CS 134 – Distributed Systems

Spring 2025 – Professor Konstantinos Kallas

Lecture 1: What is a Distributed System?

Distributed System: Multiple interconnected computers that cooperate to provide some service

Challenges in Distributed Design

- Partial Failures
- Ambiguous Failures (we don't know what the errors are)
    - Server failure, Network failure, or both
- Concurrency
    - Ensure consistency of distributed state

Lecture 2: MapReduce, Clock Synchronization

MapReduce (and Hadoop)

- Goals
    - Scalability: Throughput grows with number of machines
    - Fault Tolerance: Makes progress despite machine failures
    - No latency requirement: MapReduce is used for batch jobs
- Naïve Approach
    - Idea: Partition the data, split across servers. Each server is responsible for own data
    - Issue: We may need to access data split across machines
- MapReduce Algorithm
    - Partition: Split dataset into partitions
    - Map: Outputs Key Value Pairs
    - Coalesce: Organize Key Value Pairs into new partitions
    - Reduce: Aggregation using a reducer function
- Issues
    - Synchronization barrier: Cannot run reduce tasks until all map tasks finish

Paradigm: Fault Tolerance via Master Node

- To achieve fault tolerance, the master node:
    - Assigns map/reduce jobs to workers
    - Keeps track of status of tasks
    - Keeps track of machine failures (by retransmitting)
- Abstractions
    - Control (fork) vs Data Communications
    - Client-Server
        - Client makes requests to servers
        - Server listens for requests, processes and returns responses
        - Issue: Cannot Scale
    - Client-Server with threading

        - Listener thread spawns thread to handle each request it receives
        - Listener thread puts requests in a queue. Thread pool is fixed
    - RPC (Remote Procedure Calls)
        - Server offers API of methods with signatures
        - Clients invoke a method using args, waits for responses
- Issue: If master fails, we lose all state of task execution status

Checkpointing

- Issue: If any partition fails, reduce stage will never start
- Solution: Checkpoint after each phase to fall back
    - Slow
    - Writes to disk → Expensive
        - Orders of magnitude worse than writing to memory
- Apache Spark
    - Operations completed in RAM, changes only persist after commits

Replicated State Machines

- Idea: Replicate master state to tolerate failures
- Issue: Inconsistent replicas
    - Workers can potentially communicate faster than the masters communicate to each other:
    - W2 finishes task 2 before M1 tells M2 that W1 has been assigned task 3
- Solution: Synchronization
    - Idea: Apply updates in the same order across all replicas
    - Approach: Given list of received updates, order them based on time of receipt
        - Issue: Clocks are not in sync across replicas
    - Store the sequence of transitions to construct the state
        - Abstractly defined as a state machine
    - Approaches:
        - Replication over Time:
            - Start new replica when one goes down

        - Assumes execution state from failed machine is available to continue from
    - Replication over Space:
        - Run multiple replicas simultaneously
        - Assume that when something fails, all state is lost

Clock Synchronization:

- Naïve Approaches
    - GPS broadcasting (accurate ~1ms) → Power hungry, doesn't work indoors
    - Correct skew based on reference clock → Clock drift, unbounded network delays
        - Potential workaround using Network Time protocol
- Christian's Algorithm
    - Assume time server is available
    - Client sends request packet using local timestamp
    - Server timestamps its receipt of client's request using its own local time
    - Server responds with packet containing its local time on receive and send
    - Client locally timestamps receipt of server's response of send
    - Round Trip Time = $\partial_{req} + \partial_{resp} = (T_4 - T_1) - (T_3 - T_2)$

Lecture 3: Clock Properties, Lamport Clocks, Vector Clocks

Desired Ordering Properties

- Idea: We just want the order between events to be the same
    - All replicas agree on an order:
        - For all i, j, $C_i(a) < C_i(b)$ iff $C_j(a) < C_j(b)$
    - Total Order: for all a, b, $C(a) < C(b)$ or $C(b) < C(a)$
    - Accuracy: If a happened before b, $C(a) < C(b)$
- $C_i(a) < C_i(b)$ → a is ordered before b

Causality

- If A caused B, then A happened before B
- Synchronous Model: All messages are delivered in less time than N
- Asynchronous: A message can take an arbitrary amount of time to be delivered

Logical Clocks

- Idea: Associate every event with a logical time
    - Across Processes, Transitivity, In same process
- Allows all replicas in an RSM to execute updates in the same order
    - Note: cannot handle failures (using only logical clocks)

Lamport Clocks

- Idea:
    - Associate all events with a clock time, order based on clock time
- Algorithm
    - Each process p_i maintains a local clock C_i
    - All process clocks start at time 0
- Issue: We cannot tell if an event occurs before another
    - 1, 2 can occur before 1,2,3,4,5
- Issue: Concurrency
    - If events A, B are concurrent, we cannot establish a total order
    - Solution: Break ties in the same manner across all replicas

        - i.e. Append the process number to each event
    - Process P_i timestamps event e with $C_i(e) \bullet i$
        - $C(a) \bullet I < C(b) \bullet j$ when:
            - $C(a) < C(b)$ OR $C(a) == C(b)$ AND $i < j$

RSMs with Lamport Clocks

- Idea: Place events into a local queue sorted by increasing C(x)
    - We only execute a process if we receive an ACK from all other processes that is greater (given causality)
- Optimizations
    - Reducing waiting
        - Periodically ping all other nodes to sync clock
        - Tolerate temporary inconsistency
            - Apply update immediately upon receiving it
            - Maintain log of updates to roll back
    - Use causal ordering rather than total ordering
        - Not all events require ordering with everyone else
    - Issue: If any one replica goes does, all other replicas are unable to make progress

Vector Clocks

- Idea: Determine a precise partial order of events in distributed system
- Algorithm
    - Label each event with a vector $[c_1, c_2, \ldots c_i]$
        - C_i is a count of events in process I that causally precede e
    - All vectors start at [0, 0, … 0]
    - Updating a vector:
        - For local event on same process, increment c_i by 1
        - If receiving a message from another process with vector $[d_1, d_2, \ldots d_i]$:
            - Modify local vector to me $c_k = \max\{c_k, d_k\}$
            - Increment local entry c_j in vector by 1
- Comparing Vector clocks

- V(a) == V9B) → $a_k == b_k$ for all k
- V(a) < V(b) → $a_k \leq b_k$ for all k and V(a) ≠ V(b)
- Concurrency check:
  - A || b → $a_i < b_i$ and $a_j > b_j$ for some i, j
- V(a) < V(z) if and only if there is a chain of events linked by → between a and z

Lecture 4: Failures, Chandy-Lamport Algorithm

Logical clocks break on failures

Fault Tolerance

- A system should be consistent and should guarantee that everyone will eventually see updates
    - A system S satisfies its specification Spec given that some classes of faults happen to it

Failures

- Crash failures
    - Component is unreachable and never comes back to life
    - We cannot reassign work back to original thread, another process takes over
- Network failures
    - Messages are dropped or reordered
    - Solved through retransmission and attaching identifiers to deduplicate
- Byzantine failures
    - Modified or corrupted messages or processes
    - Component does not follow protocol and executes arbitrarily

Recovering from failures: Checkpoint Storage

- Algorithm
    - Every N seconds
        - Store checkpoints on persistent storage (disk)
    - If a component crashes:
        - Retrieve the latest checkpoint
        - Restore the component state based on it
        - Continue execution from that point on
- Issues
    - Domino Effect: Uncoordinated checkpointing across processes
        - Solution: Coordinated Checkpointing/Chandy-Lamport

    - Inconsistency if a message is transferred between checkpoint executions

Global Checkpointing (Uncoordinated)

- Idea: Take a coordinated snapshot of a distributed system asynchronously
    - Useful for garbage collection, deadlock detection, and evaluating stable properties
- Token Ring
    - Idea: Rotate lock possession across processes in circle/ring order
    - Issue: Snapshots may result in multiple processes thinking they have the lock after recovery
        - P_1 and P_2 both have receives but no sent
        - Note: In weak guarantees that use one lock, there is no consistency issues

Global Snapshots (Uncoordinated)

- Idea: captured state reflects causality in regards to state of individual processes and every communication channel (things sent that are not received)
- Issue: Some snapshots may make the locks unrecoverable
    - P_2 thinks lock is sent, P_3 has not received

Strawman (Coordinated Screenshots)

- Idea: When a snapshot message is received, all processes stop working and prepares for a snapshot
    - High cost on execution
    - Consistent

FIFO Channels

- Idea: No message sent before another arrives AFTER another (using TCP)

Chandy-Lamport Snapshots

- Idea: We need to handle crash failures
- Implementation
    - N processes in system
        - Assume processes cannot fail during snapshot

        - Any process can initiate a snapshot
    - Two FIFO unidirectional channels between each process pair
    - Assume no message duplication or loss
- Core Ideas
    - After taking a checkpoint, send a broadcasting marker message to other processes to also take checkpoints
        - Marker enables processes to discover the need for taking a snapshot
        - Marker servers as a "barrier" on every network channel that all messages received before the marker are included
- Algorithm
    - Note that in this class, we capture using the SENDS of a message
    - Let process p_k initiate the snapshot
        - p_k records its local state (takes a checkpoint)
        - p_k sends out marker messages on every outgoing network channel
        - p_k starts recording messages on every incoming channel
    - Let p_i receive the marker message from p_k (via channel c_{ki})
        - If marker is first marker p_i has seen
            - p_i records its state
            - p_i marks channel c_{ki} as empty
            - p_i sends a marker out to every outgoing channel c_{ij}
            - p_i starts recording incoming messages on all channels other than c_{ki}
        - If p_i has already seen a marker
            - p_i stops recording on channel c_{ki}
            - p_i sets c_{ki}'s final state as the sequence between the start and end of recording
    - The snapshot is complete when every process has received marker messages from every incoming channel
        - i.e. (Processes • (Processes – 1)) markers are sent
- Rollback
    - Recognize there is a failure in the system
    - All active nodes roll back to latest completed snapshot

    - A new/restarted node recovered from its snapshot
        - Loads everything into memory
    - Continue
- Issues
    - If message themselves are nondeterministic, we might lose message on rollback
        - Ex: coin flip

Nondeterminism:

- Sources: Randomness, user input, network delays, process scheduling, OS interrupts
- Issues
    - Reasoning within distributed systems (state explosion)
    - Messes with replication and recovery
- Solution: Message logging
    - Idea: between checkpoints, log all events that lead to nondeterminism
        - Log enough to ensure that same messages are sent as in original execution
        - Thus, we can now guarantee. That we can roll back and execute deterministically

Lecture 5: Fault Tolerance, Primary Backup Replication, Request Types

Fault Tolerance Trade Offs

- Cost on normal execution (latency/CPU/memory)
- Recovery freshness: how far back of a snapshot do we recover from
    - High freshness from storing state
- Time to recover
- For reference, Chandy-Lamport is non-blocking on normal execution, kind of fresh (some messages are lost), and time to recover is just ok

API Request Types and Workloads

- Service APIs: Supports many API methods, used to interace between client and server
- API Variability
    - Modifying storage and state → Reads/Writes
    - Accesses a lot of states → IO/CPU heavy
    - Accessing many items from state → implies coordination
- Workload Variability
    - Uniform load across methods
    - Read heavy vs Write heavy
    - Skewed access of a few keys
    - Load spikes/bursty traffic
- API Optimization
    - Caching works for read heavy applications (but sucks for write heavy)
    - Key value stores are assumed to be ready heavy for this course
    - Batch requests if IO/network costs are high
        - Send N requests to replica at once
            - Issue: If requests arrive infrequently, we might have to wait for a long time
            - Solution: Batch timeout
                - If T seconds have passed since last batch, send an incomplete batch

        - Throughput increases but latency increases too
    - Pipeline to maximize CPU usage during I/O
        - Before processing requests, send requests to backup in parallel

Primary Backup Replication (Fault Tolerance)

- Idea: Two servers (primary and backup) run concurrently
    - Primary interacts with client
    - Backup copies primary's state
        - Backup replaces primary if it dies
- Big Ideas
    - If primary fails, replace with backup
        - Backups need to be aware of who the new primary is
            - Naïve Approach: Hardcode primary into the app code
                - We have the same issue
            - Approach: Use a View Service
    - If backup fails, add another machine as backup
    - Sync primary to backup before anything is externally visible/observable to the user
        - This allows for rollbacks to be possible
        - It is ok for the primary to be out of sync if the change is not externally visible
    - Resend messages that are dropped. Use random number as hash to deduplicate
        - After some time, if no response, assume backup is dead and bring in a new backup
            - We must wait for the backup to acknowledge it has received the message
    - Transferring from Primary → Backup
        - Bootstrap
            - Snapshot of primary's state (send all state)
                - Requires serialization, sending over network, then deserialization into memory

        - i.e. it is slow
    - Steady state
        - Send every operation and all sources of nondeterminism
            - Includes external inputs/requests
- Issues
    - Sending a response and it fails
        - Client deduplication
    - Client has data which backup does not have
- Optimizations
    - Primary does not need to consult backup for all operations (specifically reads)
        - This is only OK if the backup is externally consistent with the primary

Implementing Primary Replicas

- Naïve approach: RSMs with logical clocks
    - Idea: Implementation is oblivious to clocks and uses a library to keep things in sync
        - Functions receive messages from client and syncs with backups before sending a response
    - Issue: Does not capture nondeterminism in execution
- Virtual Machines
    - Idea: Isolate application, monitor nondeterminism
        - Primary and backup execute on two VMs (ideally on different machines)
            - Primary logs inputs and nondeterminism (interrupts)
                - Logs delete after some time
            - Backup applies inputs from log
                - Primary waits for backup to return
            - Primary and backup monitor each other
- Log-based VM replication (Virtual Machine Monitors)
    - Idea: Primary and Backup VMM
        - Sources of nondeterminism are decided upfront by VMM
        - Primary and backup never diverge

- Approach:
  - Primary VMM sends log entries to backup VMM over the logging channel
  - Backup VMM (hypervisor) replays log entries
    - Stops backup VM at next input event or nondeterministic instruction
    - Delivers same input event as primary used
    - Delivers same nondeterministic value as primary

Lecture 6: View Service and Split Brain

View Service

- Idea: Maintain current membership of primary-backup service via a view
- Big Ideas:
    - View service changes view when:
        - Primary OR any backup fails
        - Periodically send heartbeats
    - Transitioning between views
        - View Service broadcasts view change to all replicas
            - If view service dies, clients cannot find primary
        - Primary syncs with new backup if there is one
            - New primary is up to date as we only promote a previous backup
        - New primary must ACK new view once backup is up to date
            - Can become stuck if primary fails during view change
- Scalability
    - Client does not need to contact view service before every operation
        - **Cache** view across operations
    - If client gets no response/bad response from replica it thinks is primary, it invalidates its cache
- Fault Tolerance
    - View Service is not on the critical load path
    - Assuming a reliable network, we can still end up in split brain

Split brain

- Two partitions of nodes disagree on the state of the system
    - Leads to arbitrary inconsistency of the system
- Solution: Primary must forward all operations to backups
    - i.e. get ACKs from backups that they too recognize the primary
    - Backups know who is primary as only a backup can be promoted as a primary

Lecture 7: Linearizability and Consistency Models

Desired Ordering Properties

- Total Ordering of Writes
    - Abstracts application into single thread process for the user
- Reads relative to writes
    - Once a read returns a particular value, all later reads return the same value or the value of a later write
        - i.e. No taking back values after confirming
    - Once a write completes, all later reads should return the value of the write or the value of a later write
        - Ensures freshest results

Trace: Sequence of requests and responses

- A trace is sequential if no requests overlap
- A trace is linearization of another trace if all overlapping events are made non overlapping and the order of non overlapping is preserved

Linearizability (Operational)

- Properties
    - Total ordering of writes
    - Read returns the last completed write
- Single copy semantics
    - Externally visible effects of writes and reads are equivalent to if there existed a single copy
- Linearizability is always defined based on a specification
    - Given a sequential specification S, an implementation I is linearizable iff for all traces t in I, there exists a linearization t' that belongs in S

Consistency Models

- Consistency models are properties that all executions of a given system satisfy
    - Performance and Consistency are on an inverse scale

- Eventual (Easiest to write)
- Read-after-write
- Causal
    - Idea: Order of causally related writes must be preserved in values returned to reads
        - Allows for lazy synchronization between replicas
        - If $W_1 \rightarrow W_2$, if a read sees the effect of W_2, it must see effect of W_1
- Sequential
    - Idea: There exists a sequential order of all events that can explain the execution
        - Read time order does not matter for independent clients
    - Same as linearizability but allows arbitrary reordering across clients
        - Global sequential order of all events across all clients
            - Requests within a client cannot be reordered
            - i.e. a global order of writes
                - Implemented using Lamport Clocks or having single primary doing all writes
                    - Increases blocking and performance overhead
    - Does not hold when clients are
        - No longer independent (use shared resources like channels)
        - creating diverging states
            - Should be prevented by the global order
- Linearizability (Most consistent, more expensive)

Lecture 8: Consensus, Safety, Liveness, Paxos

Safety and Liveness

- Systems can be defined via a set of execution traces
- Systems satisfy a property P if all of its execution traces satisfy a property
- We can prove a system does not satisfy a property via a counterexample trace
- Safety: System never reaches an undesirable state
    - Counterexample: a finite trace
    - Easier to prove, check, and guarantee
- Liveness: System makes progress eventually
    - Counterexample: an infinite trace

Consensus

- Issue: RSM is not tolerant to network partitions. Operations block when some machines are unavailable
- Idea: Apply update only if majority of replicas commit to it
    - If 2f + 1 replicas, we need f + 1 to commit
        - Higher tolerance if there are more replicas
        - 2f + 1 → Avoids ambiguous acceptance of other updates
- Assume replicas start in sync with one another
- Properties in Consensus
    - Termination: Each correct process eventually decides on a value (Liveness)
        - Race Condition prevents liveness: Between one proposer's Prepare and Accept phases, $n_{proposer}$ is updated by another proposer
        - Solution: Proposer sends propose messages to all accepters
            - Retry with higher proposal number if majority do not accept
    - Agreement: All correct processes decide on the same value
    - Validity: The agreed upon value must be one of the proposed values
- Consensus Phases: Propose → Decide → Accept

Paxos

- Processes must know the number of acceptors before the start
    - Processes can persist the value they accept
    - Impossible to have all numbers accept different values due to consensus
    - For f faults, 2f + 1 acceptors are needed
    - For f faults, f + 1 proposers are needed
- Pros
    - Only needs majority of replicas to be up
    - No centralized view service
- Cons
    - Needs two rounds of inter-replica communication
    - Arbitrary wait until next operation is accepted
- A process can play multiple roles
    - Proposers propose values
    - Acceptors contribute to choosing among proposed values
    - Learners learn the agreed upon value
- Idea: Protocol runs over multiple rounds
    - Once a value is accepted by a majority, a different value cannot be accepted in a later round
    - Prepare Phase
        - Proposer sends a unique proposal number to all acceptors
        - Waits for a majority commitment from acceptors
    - Accept Phase
        - Proposer sends proposed value to all acceptors
        - Waits for a majority acceptance
    - Learn Phase
        - Learners discover value accepted by the majority

Paxos (Technical version)

- Prepare
    - Create n, a proposal number (not a proposal value)
    - Proposer sends prepare(n) to all acceptors

        - n is unique, and higher than previous n that *this* proposer has used before
            - localized n allows us to avoid finding consensus over higher global n value
    - Acceptors (when receiving prepare(n))
        - Check if acceptor previously promised a higher proposal ($n_{acceptor} \leq n_{proposer}$)
            - If yes, respond with prepare-failed()
            - If no, check if acceptor has accepted any previous proposal
                - If yes, respond with promise(n, ($n_{acceptor}$, $val_{acceptor}$)) for previously accepted value
            - Otherwise,
                - Reply promise(n)
                - Set $n_{acceptor} = n_{proposer}$
        - The above is called the promise phase
- Accept
    - Proposer
        - When majority agree on promise(n),
        - Send accept(n, val) to acceptors
    - Acceptors (on receiving accept(n))
        - Check if acceptor previously promised a higher proposal ($n_{acceptor} \leq n_{proposer}$)
            - If yes, respond with accept-failed()
            - If no,
                - Reply accepted(n, val)
                - Broadcast accepted(n, val) to learners
                - Set $n_{acceptor} = n_{proposer}$
                - Set $val_{acceptor} = val_{proposer}$

Lecture 9: FLP, RSM with Paxos, Paxos Efficiency

Two General Problem

- Two generals on opposite sides of an invading army
    - If they both attack simultaneously, they can win
    - If they attack at. Different times, they all die
- They have no time to coordinate at all
    - Can only send messengers through the enemy camp
    - Messengers can be captured and killed

FLP Result (Fischer, Lynch, Paterson)

- The following situation possibly never terminates
    - Binary consensus (agree on 0 or 1)
    - Asynchronous network
    - One unreliable process (failures)

Liveness Solutions for Paxos

- When a proposal fails, back off for a random period of time before retrying
- Use a pre-determined ordering of proposers
- Elect a leader to propose requests

RSM with Paxos

- Idea: Apply all updates in a consistent order at all replicas
    - Available as long as majority is able to communicate
- Approach
    - Set of replicas (must be an odd number)
        - Replicas play all roles and agree on sequence of requests to be processed
    - Shared log of requests
        - Each replica has its own copy of the log
    - Paxos comes to consensus about each slot of the log
        - Results in performance issues
        - Requires implementation of garbage collection of log

    - If another requests arrives while another is pending, store it in a local queue of requests to apply

Improving RSM Paxos Performance

- Idea: Once a proposed value is accepted, jump directly into acceptance phase
    - Skips a round of communication per proposal
    - General industry practice too
- Desirability Property: Improve the best case without making the worst case worse

Leader-Based Paxos (Optimization)

- Use Paxos to elect a leader rather than selecting values via a shared sequence of views
    - Operate as primary-backup after election (all proposals sent to leader)
- Logistics
    - Leader Crash → Elect a new leader by deciding on a new view with Paxos
    - When to View Change → Make views have a validity period and make replicas send heartbeats to each other
    - Drawbacks → Leader may be far from the client

Lecture 10: Chubby, Zookeeper

Chubby

- General Info:
    - File system service for coordination
        - Each file is associated with a lock
            - Used to abstract away Paxos
        - Clients assume service doesn't fail
    - Chunk-master (leader)
        - If lock acquired, declare self as chunk-master
        - If not acquired, find the chunk-master
    - Chubby as a lock service (Service Architecture)
        - Libraries are maintained by users managing the machines
        - Services handle RPC calls and decouples deployment and operation
            - Maintained by organizations
    - Replicated Service using Paxos to implement fault-tolerant log
        - Chubby cells can have many locks
- Chubby Goals
    - High availability
        - Thousands of clients interact on small number of files
        - Event notification to reduce polling
    - Scalability (service cannot bottleneck)
    - Access control (avoids financial and security issues)
- Workflow
    - Chubby clients ping the Chubby service to acquire lock
    - If Chubby session with the lock dies, lock is released
        - Locks have timed lease to guarantee validity
        - Client sends heartbeat messages to service before expiration
        - If lease expires, lock is released
            - Assumes clock drift is bounded by <<<1s
                - i.e. we assume the lock is held longer than it actually is

    - Issue: Latency of storage service extends past lease expiration
        - i.e. effects (from non-Chubby services) can happen after a lock is released
    - Solution: Client passes a lockID object to storage
        - Storage checks with lock service whether lock is still valid
        - i.e. Lock ID piggybacking
- Scalability
    - Caching client data read
        - Issue: Local reads from cache can return stale results (violating linearizability)
        - Solution: Master invalidates cached copies upon update
            - Master stores all clients who may have cached a data item
            - Traffic is read heavy and Chubby has no latency requirements on writes
    - Proxies
        - Proxies batch heartbeats to forward less over network

Lock Granularity

- Fine-grained locks (grab lock per action/transaction)
    - Requires higher scalability of lock service
    - Brief lock service unavailability stalls everyone
    - Clients must be prepared to lose locks frequently
- Corase-grained locks (grab lock for minutes) (Chubby)
    - Scaling responsibility falls to clients
    - Unavailability (seconds) is OK
        - Client with lock can keep working
    - Locks are not lost

Reads in Paxos-based RSM

- Goals:
    - Linearizability: A read must see the effect of all accepted writes
        - Get read accepted to one of the slots in the replicated log

            - Concurrent processing does not matter
        - Issue: Poor performance at scale
        - Alternative: serve reads at all replicas
    - Good Performance
- Approach:
    - For every key, every replica stores (value, version)
    - Return latest version out of majority of replicas
    - Issue: Different replicas may have different versions

Zookeeper

- General Info
    - Open-Source coordination service
    - Vs. Chubby
        - Similarities
            - Service and not library
            - Small file-system like hierarchy
        - Differences
            - Lower level API allows for more flexible coordination patterns
            - Weaker guarantees for better performance
    - Establishes core primitives rather than locks
    - Leader runs consensus protocol called Zab (Zookeeper atomic broadcast)
    - Use a weaker guarantee to avoid blocking as much as possible
        - Chubby cache invalidation leads to arbitrary delays due to waiting to hear back from all clients
    - *Almost* linearizability
        - Writes are strongly consistent and ordered
        - All requests from a single client are processed in FIFO order
- Implementing locks in Zookeeper
    - Clients register to watch a file
        - Zookeeper notifies client when file is updated
        - Zookeeper does not need to block for reads

        - Zookeeper serves reads in any replica (vs only master in Chubby)
    - Lock implementation
        - Acquire a lock by creating a file
        - If file already exists, watch for update
        - Upon watch notification, try to re-acquire the lock
            - Safe as writes are strongly consistent and ordered
            - Caused extremely bursty traffic as all processes all fight for lock
    - Zookeeper writes do no invalidate caches
        - Zookeeper writes blocks for other writes
        - Writes hold the cost of computation
        - Blocking is necessary for strong consistency
- Optimizations
    - Client pipelines all updates in a FIFO so service can run them all at once

Lecture 11: CAP Theorem, Eventual Consistency, System Correctness, Conflict Free Replicated Datatypes

Strong consistency needs consensus (and blocking and order)

CAP Theorem: It is impossible for a distributed system to satisfy all 3 of the following:

- Consistency (Safety)
- Availability (Liveness)
- Partition Tolerance: (Unreliability)
    - CP and AP are most common
- Proof
    - Setup
        - Replicas in different network partitions cannot communicate
        - Partitions are transient and change
    - Responding without violating linearizability → Requires majority partition
    - Ensuring replicas always serve clients → Requires dropping consistency

PACELC

- If **P**artition: Choose **A**vailability vs **C**onsistency
    - Partitions are not the common case
- **E**lse: Choose **L**atency vs **C**onsistency
- Strong consistency means sacrificing latency

Order independent updates: Split the state of registers, clients can contact any replica

Eventual Consistency

- If no new updates, all replicas eventually converge to the same state
    - Liveness → Cannot be violated with a finite execution trace
        - Delivered updates might not be in the same state
- Issues
    - Lack of referential integrity
    - Updates may appear to be lost

        - Thus, Amazon S3 uses read-after-write consistency after initial put. Most data is write-once

Strong Convergence

- Replicas that have delivered. The same set of updates have equivalent state
    - Differentiates between same set of updates vs delivered updates

Strong Eventual Consistency

- Heavily application dependent → Application must be able to tolerate disorder
- Limited on reads

Techniques for Checking System Correctness

- Testing: Given a system, add set of assertions and checks
    - Lightweight and automatic, but limited guarantees
- Monitoring/Runtime Verification: Create a monitor system, run system in production and check monitor output
    - Lightweight and automatic, but limited guarantees and added overhead to production
- Model Checking: Model the system as a state machine, come up with a set of properties
    - Guarantees and automatic, but expensive and white box
- Deductive Verification: Come up with a set of properties and construct a proof that system satisfies properties
    - Strong guarantees, but manual effort and white box

Facebook TAO

- Social Graph Database:
    - Nodes are people, groups, pages
    - Edges are follows, friend relations
- Linearizability
    - Use monitoring to gather 12 days of execution traces
        - Generate logs for each processed request
        - Check if execution is linearizable

    - Issue: Validity due to Clock Synchronization
        - Comparison of time across different machines
        - Solution: Assume 99^{th} percentile skew of 35ms
            - 0.003% violations in linearizability

Strong Convergence

- Replicas that have delivered the same set of updates have equivalent state
    - Issue: Does not care about the resulting state
    - Solution: Require an algorithm for conflict resolution

CDRTs (Conflict Free Replicated Datatypes)

- Guarantee strong eventual consistency (as long as same updates are applied)
    - Registers, counters, sets, lists
- Operation-based synchronization
    - Each replica broadcasts all update operations
    - Replicas apply updates from clients once they are received
        - Requires updates to be commutable
    - Less communication overhead
    - Requries reasoning over partial orders of operations
- State-based synchronization
    - After each update, each replica broadcasts its state S to other replicas
    - Replicas execute merge every time they receive another replica's state
        - Merge function is customized: S x S → S
    - Simpler to reason about
    - States exchanged may be large

Lecture 12: Sharding, Data Lookup, Chord

Assumptions thus far

- Every replica has to handle every operation
    - Repeated work means we can only scale up to the slowest replica
- Every replica has to store the complete state
    - We can only support state size up to storage of smallest replica
- Solutions:
    - Vertical scaling: Upgrade the worst machine
    - Horizontal Scaling: Get more machines, fix protocols to adapt to new machines
        - Cheaper

Sharding

- Idea: Partition state of one machine across multiple machines
    - Statistical hashing for less key skew
- Issue: Not fault tolerant, if shard fails, the state is lost forever
    - Solution: Create replicas and run Paxos for each shard
        - Does NOT improve scalability

Centralized Data Lookup

- Idea: Have a lookup service to know the location of each item
    - Hadoop File System uses a Name Server and Data Server
- Contacting the lookup service:
    - When a new server joins (server learns items it needs)
    - When we have a new item (service assigns it)
- Straightforward protocol, but hierarchical and thus not scalable

Decentralized Data Lookup

- Edge Cases
    - Randomly assign data to nodes
        - Even distribution but hard to find items
    - Place everything on one machine

        - Easy to find items, but uneven distribution
- Naïve Approach: Assign key to nodes based on ranges
    - More even, but request distribution can still be uneven
    - Not all keys are used the same
- Hashing: Take a value and move it to an output space
    - Should minimize collisions
- Modulo Hashing: Compute modulo via number of servers
    - Very efficient but can lead to key skew
    - Additionally, changing machines requires moving all keys
        - Keys / Nodes is minimum number of keys to move
        - Ideally, K is large and N is small

Consistent Hashing

- Idea: Represent hash output space as a circle
    - Every node ID is also hashed and points to a point in the circle
        - Note: Hash is not equi-distance/uniform
            - Thus, some servers take double the keys
- Keys are partitioned across servers
    - To locate a key, hash the key and find its successor server
- Adding/Removing Nodes
    - Move K/N keys
        - Minimizes migration of state upon change in set of servers
    - When removing a node, the successor takes over the shard

Virtual Nodes

- Each server gets multiple random IDs
    - More IDs mean better load balancing
    - Each ID corresponds to a virtual node
    - Vary id numbers across servers (bigger node gets more)
- When adding a node, you need to move data across nodes

Data Lookups with Shards

- Successor Pointers
    - Each node knows who its successors are
    - If you do not have value for key, forward to successor
        - O(N) lookup
- Finger Table
    - Idea: Maintain log(N) pointers to reach O(1) lookup
    - i'th entry at node n points to successor of hash(n) + 2^i
        - number of entries = number of bits in hash value
    - Use binary lookup tree rooted at every node to find closest successor to value
    - Node Joins/Removals: All nodes know their predecessor
        - Update n's finger table
        - Update all other nodes' finger tables
            - Move data around accordingly
        - Periodically maintain successors and predecessors using a stabilization protocol
    - Fault Tolerance
        - Replace crashed node data at its closest successors
            - Maintain a successor list of size log(N)

Lecture 13: Amazon Dynamo, Data Store

Data Store

- NoSQL
    - Easier to get started and prototype
        - Good for small set of keys that each request needs to access
    - Lighter and thus more efficient. Flexible
    - Pushes more work onto the application
        - No schema
- SQL
    - Highly optimized for joins and other SQL constructs
    - Strong Consistency guarantees
    - Harder to implement

Service Driven Applications

- Frontend is ideally stateful
- Backend is ideally stateless (used to compute aggregates)

Service Level Agreements (SLAs)

- Contracts that describe service performance
    - Represents distribution of response times given load
- Fan out application:
    - End to End SLA Percentile:
        - For 3 backends, 90^{th} percentile 90ms
            - 0.9 * 0.9 * 0.9 = 0.73
            - End to End SLA = 73 percentile 100ms
    - Parallel requests to backend that get aggregated
        - More services entail harder guarantees for tail

Amazon Dynamo: Highly available distributed data store

Dynamo Motivation

- Datacenters across the world composed of thousands of servers and network components
    - Components fail continuously
    - Requires persistent state given failures

Dynamo Goals

- Sacrifice Consistency to maximize Availability

Dynamo Big Picture

- Provide KeyValue API (NoSQL) instead of SQL interface
    - No operations span across multiple keys
- Weak Consistency
    - Strong consistency not possible with partitions and availability
- Conflict resolution (for concurrent requests)
    - Resolved during reads → Reads uncover conflicts
        - Writes are never rejected, users can proceed if all is well
        - Dynamo supports Application and Data store conflict resolution
            - Application allows for hand-tuned conflict resolution at the cost of developer overhead
            - Data store is automatic but not natural

Dynamo Design and Implementation

Problem to Address	**Solution**
Partitioning	Consistent Hashing
High Availability for writes	Vector Clocks
Consistency	Sloppy Quorum
Temporary Failures	Hinted Handoff
Permanent Failures	Merkle Trees
Membership and Failure Detection	Gossip-based Membership

Dynamo Decentralization: All nodes do same work

- Helps with provisioning, maintenance, scaling, and reliability

Dynamo uses Virtual Nodes for Consistent Hashing

- Creates uniform data distribution, awareness to node heterogeneity

Dynamo Replication

- Replicate value for every key at N nodes in clockwise fashion
    - If node is a duplicate physical node, skip it
- Execution of writes
    - Write received by coordinator
    - Coordinator forwards to successors
        - Declare success if R reads or W writes succeed

Data Look Up

- Any server can receive and process requests
    - If server receives a key request, if it doesn't have the key, ask the next node (O(N) worst time)

Gossip

- Once per second, each server contacts a randomly chosen other server
    - Both server exchange their lists of known servers
    - If servers do not change frequently, results in O(1) Lookup time

Consistency via Sloppy Quorums

- Rather than consensus, select any available server to handle requests
    - N replicas for every key
    - Serve reads and writes by forwarding request to first N-1 reachable successors
    - Eventual Consistency but temporary inconsistencies
        - System supports custom merge operations

Vector Clocks

- Naïve Approach: Store a vector clock with each key-value pair
    - Not scalable, only a few nodes can write to item
- Dynamo Approach: Dynamic list of (coordinator node, counter) pairs
    - Determines if a version supersedes another

        - Tracks causal relationships between updates
        - Can check for concurrent requests, performs reconciliation via a custom merge function
    - Get(key) → List of <value, context> pairs
        - When responding to GET, Dynamo always returns the vector clock for the value returned
    - Put(key, value, context)
        - Context indicates which versions this version supersedes or merges
- Clock Truncation
    - Case: More nodes processes PUTs to same key (after fault or network partition)
        - Dynamo stores the time of modification with each version vector entry
        - When version vector is more than 10 nodes long, Dynamo drops the least recently processed key
            - Causes false concurrency
- Comparisons
    - Vs Chord
        - Simpler protocol, worse “worst case” scenarios, but those never ocme up in practice
    - Chubby/Zookeeper
        - Both are services for users
            - Dynamo is fault tolerant data store, Chubby helps build one
            - Dynamo is not strongly consistent and surfaces inconsistencies to clients

Lecture 14: Failures in Dynamo, Merkle Trees, Two Phase Locking

Failures in Dynamo

- Temporary: Node is unreachable temporarily or due to network partition
    - Issue: Suppose data must be written to 3 nodes to be successful
        - If one node is unreachable, we must either delay the response or respond without durability
    - Solution: Hinted Handoff
        - Coordinator tries writing to subsequent nodes (beyond N successors of key)
        - Coordinator informs recipient of the intended node (which is currently down)
            - Recipient keeps "hint" in its metadata
            - Once intended recipient is back, we forward the key they should have and delete as they are adequately replicated
        - Issue: Hinted Handoff node may crash before it finishes replicating data
        - Solution: Replica synchronization
            - Nodes nearby on ring periodically gossip
                - Compare (k, v) pairs, copy missing keys
                    - We need to compare all key pairs
- What if W=1?
    - High availability, no durability. If node fails, we lose access to some data
- If W > 1 and whole data center fails, we lose data. Thus, we construct preference list to span multiple data centers
- Permanent: Node is down permanently and won't return
    - Requires rebalancing by manual adding/removing ndoe

Efficient Synchronization with Merkle Trees

- Merkle tree hierarchically summarizes key value pairs a node holds
    - Leaf holds hash of one key-value pair
    - Internal node = hash of concatenation of children hashes

- Comparing roots: if match, values match
    - If no match, compare children. Iterate down
- Finding differences (Reconciliation)
    - Exchange and compare hash nodes from root downwards, prune when hashes match

Local vs Global Failure Detection

- Most failures in Dynamo are temporary
    - Thus, Dynamo does not gossip failure information and stores it locally

Performance

- Tail is much higher than average
- Real workloads are diurnal
- Writes take more than reads (due to disk access)
- Some applications need even lower 99.9 writes
    - Use pipelining: Each node pushes to local in-memory buffer
        - Sacrifices durability (if node crashes, buffer is lost)

Consistent Hashing with Merkle Trees

- Copying new data over requires recomputing Merkle Trees
- Solution: Partition space into statically fixed number of equal sized shards
    - Place shard on first N virtual nodes after its end
    - Thus, we only need to identify which shards to hand off when adding node

Sharding Coordination

- Idea: We must enable load balancing across partitions
    - If operations touch multiple partitions

Atomicity of Transactions (Sequences of Operations)

- All-or-nothing → However, can cause some operations to be lost
- Goal:

    - Serializability: For each execution of the system, there exists an equivalent execution that is fully serial
- Solution: Two Phase Locking
    - Apply locks on all necessary shards before attempting transaction
    - Algorithm
        - Client submits transaction to a coordinator (Transaction Coordinator)
        - TC acquires locks on all data involved
        - Once locks acquired, execute transaction and release locks
    - Ideas:
        - Disjoint transactions can execute concurrently
        - Related transactions wait on each other
- Alternative: Snapshot Isolation
    - Each transaction operation on a snapshot of the database can't read different data items from different times
    - Issue: Transaction cannot succeed if any partition becomes unavailable
        - Solution: Every partition should be implemented as an RSM → Higher availability

Lecture 16: Guest Lecture, Beaver, Cuttlefish, OrbWeaver

Partial Distributed Snapshots (Beaver – OSDI 2024)

- Challenge: Classic snapshot protocols (e.g., Chandy-Lamport) assume full control of all participants and closed-world causality — unrealistic in cloud microservices with external calls and partial observability.
- Solution:
    - Beaver introduces Partial Snapshots that capture causally consistent views even with unknown external interactions.
    - Uses Monolithic Gateway Marking (MGM) by repurposing existing SLBs (Software Load Balancers).
    - Optimistic Gateway Marking (OGM) handles coordination with multiple asynchronous SLBs.
    - Enables fast, zero-overhead, non-blocking snapshots suitable for debugging, analytics, and GC.

Fair Cloud Execution for Financial Exchanges (Cuttlefish – WIP)

- Challenge: Traditional exchanges achieve fairness via physical proximity and cable length. In the cloud, latencies are variable and paths are unknown.
- Solution:
    - Cuttlefish introduces a Virtual Time Overlay, where every operation is quantized in virtual time.
    - Ensures communication and computation synchrony through frozen/advanced virtual cycles, independent of real-time variability.
    - Provides fair and deterministic execution across heterogeneous systems and network conditions.

Opportunistic Failure Detection (OrbWeaver – NSDI 2022)

- Challenge: Traditional heartbeats trade off detection speed for overhead; message loss and failures are indistinguishable.
- Solution:

- OrbWeaver uses weaved streams, inserting tiny metadata packets into natural inter-packet gaps (on the order of 100ns).
- Guarantees high-fidelity, low-cost in-band telemetry and coordination (e.g., clock sync, failure detection) with minimal power/latency overhead.
- Uses programmable switches (e.g., Tofino) to achieve detection times <1μs at 100 Gbps.

Lecture 17: Cloud Native Applications, Microsoft Azure

Storage Types

- Blobs
    - Unstructured files with binary data
    - Media, log, text files
- Tables
    - Structured with schema – each row is a data item
    - User profiles, business data
- Queues
    - Message Delivery
    - Orchestrate Workflows

Cloud Native Applications

- Applications are composed of services
    - I/O data using blob
    - Intermediate and final analysis on tables
    - Orchestration and workflows with queues
- Idea: Cloud applications are orchestrated using workflows
    - Fan Out, Fan In, Timer
    - Developers don't deal with development
    - Providers can multiplex
    - Makes optimization and analysis hard

Windows Azure Storage

- Main Ideas
    - Global Namespace → finds data in huge storage
        - Uses DNS, Data is accessible through URI
        - All items in a transaction MUST be in the same partition
    - Disaster recovery: replication across data servers
        - Primary Backup Replication
    - Scalability and load balancing → Serve many users

- Multi tenancy → Dedicated storage leads to waste

Cloud Architecture

- Separation of services and clusters
  - Service/application deals with application tasks
  - Cluster managers handle other tasks

Azure Storage Architecture

- Storage Stamp: Cluster of storage nodes, one per account
  - Find account stamp using the location service
  - Stream Layer
    - Distributed file system, manages extents durably
    - Only allows reads and appends
      - Stream: Extent pointer seq
        - Cheap to construct, logical
      - Extend: Append block seq
        - Expensive to construct, physical
        - Sealed means no more appends
        - Append many small items together until they fill up an extent
    - Reversal of extents is fast due to modifying list of pointers
    - Stream Manager: Paxos replicated service, not on data path
    - Extend Node: Actual data servers, handles replication
  - Partition Layer
    - Maps abstractions into streams, manages transactions and consistency, handles scaling through sharding
    - Idea: Stateful operations are saved in a log via range partitions
      - Keeps track of materialized checkpoint of state and rest of commit log
    - Each partition is served by a different partition server

        - Partition Manager: Assigns partitions to Partition Server, load balances
        - Partition Server: Transactional guarantees for partition
        - Lock Service: Paxos lock service, Election for PM, Lease for PS to serve partitions
- Range Partition
    - Storage Stream
        - Metadata stream: All information to load a partition to a server
        - Commit log stream: Holds all operations since last checkpoint
        - Row Data Stream: Stores checkpoints of data
    - In Memory State
        - Memory table: All updates since last checkpoint
        - Row data cache: In memory version of checkpoints
- Workflow
    - Partition server starts serving a range partition by loading it
    - Update operations go to commit log and memory table
    - Reads first go into memory log and then to in memory checkpoint
    - When memory table or commit log reach a limit, make a checkpoint
- Pros
    - Optimized for streams
    - No need to replay entire operation log from beginning
    - No full checkpoint needed to recover
- Load Balancing
    - Partition Server checkpoints range partition
        - Stops serving traffic
    - Partition Server creates new streams
        - Very fast as we just change pointers
    - Modify new partition metadata streams

            - Start serving requests
    - Frontends
        - Stateless routers that handle authentication and data access
        - Optimizations:
            - Cache frequently accessed data
            - Streams data directly from the stream layer when possible
- Goal: Use stamps at 70%
- Stamp Replication
    - Intra Stamp
        - Synchronous, on critical path of request, durable
            - Low latency
    - Inter Stamp
        - Off critical path on per object basis, for disaster recovery
        - Optimal use of network bandwidth
- Eraser Coding
    - Reduces storage cost post-extent sealing, increases read latency
    - For any extent, store one original copy
    - Store (M+N) erasure coded chunks

Lecture 18: Spanner

Goals

- Scalable
- Multiversion
- Globally Distributed → Reliable
- Synchronously Replicated → Fault Tolerance and Locality
- Distributed/Cross Partition Transactions

Guarantees

- Externally consistent transactions (Strict serializability)
    - Also called real-time ordering guarantee
    - Bans reordering of non-overlapping transactions
- Snapshot isolation transactions
    - No need to lock, just serve reads for specific timestamp

Cross Shard Geodistributed Transactions

- Idea: Commit timestamps for transactions using TrueTime API
- TrueTime API
    - API exposes real-world clock uncertainty across machines
    - Guarantees depend on uncertainty bounds
        - If large uncertainty, Spanner slows down and waits
        - Uncertainty is reported conservatively for correctness

Spanner Architecture

- Big Picture
    - Split up into zones
        - Zones are added, removed, and replicated
    - Datatype is (key, timestamp) → value
        - Many timestamps of same key live in the Database (multi version)
    - Data is split into tablets, which compose spanservers
    - Storage of files and commit logs

- Replication Management
    - Paxos RSM for each tablet
        - Long lived leader and time-based leases
    - Writes go to leader to start proposal
    - Reads are served from any up-to-date replica
- Concurrency Control
    - Each leader has a lock table
    - Transactional operations acquire locks
        - Other operations bypass the lock table
- Cross-zone Transactions
    - For local transactions, a lock table is enough
    - For cross-tablet, we need a two-phase commit
        - One leader of tablets is chosen as coordinator
        - Each leader has a transaction manager data structure
            - State of transaction manager is started with Paxos
                - Strong consistency for leader faults
- TrueTime API
    - Each machine knows an interval for the real time it can have
        - TT.now(), TT.after(t), TT.before(t)
    - Idea: We can guarantee that T1.start < T2.end
        - Thus, we can establish causality
    - Implementation
        - GPS based timemasters, some atomic
        - Servers sync with timemasters
- Implementing Spanner guarantees
    - Paxos clusters
        - Replicas use Paxos for lease votes
            - Lease interval starts when leader discovers it has quorum, ends when some lease votes have expired
        - Lease votes are implicitly extended on successful writes to state
    - Correctness depends on lease disjointness

        - Each Paxos leader interval is disjoint from past and future leaders (they cannot overlap)
        - Ensuring lease disjointness
            - Issue a Paxos write to write the lease interval
            - Next leader waits for all intervals to pass before starting its lease
        - Optimizations:
            - Each replica follows a single-vote rule: If it grants a lease at time T, it waits until TT.after(T.end + lease_len) before it grantsr another
            - At least one replica will be common between any two consecutive leaders (Use quorum for lease votes)
- Read-write transactions
    - Spanner wants to provide strict serializability
        - Monotonically increasing commit timestamps per tx
            - Establishes a sequential order
            - Exploit disjointness to establish a monotonic timestamp across leaders
    - Real Time Ordering
        - If start of T2 occurs after the commit of T1, then T2 > T1
            - Commit timestamp s is after coordinator's latest time
            - Wait until my earliest time has gone beyond s before writing
        - Implementation
            - Writes are buffered at the client until commit
            - Client issues reads to leaders of groups
                - To avoid deadlock, the locks have a timeout
            - When all reads are done, start a two pahse commit
    - Two Phase Commit
        - First, client sends messages to all leaders
        - Non-coordinator (upon receiving)
            - Acquires write locks
            - Prepares timestamp T that is larger than any previously assigned

        - Logs the prepare(T) record using Paxos
        - Sends prepare message to coordinator
    - Coordinator leader (on receiving commit)
        - Acquires write locks
        - Chooses commit timestamp T_s that is greater than:
            - All prepare(T) timestamps
            - TT.now().end at the time of receiving commit_tx
            - All previous assigned transaction timestamps
        - Logs the commit(T_s) record using Paxos
        - Wait until TT.after(T_s)
        - Send message to client and other leaders
        - Other leaders log the write and then release the locks
- Serving snapshot reads
    - No need for blocking
    - Each replica has a t_safe time and can only serve reads up to it
        - T_safe = min(t_paxos, t_tm)
            - T_paxos = timestamp of highest applied Paxos write
            - T_tm depends on prepared transactions

Lecture 19: Microservices, Mucache

Microservices

- General Info
    - Splits functionality into multiple services
    - Issue: They can be written in multiple languages with different data representations
    - Solution: Serialization Libraries
        - Offers unified representation across frontend implementations of data
    - Communication between services using established APIs and RPC
    - Each service might have its own state and are deployed in their own machines
- Microservice Application Requirements
    - Need to find which other services exist in the network and how to call them
    - Authenticate requests from other services
    - Serialization/Deserialization
    - Encrypt requests for privacy
    - Monitor request-level metrics
    - Often implemented using a Service mesh (application independent)
- Service Mesh
    - Used to build microservices by being its functionality
    - Come together via a proxy/sidecar
        - Sidecar intercepts all incoming and outgoing communication
- Microservice Benefits
    - Mostly organizational
        - Allows for easier delegation of work into teams
        - If service fails, we can simply call the team versus the entire development staff
- Microservice Graphs
    - Note: edges imply one or more requests
        - Also, each service can be deployed on many nodes
            - Sharded for scalability

        - Replicated for fault tolerance
- Shortcomings
    - Performance worsens from sidecars
        - Data goes twice through network per request
    - Programmability, Correctness
- Optimizations
    - Improve performance of stack (serialization algorithms)
    - Running sidecar on the same VM
    - Colocating services
    - Caching

Caching

- Service calls without side effects can be cached and not remade
    - If downstream data has not been modified
- Issue: Coherence: Database updates affect validity of upstream caches
- Solutions
    - Backend only Caching
    - Custom invalidation protocols per application
    - Give expiry to cache items(Time-to-Live eviction per service)
        - Ambiguous to set the time of TTL

MuCache

- Automatic cache invalidation for microservice applications
- Goals
    - Correctness: Behaves the same as original
        - Determined using traces
        - Create traces of client requests. Should be in the original set of traces
        - Sometimes requires linearizability as subset is an impossible execution (needs synchronization)
            - Clients do not need real time ordering

        - Given a set of cache-enabled traces, there exists a reordering such that the new order is in the original
    - Low Overhead: Cache management off the critical path
    - Support Real Microservice Deployments: Sharding and dynamic call graphs
    - Plug and Play
    - Incremental Deployment: Support evolving graphs where MuCache is not deployed in all services
- Architecture
    - Developer declares read only methods
    - MuCache Components
        - Cache
        - Wrappers (Intercepts communication to database)
            - Library that wraps all calls to services and database
            - Adds messages to cache manager queue
                - Startcall, Endcall, Invalidatecall
            - Keeps total order of call starts, call ends, invalidations
                - If a call's results are invalidated by a later write, then the cache item upstream is evicted
        - Cache Manager (invalidates caches)
            - One thread per shard
            - Has Input message (FIFO) queue
- MuCache Invalidation
    - Diamond patterns
        - S1 may become stale if S4 is updated without notifying S2 or S3
    - Issue: Invalidation might overtake the save and create divergence
    - Solution: Calls are invalidated unnecessarily
        - Cache manager cannot differentiate between write and read
            - Thus, we invalidate the write and assume all reads are batched before any writes
- Dependency Tracking
    - Each call keeps track of its visited services

- Before checking the cache at another request going through, we see that visited services intersects with cache results
    - Thus we do not use the cache