link loss is high and asymmetric

Routing protocol (Srcr) - route computation

- Goal: find highest-throughput route between any pair of Roofnet nodes
- Each link has an associated **metric**
- Data packets contain **source routes**
- Nodes keep **database** of link metrics between pairs of nodes
    - Forwarded packets - metric of link written to packet's source route
    - Overheard queries - overheard link metrics added to database
    - New packet - if route not found in database, node floods route query
        - Route queries contain route from requesting node

        - Link metrics learned from responses added to database
- Dijkstra's algorithm used on database to compute routes

Routing protocol (Srcr) - querying

- Ordinary operation - nodes only talk to gateway
    - Gateways flood dummy queries for non-existent destination addresses
        - Result 1: all other nodes learn route to gateway
        - Result 2: when node sends data to gateway, gateway learns route back
- Problem: flooded queries don't necessarily follow best route
    - Solution: update and re-broadcast query with computed best route
        - Step 1: add link metric info in query's source route to database
        - Step 2: compute best path from query's source
        - Step 3: replace query's path from source with computed best route
        - Step 4: rebroadcast modified query
    - Corollary: queries forwarded as long as best route with lower metric found

ETX: Expected Transmission Count

- Link ETX - predicted number of transmissions
    - Calculated using forward and reverse delivery rates
    - To avoid retry, data packet and ACK must succeed
    - Link ETX = 1 / P(delivery) = 1 / (d_f * d_r)
        - d_f = forward link delivery ratio (data packet)
        - d_r = reverse link delivery ratio (ack packet)
- Path ETX - sum of link ETX values on a path

Link delivery ratio

- Nodes send broadcast probe packets
    - Sending period of probes is fixed and known to all nodes
    - Loss rates computed based on probe arrivals per measurement interval
- Loss measurements enclosed in transmitted probes
    - B tells A link delivery ratio from A to B

ETT: Expected Transmission Time

- Metric used by Srcr to choose routes
    - Route with lowest ETT chosen (seen as proxy for high throughput)
- Interpretation: total amount of time expected to send data packet along a route
    - Takes into account highest-throughput transmit bit-rate, delivery probability
- Mechanism
    - Each Roofnet node sends:
        - 1500-byte broadcasts at every available bit rate b
        - 60-byte (min size) broadcasts at 1MB/s
    - Delivery probability at a bitrate is the product of:
        - (1) Fraction of 1500-byte broadcasts delivered

            - Forward link delivery rate $d_f(b)$
        - (2) Fraction of 60-byte 1Mb/s broadcasts delivered in reverse direction
            - Reverse link delivery rate d_r
- Computation
    - For each bit rate b, ETX_b = 1 / P(delivery) = 1 / $d_f(b)$ * d_r
    - For each packet of length S, ETT_b = (S/b) * ETX_b
        - S/b - time required to send data S at speed b, if no retries
    - Link ETT = $min_b(ETT_b)$

ETT and throughput

- Path throughput estimate t given by t = 1 / ($sum_{hop\ i\ ->\ path}$ $1/t_i$)
- Does ETT accurately predict throughput?
    - Reasonably accurate for short routes, where at most hop can send at a time
    - Underestimates throughput for long (4-hop) paths
        - Distant nodes can send concurrently without interfering
    - Overestimates throughput when transmissions on different hops collide, are lost

Bit rate selection

- 802.11b bit-rates quantized at powers of two {1, 2, 5.5, 11}
- Bad policy (Prism radio firmware): avoid bit-rates with significant loss-rates
- Observation: throughput at bit-rate 2R with up to 50% loss > throughput bit-rate R
- SampleRate
    - Chooses bit-rate based on measured delivery probabilities
    - Approach
        - Sends most data packets at currently believed highest bit-rate
        - Occasionally sends data packets at other bit-rates to update P_b(delivery)
        - Switches to different bit-rate if better throughput estimate
    - Comparison to ETT
        - Similar - delivery probabilities used to judge route throughput
        - Improvement - uses actual data transmissions, as opposed to periodic broadcast probes (allows quicker, more accurate adjustments)
            - Only one hop of information required
            - Delivery ratio estimates not periodic, but per packet

Evaluation

- Test conducted with background traffic (actual users on network)
- Datasets
    - Multi-hop TCP dataset
        - 15-second, 1-way bulk transfers between each pair of nodes
    - Single-hop TCP dataset
        - TCP throughput measured on direct links between each pair of nodes
    - Loss matrix dataset

        - Loss rate measured between each pair of nodes via 1500-byte broadcasts at every 802.1 bitrate
    - Multi-hop density dataset
        - Multi-hop TCP throughput measured between fixed set of 4 nodes
- End-to-end throughput
    - (Results for each pair of nodes in network)
    - Wide spread - between 500-4000 Kb/s
        - 80% between 0-1000 Kb/s
    - Higher hop count correlates with lower throughput
        - Neighboring nodes interfere with each other
    - Comparison with theoretical, loss-free throughput
        - One-hop routes comparable at 5.5Mbps
        - Longer routes far slower than 5.5Mbps
            - Interference between successive forwarding hops
- Gateway throughput (Internet use)
    - Communicating with gateway is typical use case for Roofnet users
    - Throughput higher than E2E for each hop count
        - Roofnet gateways more centrally located than average Roofnet node
    - Even at 4 hops, TCP throughput comparable to DSL
    - Interactive latency okay for a few hops; bothersome over 9 hops
- Other observations
    - Single-hop TCP workload - srcr favors short, fast links
    - Multi-hop TCP workload - srcr somewhat favors short, fast links
    - Lossy links can be useful - >25%-loss links used half the time
    - Good use of mesh architecture - nodes route through a diverse set of neighbors
- Access point network
    - Ensure all nodes are one hop away from Internet gateway
    - Gateway placement
        - Optimal - at each step, add GW that maximizes newly connected nodes
        - Random - choose median set (by number of connected nodes) of gateways, among 250 randomly-selected of a particular size
    - Complete coverage requirement
        - Optimal - 5 for single-hop; 1 for multi-hop
        - Random - 25 for single-hop; 8 for multi-hop
    - Results
        - Optimal, random - multi-hop GWs provide better connectivity, throughput
        - For <=5 GWs, random multi-hop GWs outperform optimal single-hop
        - For larger numbers of GWs, optimally chosen single-hop GWs better

Big Data

Graph Processing

MapReduce

- Suitable for data-parallel tasks
    - Examples: feature extraction, algorithm tuning, big data processing
    - Data can be divided up and assigned to workers; minimal communication needed
- Doesn't efficiently express data dependencies
    - Users must code data transformations
    - Costly data replication is involved
- Iterative MapReduce
    - Only subset of data needs computation in each stage
    - Must pay startup and disk penalty with each stage

Graph-parallel algorithms

- Example: label propagation algorithm
    - Recurrence algorithm - Likes[i] = $\text{Sum}_{j \in \text{Friends}[i]}\{W_{ij} * \text{Likes}[j]\}$
    - Iterate until convergence
    - Parallelism potential - can compute Likes[i] in parallel
- Properties
    - Dependency graph
    - Factored computation
    - Iterative computation - compute B[i] based on A[i], compute A[i+1] based on B[i]

GraphLab paper link
Distributed GraphLab paper link

GraphLab Framework

- Three key components
    - Graph (data representation)
    - Update functions (user computation)
    - Consistency model
- Data graph
    - Graph - social network
    - Vertex data - user profile, current interest estimates
    - Edge data - relationship (friend, classmate, relative)
- Distributed graphs
    - Graph is partitioned across multiple machines
        - Cut using HPC graph partitioning tools (PartMetis, Scotch,...)

    - Ghost vertices maintain adjacent structure, replicate remote data
- Update functions (user computation)
    - User-defined program applied to vertex that transforms data adjacent to vertex
    - Applied asynchronously in parallel until convergence is achieved
- Distributed scheduling
    - Each machine maintains schedule (queue of tasks) over vertices it owns
    - Uses distributed consensus to identify completion
- Consensus model
    - Goal: ensure overlapping computations do not involve data races
        - Example - PageRank computation
            - Vertex pagerank is modified for each incoming link
            - Fix: use temporary variable, update page rank at end
    - Implications of weak/no concurrency control (e.g. locking)
        - Higher throughput (updates/sec)
        - But… potentially slower convergence of algorithm
        - Net impact: not necessarily better performance!
    - Tunable consistency
        - User can tune consistency level to tradeoff parallelism with consistency
    - Serializability
        - For every parallel execution, there exists a sequential execution of update functions that produces the same result
- Distributed consistency
    - Edge consistency
        - Update functions one vertex apart run in parallel
    - Solution 1 - Chromatic engine
        - Uses graph coloring to achieve edge consistency
            - Adjacent vertices assigned different colors
        - Algorithm
            - Stage k: tasks executed on all vertices of color k
            - Barrier synchronization between stages
        - Issues
            - Requires graph coloring to be available
            - Barriers are inefficient when only some vertices are active
    - Solution 2 - Distributed locking
        - Acquire write-lock on active vertex, read-locks on adjacent
        - Problem - acquiring lock from neighboring machine incurs latency penalty
        - Solution - pipeline lock acquisitions (GraphLab)
            - Acquire locks for different update functions together
- Fault-tolerance - handling machine failures
    - Strawman scheme - synchronous snapshot checkpointing
        - Stop the world, write each machine's state to disk
        - Problem: snapshot slows system performance
    - Chandy-Lamport checkpointing

        - Implemented in GraphLab via an update function
        - Asynchronous snapshotting offers performance improvement

Chandy-Lamport Snapshotting

Use of snapshots

- Checkpointing - for crash recovery
- Collecting garbage - remove objects that don't have references
- Detecting deadlocks - can examine current application state
- Other debugging

Global state

- Process state for every process in system
- Channel state for every pair of processes (one in each direction)
    - Includes: sent, in-flight, and received messages

System model and goals

- Problem - record global snapshot (state for each process and channel)
- Components
    - N processes in the system
    - Two FIFO unidirectional channels between every process pair
- Operation
    - No failures
    - All messages arrive intact, not duplicated
- Requirements
    - Snapshotting shouldn't interfere with normal application behavior
    - Any process should be able to initiate snapshot
    - Collect state in a distributed manner

Algorithm

- Initiating a snapshot
    - Process P_i initiates the snapshot by recording its own state
    - P_i sends marker message to all other processes (on N-1 outbound channels)
    - P_i starts recording all incoming messages (on N-1 inbound channels)
- Propagating a snapshot
    - Consider message on channel C_{kj} for process P_j (including initiator)
    - If marker message seen for first time
        - P_j records its own state and marks C_{kj} as empty
        - Marker message is relayed to other process (on N-1 outbound channels)
        - Starts recording all incoming messages on channels C_{lj}, $l \neq k$
    - Otherwise

        - End recording messages on inbound channel C_{kj}
        - C_{kj} state consists of all messages seen since recording began
- Terminating a snapshot
    - All processes have sent markers and recorded their state
    - All processes have received a marker on all their N-1 incoming channels
- Central server gathers partial state to build global snapshot

Implications

- Terminology
    - An event is presnapshot (postsnapshot) if it happens before (after) local snapshot
- Causal consistency
    - If event A happens causally before B, and B is presnapshot, A is too
        - Trivially true if A and B occur on same process
        - If A is a send event on p and B is the corresponding receive event on q...
            - If B is presnapshot, then q hasn't received any marker messages
            - This in turn implies that p hasn't sent any before A
            - This implies that A is also presnapshot
    - Analogously, if A is postsnapshot, B must be too
- Message inclusion
    - For message m sent from process p to q to be in snapshot, snapshot must occur:
        - After q has received its first marker (start C_{pq} recording condition)
        - Before q receives a marker from p (end C_{pq} recording condition)
    - Condition: p (sending) must be presnapshot, q (receiving) must be postsnapshot

Stream Processing

Simple stream processing

- Data read from socket, processed, transformed (via function), and outputted
- Stateless conversion
    - Example: convert Celsius temperature to Fahrenheit
        - **emit** (input * 9 / 5) + 32
- Stateless filtering
    - Example: filter input below some threshold value
        - if (input > threshold) { **emit** input }
- Stateful conversion
    - Example: compute EWMA (exponentially-weighted moving average) temperature
        - new_temp = a * (CtoF(input)) + (1 - a) * last_temp
        - last_temp = new_temp
        - **emit** new_temp
- Stateful aggregation
    - Example: average value per window

- Composing functions
    - Chained operations - CtoF -> Avg -> Filter
    - Directed graph
        - {sensor 1 -> CtoF, sensor 2 -> KtoF} -> Avg -> {storage, Filter -> alerts}

Challenges

- Large amounts of data to process in real-time
- Vertical scaling - micro-batch processing
    - Buffer reads/writes over multiple input elements
    - Pro - higher throughput
        - Read/writes are kernel system calls; buffering reduces context switch time
    - Con - higher latency
        - Individual input is transformed/outputted only after batch is processed
- Horizontal scaling - parallelization
    - Stateless operations (e.g. C to F conversion)
        - Trivial - just add more machines
    - Stateful operations (e.g. averages)
        - Need to join results across parallel computations
        - Can compute partial aggregations for each stream (e.g. sum, count)
        - Problem: failures
            - Need to ensure exactly-once semantics
            - Need to pause/synchronize on aggregation step (e.g. average)
        - Parallelizing joins
            - Example: find trending keywords in tweets
            - Process streams independently, computing (key, sum) tuples
            - Hash partition tweets by keyword
                - Map **m** input streams to **n** output streams
                - Sort (key, sum) tuples, find top-k keywords in output streams independently

Apache Storm

- Components
    - Data - streams of tuples, e.g. Tweet = (author, message, time)
    - Data sources - "spouts"
    - Operators to process data - "bolts"
        - Multiple processes (tasks) run on a bolt
    - Topology - directed graph of spouts and bolts
- Parallelization
    - Incoming streams are split among tasks on bolts
        - Shuffle grouping - round-robin distribution of tuples to tasks
        - Fields grouping - tuples partitioned by key/field
        - All grouping - all tasks receive all tuples (e.g. for joins)
- Fault-tolerance

    - Goal: ensure each input is fully processed
    - Approach: track tree edges in DAG
        - Spouts assign unique IDs to each tuple
        - Record edges that get created as tuple is processed
            - Bolts "emit" dependency tuples
        - Wait for all edges to be marked done
            - Informs source (spout) when complete (via ACK)
                - Spout garbage collects tuple
            - Otherwise, bolt reports failure (ACK)
                - Spout retransmits tuple
    - Challenge: "at least once" semantics
        - Bolts can receive a tuple more than once
        - Tuple replay can be out-of-order
    - Best-effort delivery
        - Dependencies not generated on downstream tuples

Apache Spark Streaming

- Stream split into small, atomic batch jobs (each X seconds in length)
    - Window-based operation
- Individual batches passed to Spark "batch" framework for processing
    - Similar to in-memory MapReduce
- Spark emits micro-batch results
    - Encapsulated in RDDs ("Resilient Distributed Datasets")

Fault-tolerance approaches

- Micro-batches (Spark Streaming, Storm Trident)
    - Each micro-batch may succeed or fail
        - Can recompute as needed; more difficult if stateful
    - DAG specifies pipeline of transformations from micro-batch to micro-batch
        - Can use lineage info to do selective recomputation
    - Failure recovery
        - RDDs occasionally checkpointed via replication to other nodes
        - Approach
            - Get last checkpoint
            - Determine upstream dependencies
            - Recompute using upstream dependencies, starting at checkpoint
- Transactional updates (Google Cloud Dataflow)
    - Computation represented as long-running DAG of continuous operators
    - For each intermediate record at operator
        - Create commit record = (input record, state update, downstream records)
        - Write commit record to transactional log
    - Recovery
        - Restore consistent state of computation

        - Replay lost records
    - Requires: high-throughput writes to distributed store
- Distributed snapshots (Apache Flink)
    - System-wide snapshots taken and saved to durable storage
        - Markers (barriers) added to input data stream to indicate to downstream operators where to snapshot
        - Algorithm based on Chandy-Lamport, but also captures stream topology
    - Recovery
        - Recover latest snapshot from durable storage
        - Rewind stream source to snapshot point, replay inputs

Optimizing stream processing

- Optimal DAG construction
- Scheduling - choice of parallelization, use of resources
- Where to place computation

Cluster Scheduling

Source of resource demand

- Services
    - External demand - must scale supply to match
- Internal jobs
    - Can tradeoff scale (resources requested) with completion time
        - Use 1 server for 10 hours vs. 10 servers for 1 hour
    - Source of demand elasticity
        - Reason why Amazon spot-pricing option exists

Scheduling

- Applications
    - CPU allocation
    - Bandwidth allocation
- Desired properties
    - Isolation
        - Guarantee that misbehaved processes will not affect others
    - Efficient resource usage
        - Resource not idle when there is a process with not fully satisfied demand
        - Work conservation - not achieved by hard allocation
            - A work conserving scheduler always tries to keep scheduled resource busy, if there are submitted jobs
    - Flexibility
        - Ability to express priorities (strict or time-based)

- Fair-sharing
    - Basic - if n users want to share a resource (e.g. CPU), give each 1/n
    - Max-min fairness - if a user wants less than its fair share, can allocate to others
    - Weighted max-min fairness - users assigned weights according to importance

Max-min fairness

- Priorities - can simulate priorities via weights
- Reservations - to ensure process u1 gets 10%, assign u1 weight 10 and all other process weights s.t. sum(weights) <= 100
- Deadline-based scheduling - given job demand and deadline, compute weight
- Isolation - users can not affect others beyond their assigned share
- Implementation
    - Fair queueing
        - Each flow receives $\min(r_i, f)$ where
            - r_i - arrival rate of flow i
            - f - link fair rate
                - f is computed such that $\text{sum}(\min(r_i, f)) = \text{link capacity } C$
    - Weighted fair queuing (WFQ)
        - Each flow receives $\min(r_i, f * w_i)$, where
            - w_i - weight assigned to flow i
            - f - link fair rate
                - f is computed such that $\text{sum}(\min(r_i, f * w_i)) = \text{link capacity } C$
- Properties
    - Share guarantee
        - Each user gets at least 1/n of resource
        - But will get less if demand < 1/n
    - Strategy-proof
        - Users are not better off by asking for more than they need or lying

Multi-resource demand

- Example scenario
    - User 1 wants <1 CPU, 4GB> per task
    - User 2 wants <3 CPU, 1GB> per task
    - How to fairly allocate CPU and RAM?
- Asset fairness - equalize each user's sum of resources shares (strawman)
    - Cluster has 28 CPUs, 56 GB RAM
    - U_1 needs <1 CPU, 2GB RAM> (<3.6% CPU, 3.6% RAM>) per task
    - U_2 needs <1 CPU, 4GB RAM> (<3.6% CPU, 7.2% RAM>) per task
    - Systems runs 12 tasks for U_1 and 8 tasks for U_2
        - U_1 allocated <43% CPU, 43% RAM> (total share 86%)
        - U_2 allocated <28% CPU, 57% RAM> (total share 86%)
    - Problem - violates share guarantee
        - U_1 receives < 50% of both CPU, RAM

- Problem - possible to game the scheduler
    - Goal: achieve share guarantee + strategy proofness for sharing
    - Approach: generalize max-min fairness to multiple resources
- Dominant Resource Fairness (DRF)
    - Dominant resource - resource user has biggest share of
    - Dominant share - fraction of dominant resource allocated
    - Apply max-min fairness to dominant shares
        - Goal: maximize dominant share of users s.t.
            - Requested resource ratios are satisfied
            - (With total allocation for no resource exceeding 100%)
        - Example: cluster has: 9 CPUs, 18 GB RAM
            - U_1 requests <1 CPU, 4 GB> - d.r. is RAM
            - U_2 requests <3 CPU, 1 GB> - d.r. is CPU
    - Online DRF scheduler
        - Whenever available resources and tasks to run, schedule task to user with smallest dominant share

System architecture for big-data scheduling

- Heterogeneity
    - Hadoop | Spark — for data-parallel computations
    - Storm | Spark Streaming | Flink — for streaming
    - GraphLab — for graph processing
    - MPI — for distributed computation
- One framework per cluster challenges
    - Inefficient resource usage - resource wastage; not work conserving
    - Hard to share data - hard to share data (must copy/access across clusters)
    - Hard to cooperate - difficult to share intermediate results, outputs
- Abstraction hierarchy
    - Framework (e.g. Hadoop, Spark) manages 1+ jobs
    - Job consist of 1+ tasks
    - Task (e.g. map, reduce) involves 1+ processes
- Approach 1: global scheduler
    - Inputs
        - Policies, resource availability, estimates, job requirements, execution plan
    - Output
        - Task schedule
    - Advantages
        - Optimal
    - Disadvantages
        - Complex, harder to scale
        - Requires anticipating future requirements
    - Example: Google's Borg
        - Central Borgmaster dispatches tasks to localized Borglets

        - Goal: find machines for a given job
        - Allocation
            - Minimize number/priority of preempted tasks
            - Pick machines which already have a copy of task's packages
                - E.g. for MR, servers which already have input data
            - Spread over power/failure domains
            - Mix high/low priority tasks
- Approach 2: offers, not schedule
    - Master sends resource offers to frameworks
        - Offers consist of vector of available resources - e.g. (4 CPU, 16 GB)
    - Frameworks
        - Select which offers to accept
            - Accepts if offer matches constraints and preferences
        - Perform task scheduling
        - Unlike global scheduler, require another level of support
    - Example: Mesos
        - Principle: two-level resource offers
        - Master makes resource offers to frameworks
        - Framework consists of two components
            - Scheduler - selects which resource offers to use
            - Executor - runs framework's tasks
    - Two questions
        - How long must a framework wait?
            - Depends on distribution of task duration
            - Depends on how picky framework is (hard vs. soft constraints)
        - How should resources of different types be allocated?
            - Dominant Resource Fairness (DRF)
                - Cluster master makes offers to frameworks
                - Offer assigned to user with smallest dominant share
- Ramp-up time
    - Time job waits to get target allocation
    - Improving ramp-up time
        - Preemption: can preempt tasks to give job chance to run
        - Migration: can move tasks around to increase choice
    - Frameworks implement
        - No migration: expensive to migrate short tasks
        - Preemption with task killing: expensive to checkpoint data-intensive tasks
    - Macro-benchmark
        - Simulate 1000-node cluster
            - Job and task durations based on Facebook traces (Oct. '10)
            - Constraints modeled after Google's
        - Fair sharing allocation policy
        - Schedulers compared

        - Resource offers - no preemption, no migration
        - Global-M - global scheduler with migration
        - Global-MP - global scheduler with migration and preemption
- Results
    - Very little difference in CDF of completion times vs. job durations
- Implication
    - Resource offer strategy (e.g. Mesos) can perform just as well as a global scheduler, but offers other benefits, such as not needed to hardcode partitions for different frameworks