Research Paper Notes

You

March 18, 2018

Abstract

Your abstract.

1 Amazon

1.1 Dynamo: Amazon's Highly Available Key-value Store

- System Assumptions and Requirements

  - Query Model: items are binary blobs that are uniquely identified by a primary key. No complex operations. No relational schema

  - ACID properties: data stores with ACID guarantees provide poor availability when scaling. Dynamo is good for applications with a looser consistency requirement. No isolation guarantees, so only single key updates

  - Dynamo is assumed to be run in a non-hostile environment at Amazon, so there are no security requirements.

- Design Considerations

  - Synchronous replication of data will give strong consistency, but poor availability in the presence of certain failures

  - Availability can be increased for systems prone to network or server failures by using eventually consistent replication techniques. Writes are propagated to replicas asynchronously and in the background. Conflicting updates must be resolved. When to resolve updates and who resolves them?

  - Update conflicts are resolved during reads so that Dynamo can be an "always writeable" store. This is in contrast to traditional data stores which handle resolution during writes in order to keep reads simple (this can lead to writes being rejected if they cannot reach or are not accepted by the replicas).

  - Either the data store or the application can be the entity that resolves write conflicts. Application developers have better knowledge of the schema and might be able to perform better resolutions. Having the data store resolve conflicts simplifies the application logic. A typical conflict resolution strategy at the data store level is "last write wins".

  - Dynamo should be incrementally scalable

  - Dynamo instances should be symmetric/homogeneous (i.e. no distinguished nodes with extra responsibilities)

  - Dynamo should be decentralized and use peer-to-peer techniques over centralized control. Amazon believes decentralization results in higher availability

  - Dynamo should be able to run on heterogeneous hardware so that portions of a Dynamo cluster can be vertically scaled without having to scale all the nodes at once.

- What Dynamo is built for

  1. Targets applications that need an "always writeable" data store with no rejected updates.

  2. Built for a network where all nodes are trusted

3. Targets applications that do not need hierarchical namespaces or relational schema

4. Built for latency sensitive applications. Multi hop routing for Distributed Hash Tables (e.g. Chord) incurs more latency and variability in terms of SLA. Dyanmo is a zero-hop DHT

- System Architecture
  - Techniques used by Dynamo
    * Consistent Hashing solves the problem of Partitioning with the advantage of Incremental Scalability
    * Vector clocks with reconciliation during reads are used to allow for high availability of writes with the bonus that Version size is decoupled from update rates
    * Sloppy Quorum and hinted handoffs solves handling temporary failures with the bonus of providing high availability and durability guarantees when some of the replicas are not available
    * Anti-entropy using Merkle Trees solves recovering from permanent failures with the bonus of synchronizing divergent replicas in the background
    * Gossip based membership protocol and failure detection solves membership and failure detection with the bonus of keeping membership information decentralized
  - System Interface
    * Exposes two operations $get()$ and $put()$
    * $get(key)$ returns the object associated with key or a list of objects with conflicting versions along with a $context$.
    * $put(key, context, object)$ stores the replicas of $object$ based on key and writes the replicas to disk.
    * $context$ encodes system metadata (such as the version of the object) about the object that is hidden from the caller. The $context$ is stored along with the $object$ in order for the system to do validation.
    * Both the $object$ and key are treated as an array of bytes. Dynamo computes an MD5 hash of the key to get a 128-bit identifier to determine the storage nodes responsible for serving the key.
  - Partitioning Algorithm
    * In order to scale incrementally, Dynamo needs a way to dynamically partition the data over the set of nodes in the system. Therefore, Dynamo relies on consistent hashing to distribute load across multiple Dynamo instances.
    * Consistent hashing works by treating the output of the hash function as a fixed circular array or ring (the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random position in the ring. The keys assigned to a node are all of the keys that lie between its position in the ring and its predecessor's position. The main advantage of consistent hashing is that departure or arrival of nodes in the system only effects the immediate neighbours of the node in the ring.
    * Basic consistent hashing algorithms can lead to non uniform data and load distribution since each node is assigned a random spot in the ring. Also it doesn't take into account the heterogeneous hardware that each node is running on.
    * Dynamo uses a modified consistent hashing algorithm where each node in the system gets mapped to multiple positions in the key space. Nodes can be responsible for many of these "virtual nodes" (single positions in the key space). When a new node joins the system it is assigned many multiple positions in the key space (also called "tokens").
    * Virtual nodes allow even load shedding across the rest of the nodes in the system if a single node leaves. When a node joins the system, it takes on roughly the same amount of load from each node. The number of virtual nodes a host is responsible for can be tuned to the amount of capacity the host has based on the hardware it is running on

- Replication
  * Dynamo provides high availability and durability by replicating data across multiple nodes.
  * Each node stores all keys that fall within its range locally and replicates them across the next $N-1$ successor nodes (where N is configurable per node) so that all data is replicated across N nodes. This implies that each node is responsible for storing keys in the range of itself and its Nth predecessor.
  * The $preferencelist$ contains the nodes which are responsible for storing a particular key.
  * Each node contains enough information so that it can determine the preference list for any key
  * The preference list has more than N nodes to account for failure
  * The preference list contains distinct nodes. When building the preference list, virtual nodes that belong to nodes that are already in the list are skipped.
- Data Versioning
  * Dynamo allows write to propagate asynchronously, providing eventual consistency
  * Dynamo treats an updated object as a new, immutable version of the object and allows for multiple versions of an object to be present within the system.
  * Newer versions of an object normally subsume older versions; however, this might not be the case in the presence of failures and concurrent updates. In this case where version branching occurs, the application must do conflict resolution on a subsequent read to collapse these versioned branches
  * Dynamo uses vector clocks in order to determine which versions of an object "happened before" other versions. A vector clock is a list of (node, counter) pairs and each version of an object has its own vector clock. If all of the counters in a vector clock are less than or equal to all of the counters in vector clock of a second version, then the first is an ancestor of the second and can be forgotten, else the two versions are in conflict
  * Updating an item in Dynamo is done by passing it a context (which contains the vector clock info) which was obtained by an earlier read operation. If a read request for an object detects that there are multiple versions of the object in conflict, Dynamo will return all of the objects and put all of the vector clock information for each version in the context returned. A subsequent write with this context is assumed by Dynamo to have reconciled all of the different versions, so it will collapse the multiple versions
  * Theoretically the size of the vector clock could grow to a large size if many nodes are coordinating writes for an object. In practice, this shouldn't happen since only the N nodes in the preference list should be handling writes for the object; however, network partitions might make this not be the case
  * Dynamo truncates the vector clock to a fixed size. It adds to each (node, count) pair a timestamp of when the node last serviced the object. If the clock is full it evicts the entry with the oldest timestamp. This can lead to situations where causal dependency between object versions cannot be accurately reconstructed, but Amazon has neither encountered or explored this
- Execution of get() and put() without failures
  * Operations are invoked over HTTP
  * Clients can either send requests through a load balancer to route to a node or link in a partition-aware library that routes requests directly to the right coordinator node
  * A loadbalancer might send a request to a node that is not in the N number of nodes for the keys preference list. This node will then send the request to the first node in the preference list for the key
  * Read and write operations use the first N healthy nodes in the preference list for a key, skipping unhealthy nodes. When there are node failures, lower ranked nodes (those not in the first N nodes of the preference list) will be employed

* Dynamo has two configurable values (R and W) which it will use in its quorum-like consistency algorithm to maintain consistency between replicas for a key. R and W are the minimum number of nodes that must participate in a successful read or write operation respectively. Setting $R + W > N$ yields a quorum-like model. However, since the latency of a ready (or write) operation is dominated by the slowest of the R (or W), replicas, R and W are configured to be much smaller than N
                    * For a put() request, the coordinator generates the new vector clock and version and stores the version locally. It then sends both to the N highest reachable nodes and waits for at least $W - 1$ successful responses before returning success.
                    * For a get() request, the coordinator requests all existing versions of data for a key from the N highest reachable nodes and then waits for R (not $R - 1$ I guess?) responses before returning success. If the coordinator gets multiple versions for the object, it returns all of the ones it deems causally unrelated (needing to be resolved)
            – Handling Failures: Hinted Handoff

2 Blockchain

2.1 Tezos a self-amending crypto-ledger White paper

Proposes a new cryptocurrency where users can vote to change the blockchain policy at any point.

- A blockchain protocol can be broken down into 3 distinct protocols

    – Network protocol: discovers blocks and broadcasts transactions

    – Transaction protocol: determines what makes transactions valid

    – Consensus protocol: forms consensus around a unique chain

- A blockchain protocol is fundamentally a monadic implementation of concurrent mutations of a global state. This is achieved by defining blocks as operators (functions) acting on this global state. The free monoid of blocks acting on the genesis state forms a tree structure (set of all possible block combinations). A global, canonical, state is defined as the minimal leaf for a specified ordering.

- Let $(S, \leq)$ be a totally ordered, countable set of possible states. Note: a totally ordered set is a set X along with a binary relation $\leq$ such that

    – Reflexive: $a \leq a$ for all $a \in X$

    – Antisymmetric: If $a \leq b$ and $b \leq a$ then $a = b$ for all $a, b \in X$

    – Transitive: If $a \leq b$ and $b \leq c$ then $a \leq c$ for all $a, b, c \in X$

    – Totality: $a \leq b$ or $b \leq a$ for all $a, b \in X$

    Totally ordered sets form a category and can be understood as partially ordered sets with additional structure

- Let $\emptyset \notin S$ be a special, invalid state

- Let $B \subset S^{S \cup \{\emptyset\}}$ be the set of blocks (note that $S^{S \cup \emptyset}$ is the set of functions with domain $S \cup \emptyset$ and range S). The set of valid blocks in $B \cap S^S$

- We extend the total order on S with $\forall s \in S, \emptyset < s$. I believe this means $\emptyset$ is the bottom of S. This order determines which leaf in the block tree is the canonical one

- Any blockchain protocol can be identified by $(S, \leq, \emptyset, B \subset S^{S \cup \{\emptyset\}})$

- In Tezos blocks can not only act on the global state but also the protocol itself. The set of all possible protocols is defined recursively as $\mathcal{P} = \{(S, \leq, \emptyset, B \subset S^{(S \times \mathcal{P}) \cup \{\emptyset\}})\}$. The last element of the tuple is the formalisation that blocks can act on both state and the protocol itself.

4

- The network shell
  - knows how to build the block tree
  - Knows of 3 types of objects: transactions, blocks, and protocols. In Tezos, protocols are OCaml modules used to amend the existing protocol.
  - the hardest part of the newtwork layer is to protect the user against DOS attacks
- Clocks
  - Every block has a timestamp
  - Blocks with timestamps a reasonably amount of time in the future can be buffered, others are rejected
  - Must be able to deal with falsified timestamps and user clock drift
- Chain Representation
  - A raw block header contains: a hash to the previous block, a bytestring consisting of the block header (where transactions in this block are), a list of operations, and a timestamp in floating point notation.
  - State (or Context) is represented by an immutable, disk-based key-value store. Blocking on disk operations is avoided using asynchronous operations
  - A protocol is defined by several operations:
    * Parsing operations to translate block headers and operations from byestrings into ADTs.
    * An apply operation which takes a state, a parsed block header, and a parsed list of operations, and returns the next state. This is the monadic abstraction talked about in the beginning of the paper: A blockchain protocol is a monadic implementation of concurrent mutations of a global state.
    * a score function that projects a State onto a list of bytes so that States can be compared (first by length then by lexicographic order). This comparison function is the total order ($\leq$) on the set of states described in the intro of the paper (ties are broken based on the hash of the last block). Converting State into a bytestring also allows us to states across different protocols.
  - Ammending the Protocol
    * Two protocols: one for testing and the current protocol
    * Can swap out either protocol with a stakeholder vote
    * When a protocol is swapped out, the global State changes, and the new protocol takes effect when the next block is applied to the State.
    * A protocol is an OCaml module that is compiled on the fly and run inside a sandbox (no system calls can be made)
    * the protocol is used by the *apply* function when computing a new *State*.
    * Many rules can be voted in to amend or change the protocol
  - Protocols expose an RPC mechanism so that GUIs can be made
- Seed Protocol
  - Tezos starts with a seed protocol
  - There is a finite amount of coin in the system
  - Relies on a combination of bonds and rewards to incentivize miners to secure the blockchain
  - Bonds are one year security deposits that are purchased by miners and forfeited if they break the rules
  - After a year, miners and endorsers receive a reward for their service as well as their security deposit back

- Inactive addresses are not selected to propose blocks or vote for blocks in the consensus algorithm
- Protocol amendments are proposed and voted over a period of time every 3 months. A quorum is needed for an amendment to pass.
- During the first quarter of voting, participants propose the hash of a a tarball of an OCaml module representing the new protocol. Active users can choose to "approve" any number of the proposed protocols
- During the second quarter, the amendment receiving the most approval from the first quarter is subject to a vote. Voters can vote yes, no, or abstain (which counts towards the quorum).
- During the third quarter, if quorum was met and the amendment received a threshold of yes votes, then the protocol is adopted into the test network and the minimum quorum for future votes is adjusted based on the number of participants
- During the fourth quarter, users vote again to promote the test protocol to replace the active protocol
- Proof of Stake
  * Blocks are mined by stakeholders and signed by signers. Mining and signing provide rewards to users but require a short term safety deposit that is forfeited in the event of malicious activity.
  * Groups of coins are grouped into coin rolls. Coin rolls are assigned priorities for signing the next block
  * Any stakeholder can sign a block. However, each stakeholder is subject to a random delay of how long they must wait before they can propose a block. This delay is determined by the priority assigned to any coin rolls they own (as I understand it)
  * Additionally 16 signers are chosen for each block. A blockchains weight is not its length but how many signatures it has.
  * Signers are incentived to sign quickly the best priority block it knows about in order to receive a reward
- Smart Contracts
  * Tezos uses stateful accounts
  * An account is simply a contract that has no executable code
  * contracts incur storage fees since they are stored in the blockchain (I guess?)
  * Contract code is only allowd to execute for a certain amount of "steps". Multiple contracts can be used to build larger programs. The threshold can of course be changed with a protocol amendment

2.2 Secure High-Rate Transaction Processing in Bitcoin

proposes the "Greediest Heaviest Observed Subtree" (GHOST) protocol extension to scale transaction confirmation for bitcoin to be able to handle a high enough transaction rate to meet global economic demands

- Nakamoto's original paper on bitcoin assumed that blocks could be transmitted through the overlay network much faster than they are created. The more transactions, the more frequently blocks must be created or the larger the block size must be and the longer it will take for new blocks to propagate throughout the network.

- If the amount of transactions confirmed per second by the network does not rise, then transaction fees will raise and drive people to use other payment systems

- In Nakamoto's original paper when multiple valid blocks are mined, a fork occurs and is not resolved until some chain of the fork becomes longer than the others. In the case of ties, a node simply adopts the block it learned of first as the main chain

- The GHOST rule is an alternative rule for conflict resolution when forks occur. At each fork in the chain the heaviest subtree rooted at the fork is selected.

- Ethereum is using a variant of the GHOST protocol

- At a higher transaction confirmation rate, miners that are better connected may enjoy more rewards than there share of the hashing compute power in the network. Also selfish mining may become a viable strategy for weaker miners. To combat this, GHOST has an extension described in a different paper

- Double spend attacks occur when a user pays a merchant some fee and then cafts a set of blocks (longer than the current honest main chain) without that transaction (or directing that transaction elsewhere) in order to replace the main chain. However this is not computationally feasible unless the attacker has more computational power than the entire honest network combined.

- Model
    - Let V be the set of nodes in the bitcoin network and E be the set of of edges connecting nodes. The network can then be modeled as a directed graph $G = (V, E)$.
    - Every node $v \in V$ has some fraction of the computational power of the network $p_v \geq 0$ such that the total computational power in the network can be defined as $\sum_{v \in V} p_v = 1$.
    - **Aside on Poisson Processes:** A Poisson process is a type of counting process that tries to model the occurrence of random events over time (the arrival of customers at a store or the number of earthquakes that will happen in a city). λ is called the rate or intensity of the process and represents the rate at which the random event will happen per unit time.
    - The entire network creates blocks following a Poisson process with rate λ with each node generating blocks individually following a Poisson process with rate $p_v \lambda$. The value of $\lambda = \frac{1}{600}$ (which I think is supposed to be 1 block per 600 seconds = 10 minutes) was chosen by Nakamoto in his original paper.
    - The time it takes to send a block across an edge $e \in E$ is denoted as d_e.
    - The block creation rate of the honest network is $\lambda_h = \lambda$. The attackers creation rate is denoted as $q\lambda_h > 0$ for some $0 < q < 1$. Furthermore, it is assumed that attackers build chains efficiently (meaning that an attackers' chain has minimal forks because it is able to coordinate its mining nodes to reach consensus near instantly when it creates a new block)
    - $time(B)$ is the absolute creation time for a block B. The structure of the tree at time t is denoted by $tree(t)$ and by $subroot(B)$ - the subtree rooted at B. Furthermore $depth(B)$ denotes the depth of a block B in the block tree.
    - The function $s()$ is defined to be the choice function that selects which block a new block should build off of. Formally, it maps a block tree $T = (V_T, E_T)$ onto a block $B \in V_T$. It should be noted that it is possible for different nodes to have different views of the block tree because of network propagation delays when announcing new blocks.
    - The Bitcoin protocol originally defined $s()$ as selecting the block with the largest depth in the block tree. As such, $longest(t)$ defines the deepest leaf block in $tree(t)$ at time t.
    - The "main chain" is defined to be the path from the genesis block to the block that $s()$ selects to be the next block to build off of (originally in Bitcoin this was $longest(t)$). The time it takes for the main chain grows from length $n-1$ to n is a random variable defined as τ_n. The rate of block addition to the main chain is defined as $\beta = \frac{1}{E[\tau]}$ where $\tau = lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} \tau_n$. I think this is essentially saying that the rate the main chain grows is defined in terms of the expected value of the infinite series of random variable main chain growth times. It should be noted that λ denotes the rate at which blocks are added (not necessarily to the main chain) to the block tree.
    - The maximum block size in kilobytes is represented as b, and it is assumed that newly created blocks are always the maximal size
    - The primary measure of Bitcoin scale is defined to be Transactions Per Second (TPS) the system adds to the main chain and is defined to be $TPS(\lambda, b) = \beta(\lambda, b) \times b \times K$ where K is the average number of transactions per KB. Since β measures block creation over time, b measures kilobytes, and K measures transactions transactions per kilobytes, we get the transactions per seconds.

- The amount of throughput in the network determines how susceptible Bitcoin is to double-spend attacks. If an attackers compute power is is greater than the main chain block addition rate ($p\lambda_h > \beta$), then the attacker's chain will always win out in the end eventually. Conversely if $q < \frac{\beta}{\lambda_h}$, then the probability of the attacker winning decreases exponentially as the main chain grows in length. Therefore $\frac{\beta}{\lambda_h}$ is defined to be the security threshold of the system.

- Transaction throughput is controlled mostly by block creation rate (λ) and block size (b).

- Some naive attempts to increase throughput pose increased security risk to Bitcoin:
    - **Larger Blocks:** Increased block size cause more forks because of longer block propagation delays in the system. Propagation time has been shown to grow linearly with block size
    - **Accelerated Block Creation:** If blocks are created faster by the honest network (larger λ_h), then there is a loss of efficiency because nodes will be more likely to contribute to an outdated fork as opposed to the main branch (some nodes will not be fully synced on the most up to date state of the tree). Conversely attackers will also have increased block creation ($b\lambda_h$) but will not experience any of the same efficiency loss because of the assumptions that attackers have full, efficient control over their resources.

- The main issue here seems to be that trying to scale transaction throughput using the longest chain rule leads to more forks because nodes cannot reach consensus as quickly because of increased block propagation delays. As a result this becomes an attack vector bad actors can exploit with secret mining.

- Greediest Heaviest Observed Subtree (GHOST):
    - GHOST allows for cryptocurrency protocol designers to set high block creation rates and block sizes.
    - Originally, the main chain in the Bitcoin protocol is chosen to be the longest chain. At a high level, GHOST changes the weight of a chain so that it takes into account blocks that "hang" off the main chain but are not a part of the main chain (essentially forks that hang off the main chain contribute to the weight of the main chain).
    - Formally, GHOST redefines the parent selection function ($s()$). For a block B in a tree T, let $subtree(B)$ be a subtree of T rooted at B and let $Children_T(B)$ be the set of blocks directly referencing B as their parent block. Defined $GHOST(T)$ to be a parent selection function that follows this algorithm with input block tree T:
        1. $current\_node := GenesisBlock$
        2. if $Children_T(current\_node) = \emptyset$ then $return(current\_node)$
        3. else $current\_node := node\,from\,Children_T(current\_node)\,whose\,subtree\,is\,the\,largest\,size$
        4. goto step 2

        It should be noted that the size of a subtree is directly correlated with the hardest combined proof-of-work

- Properties of GHOST:
    - Every block is either fully abandoned or fully adopted into the main chain
    - The probability that a block is initially on the main chain and then becomes off the main chain (50% attack) can be made arbitrarily small given a sufficiently long waiting time τ after the blocks' creation. This means the security threshold ($\frac{\beta}{\lambda_h}$) can be made to equal 1, meaning that the attackers' block creation rate must equal that of the honest network.
    - The rate of growth of the main chain (β) using GHOST is slightly slower than using the longest chain rule. However unlike the longest chain rule, this slowdown in growth does not effect the security of the GHOST rule.

- The authors attempt to measure the efficiency of the GHOST rule and provide a framework for protocol designers to set security parameters (e.g. block creation rate) using simulations of overlay networks since the entire network for a cryptocurrency protocol cannot be known beforehand. I mostly skimmed this section, but its interesting to note in case I would ever need to do something similar

- Although the security threshold ($\frac{\beta}{\lambda_h}$) of GHOST is always 1, if the throughput of the grows limitlessly then bandwidth becomes an efficiency bottleneck

- Their simulations show that the GHOST rule and longest chain rule both can scale TPS at approximately the same rate as block creation increases (however longest chain suffers from decreased security threshold while GHOST remains at 1)

- The acceptance policy (time a merchant is willing to wait before considering a transaction as confirmed) can be modeled as a function $n(t, r, q)$ where r is the risk the vendor is willing to take and q is the upper bound on the attackers' compute power and t is the time elapsed since the transaction was broadcast to the network. Vendors wait until $n \geq n(t, r, q)$ blocks have piled up on the transaction block before considering the transaction as being confirmed.

- blocks using GHOST get confirmations from all other blocks in its subtree.

- Changes in the implementation of GHOST:
    - Only send the headers of off-main-chain blocks since all nodes must know of these blocks.
    - Deployment of GHOST rule can be gradual since it is mostly compatible with the longest chain rule at low block creation rates
    - Only award coins for mining to block creators that mine blocks that end up on the main chain. Block creators that create blocks that end up in the same subroot should not be awarded coins
    - To avoid attackers adding blocks to off-main-chain branches long in the past (where the difficulty target might be lower), it is suggested to using a checkpointing system that prevents addition of new blocks prior to some block in the tree

- A final note: I don't believe that at this time Bitcoin uses the GHOST protocol. Ethereum uses a modified rule which awards coins to block creators that contributed to blocks that hang off the main chain. Also there seems to be some criticism stating the GHOST would centralize mining because well connected nodes benefit or something or other, I dunno it kinda seemed like the jury was out plus these were just comments on reddit and y combinator so who knows if the commenters actually knew what they were talking about

3 Consensus

3.1 Paxos Made Simple

Simple description of the Paxos algorithm for distributed consensus

- Consensus Algorithm
    - Problem Statement
        * Have a collection of distributed processes which can propose values
        * Want a consensus algorithm that ensures that only a single value out of the proposed values is chosen such that 1. Only a value that has been proposed can be chosen 2. Only a single value is chosen 3. A process never learns that a value is chosen unless has actually been chosen
        * There are three types of roles in the algorithm: *proposers*, *acceptors*, and *learners*.
        * Processes communicate with one another by passing messages and we use an asynchronous Non-Byzantine model: 1. Processes operate at arbitrary speeds, can fail by stopping and restarting, has stable storage to remember information (if not then solution is impossible since all processes could fail after value selection) 2. There are no bounds for message delivery delays and can be duplicated and lost but not corrupted.

- Choosing a Value
  * A simple solution is to have a single *acceptor* which accepts the first value it receives. **Problem:** Single *acceptor* becomes single point of failure
  * Another way it to have multiple *acceptors* which *accept* different proposed values from processes. A value is chosen when a majority of *acceptors* have *accepted* it (this works because any two majorities have at least one *acceptor* process in common)
  * **P1. An acceptor must accept the first proposal it receives**
  * In order to satisfy **P1**, we must allow *acceptors* to be able to *accept* multiple values.
  * A proposal now consists of a unique proposal number (to uniquely identify proposals) and a value that is being proposed.
  * **P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.**
  * **P2A. If a proposal with value v is chosen, then every higher-numbered proposal** *accepted* **by an** *acceptor* **has value v**. We must still satisfy **P1**, so we must strengthen **P2A** to **P2B**
  * **P2B. If a proposal with value v is chosen, then every higher-numbered proposal that is proposed by a proposer has value v.**
  * Note that since *acceptors* accept proposals which are proposed by proposers, we have **P2B → P2A → P2**

3.2   The Part-Time Parliament

Describes the Paxos algorithm for reaching consensus in a distributed system

- Paxons' Parliament Description
  - Primary task was to determine the law of the land
  - Each member kept track of a numbered ledger to record the laws that were passed by the parliament
  - First requirement was that member's ledgers were consistent (i.e. no two ledgers could contain conflicting information). If member A has decree X written at position N then member B cannot have decree Y written at position N in her ledger (she can leave position N blank if she has not yet learned of decree X).
  - Note that fulling the consistency condition can be made trivial (all members leave their ledger blank), there must also be a progress condition - If a majority of the members remained in the Parliament Chamber without anyone entering or leaving for a sufficiently long period of time, then any decree proposed by a member would be passed and all passed decrees would appear in the ledger of every member

4   Facebook

4.1   SVE: Distributed Video Processing at Facebook Scale

Describes the evolution and architecture of Streaming Video Engine (SVE) which handles video uploads at Facebook scale.

- 3 key requirements of video processing at Facebook scale: low-latency, flexible API for application developers, and fault tolerance and resiliency.

- The video encoding engine validates uploaded videos and re-encodes the video into many bitrates and file formats in order to support a wide array of clients and stream video at the best quality based on network conditions.

- Facebook orginally had a system dubbed Monolithic Encoding Script (MES) for uploading and processing videos, but it did not scale well.

- SVE adds parallelism in 3 ways that MES did not: it parallelizes video validation and processing, it chunks videos into smaller pieces and parallelizes processing of the chunks over a compute cluster, and it parallelizes video processing with video storage + replication.

- SVE's programming model has developers writing tasks that work on a stream-of-tracks abstraction that forms a Directed Acyclic Graph (DAG). The stream-of-tracks abstraction breaks a video down into streams (video, audio, metadata, etc.), and the DAG model allows tasks to be easily composed and parallelized.

- 6 steps in the full video pipeline: record, upload, process, store, share, stream.

- Users upload video to Facebook's front end server which then forwards it to SVE for processing.

- The video is then validated which includes repairing the video (syncing audio + frames or fixing metadata) and validation (blocking malware and DOS attacks).

- The video is next re-encoded which is the most computationally demanding step.

- Once encoded, the video is stored in the Binary Large Object (BLOB) storage system (seems to be Facebook's in-house datastore).

- When streaming, video clients download video chunks at the best bitrate it can sustain from Facebook's CDN.

- This paper focuses on the pre-sharing + streaming phase of the video pipeline.

- Monolithic Encoding Script:

  - Had a simple architecture: Client uploads a file to the front end server as a single opaque binary blob. Once upload is completed, the front end server forwards the file directly to storage where it is picked up be a processing server which runs through the processing tasks sequentially with one encoding server handling a single video and then restored.

  - Worked but was not very flexible when adding new tasks or video applications. Hard to add monitoring

  - Not very reliable and crumbled when experiencing overload.

- Existing batch processing frameworks such as MapReduce, Dryad, and Spark all assume that the data to be processed already exists (has been uploaded). This incurs high latency

- Existing stream processing framework such as Storm, Spark Streaming, and Streamscope allow for overlapping of upload and processing but are designed to process continuous queries as opposed to discrete events.

- Steaming Video Engine Overview

  - Each server type is replicated many times, but only one server from each type handles a video.

  - 4 fundamental changes from MES architecture:

    1. The uploading client tries to break the video up into segments (called group of pictures or GOP) to upload. Each GOP is processed and encoded independently. This segmenting allows processing to happen earlier since the front end server doesn't need to wait for the entire video to be uploaded to start forwarding it to the processing servers

    2. The front end server forwards video chunks to a preprocessor server instead of directly to storage

    3. The MES server has been replaced with the preprocessor, scheduler, and workers of SVE.

    4. SVE exposes pipeline construction in a DAG interface that allows application developers to quickly iterate on the pipeline logic while providing advanced options for performance tunings

- The Preprocessor does lightweight processing (which includes validation and breaking the video up into GOP segments if an old upload client does not support this), initiates heavyweight processing, and acts as a write-through cache since it writes video segments to main memory as well as passing them off to persistent storage for redundancy. Writing the segments to storage is done concurrently with heavyweight processing, so having the Preprocessor cache video segments helps move the Storage (and its slow use of disk) off of the critical path.

- Video encoding is heavyweight since it operates at the pixel level and is done on a bunch of worker machines which are orchestrated by the Scheduler. The tasks to be run are determined by the DAG of tasks for the application that the video is being processed for.

- The Scheduler receives the DAG job from the Preprocessor as well as notifications for when video segments have been preprocessed and are available. The scheduler pushes tasks to a cluster of workers. Each cluster has a high and low priority queue of remaining tasks they can pull from. The priority of a task is set by the application programmer. Workers will always pull from the high priority queue over the low priority queue. SVE shards video encoding by video ID. This means that a DAG job might be handled by many Schedulers during the course of its runtime due to capacity and load balancing demands; however, at any given time a DAG job is being processed only by a single Scheduler.

- Workers process tasks in the DAG job for a video in parallel. They receive tasks from the Scheduler, pull the input data to complete the task either from the Preprocessor's memory cache or from Intermediate Storage, and then write the task output either to Intermediate Storage (for further processing) or to persistent Storage.

- Intermediate Storage is used in a variety of ways. The type of storage being used depends on the type of data being written. Application metadata is stored in a multi-tiered storage system that durably stores the data and makes it available through an in-memory cache because it is read much more frequently than written. The in-memory processing context of SVE is written to durable storage without caching since it is written many times and read only once. Video and audio data are written to a BLOB store and automatically freed after a few days. Facebook prefers having the storage layer automatically let the data expire before freeing it because they believe this leads to a simpler, more correct design.

- Uploads are normally bottlenecked by the speed + upload bandwidth of the client. The solutions to speed this up is to either upload less data or parallelize uploading with encoding/processing. Another option is to process the video client side, this is done in SVE when the video is large and will effect pre-sharing latency and the client has the right specs to do this. Client-side processing of the video comes with tradeoffs because you're using the resources of the clinet (battery)

- Segment size of video chunks when uploading is a tradeoff between compression and parallelization across segments. A larger segment size provides for better compression, and a smaller segment size provides for better parallelism

- Syncing the video to durable storage can incur a significant amount of latency for some videos. SVE decreases latency by overlapping processing video segments with storing them. SVE also caches some segments in memory so that load times are quicker. For more on this design read the paper **Rethink the Sync**

- DAG Execution System

  - Programmers write sequential programs that act as tasks (vertices) in the DAG job. Edges of the graph represent dataflow and dependency.

  - A stream-of-tracks abstraction is used to specify the granularity at which tasks run. A task can run on one or more tracks (such as video or audio tracks) or it can run for all segments within a track.

  - Each DAG job is dynamically generate per-video by the Preprocessor. It probes the video and runs DAG generation code based on its findings (bitrate, etc.)

– Once the DAG is created, the Preprocessor forwards it to the Scheduler which is in charge of dispatching tasks among workers and monitoring the progress of tasks and the job. The Preprocessor will alert the Scheduler when new video segments are available for processing.

– Tasks can be organized into "task-grouops" which are collections of tasks that will all be run on one worker. Each task is by default in their own task-group. task-groups allow application developers to amortize scheduling overhead. Tasks can also be marked as "latency-sensitive" which means a user is waiting for the task to finish. All parent tasks of a latency-sensitive task are also marked as latency-sensitive.

- Fault Tolerance

  – Workers that detect task failures (exception or failure exit code) will retry the task a few times and then propagate the failure to the Scheduler which will also retry scheduling the task among different workers. If the task continues to fail after retry, the task is marked as failed and the DAG job will fail if the task was critical.

  – User client's devices will anticipate intermittent uploads

  – The fron-end server will replicate state externally

  – The Preprocessor will replicate state externally

  – The Scheduler will synchronously replicate state externally

  – The Worker will replicate in time

  – Tasks will retry

  – The Storage will replicate over multiple disks

- Overload

  – Organic overload occurs when some event causes video upload and processing demand to spike beyond normal weekly spikes. New Years Eve is a good example.

  – Facebook's loadtesting framework purposefully overloads SVE to make sure that overload diagnostics are working as intended

  – Overload can also be induced by bugs in SVE (i.e. a memory leak causes the pipeline to slow or stall)

  – SVE first reacts to overload by delaying the scheduling of latency-insensitive tasks

  – SVE will then react to overload by shifting load to another regional SVE (this is done manually be an engineer)

  – Finally SVE if the other measures do not mitigate overload, SVE will delay processing of newly uploaded videos until older videos have finished processing.

5 Scalability

5.1 Harvest, Yield, and Scalable Tolerant Systems

Defines Brewer's CAP Theorem.

6 Paper Backlog

- **Post-quantum RSA** by Daniel J. Bernstein, Nadia Heninger, Paul Lou, and Luke Valenta

- **A Fistful of Bitcoins: Characterizing Payments Among Men with No Names** Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, Stefan Savage