Research Paper Notes

You

June 10, 2018

Abstract

Your abstract.

1 Apache

1.1 Spark: Cluster Computing with Working Sets

- Introduction
  - Spark is a cluster computing framework that targets applications where a working set of data is reused iteratively. Iterative jobs (such as some machine learning algorithms) and Interactive analytics (such as querying large amounts of data in a database) are prime examples of such applications
  - Spark's main abstraction is that of Resilient Distributed Datasets (RDDs) which are a read only collection of data that is partitioned across multiple machines with fault tolerance
- Programming Model
  - Programmers write a driver application which contains high level instructions for performing a parallel computation
  - The two main abstractions for parallel programming are Resilient Distributed Datasets and Parallel Operations
  - RDDs
    - RDDs are a collection of data objects distributed over a cluster of computers such that any partition can be rebuilt if a node fails
    - RDD handles contain enough information to rebuild the elements of the set from data in reliable storage. This means that elements of an RDD need not be held in physical storage and can be rebuilt if a node fails
    - RDDs can be built in 4 ways:
      1. By parallelizing a collection (e.g. an array) in the driver program into slices to be sent to worker nodes
      2. By transforming an existing RDD using operations such as flatMap, map, and reduce (Note: this suggests that RDDs form a Monad)
      3. By changing the persistence of an RDD. By default RDDs are lazy and are discarded after use. An RDD can be reused either by enabling cache hinting (e.g. cache the RDD if possible) or saving the RDD after operations have been performed on it
    - Spark supports the reduce, collect, and foreach parallel operations on RDDs. Reduce combines elements of an RDD using an associative function for collection at the driver program. Collect sends elements of the RDD to the driver program (this allows easy updates of collections such as arrays). Foreach passes each element of an RDD through a function which has side effects (e.g. copying the data to another filesystem, etc).
    - Spark supports two notions of shared state: Broadcast variables and accumulators

    * Broadcast variables are read only pieces of data that each worker node has access to. This allows programmers to only copy this data to each worker once instead of having to bundle it in every closure it sends to the workers (Note: this seems to me like a case of the Reader Monad)
    * Accumulators are values that only workers can add to using an associative function and which only the driver can read. This can be used to implement counters. Note: accumulators are Monoids.
  - Implementation
    * Spark is built on top of Apache Mesos which is a framework for cluster computing
    * RDDs are stored as a chain of objects forming a lineage. Each dataset points to its parent dataset and stores enough information to know how the parent was transormed.
    * RDDs internally all implement the same interface: getPartitions() which returns a list of partition ids, getIterator(partition) which iterates over a partition, and getPrefferedLocations(partition) which is used for task scheduling to achieve data locality.
    * Spark creates tasks for each partition of a parallel operation and attempts to send these tasks to preferred worker nodes using delay scheduling. Once a worker receives a partition, it calls getIterator() to start iterating over it.
    * Shipping tasks to workers involves shipping both the closures used to defined the RDD and the ones used to operate on it. Spark uses Java's serialization to ship closures
    * Broadcast variables are initially stored in a shared file system and then cached by worker nodes on demand. The serialization format for a broadcast variable is a path to the file storing its data
    * Accumulators are created for each thread running a task on a worker and "zeroed-out" after the task completes. Workers send the final value of task accumulators to the driver program for collection. Serialized accumulators contain a unique id and the "zero value" for the accumulator
  - Related Work
    * RDDs are an abstraciton of Distributed Shared Memory (DSM)
    * RDD and DSM differ in two ways: DSM normally uses checkpointing to recover from node failure while RDD uses cheaper lineage information to rebuild lost partitions, and RDDs push computation to the data while DSM operates on a global address space
    * The key component of Spark is allowing data to persist across iterations of a distributed job
    * Lineage on datasets is a well studied area

2 Amazon

2.1 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
    * Dynamo uses a "sloppy quorum" approach to coordinate reads and writes in order to maintain high availability in the face of network failures and partitions
    * If a node in the top N nodes of the preference list for a key is not healthy, another node (not in the top N) will be chosen
    * The node will receive metadata indicating which node was the intended recipient of the key originally.
    * The node will keep this keep in a separate local database and will periodically try to push keys in this database to their original receiver node once those nodes have come back online.
    * In order to survive data center failures, the preference list for a key should be configured such that it contains nodes spanning many different data centers
- Handling Permanent Failures

    * Hinted handoff works best if system membership churn is low and network failures are transient
    * To combat permanent failures, Dynamo uses an anti-entropy protocol along with Merkle trees to synchronize replicas
    * The Merkle tree builds a hash tree of all of the keys a virtual node is responsible for
    * Replicas compare Merkle trees with one another and fix inconsistencies
- Membership and Failure Detection
    * Dynamo uses an explicit mechanism to signal the permanent addition or removal of a node from the system (e.g. an operator use a CLI or browser to connect to a dynamo node and issue a membership request)
    * The node that services the membership change request writes the details of the request and a timestamp to persistent storage. This info is then propagated via a gossip protocol where nodes choose a peer at random to connect to every second and reconcile their membership tables
    * When a new node initially starts up, it chooses its own virtual nodes and persists this information. During membership reconciliation, nodes also reconcile key partition information. This allows nodes to have knowledge of which parts of the key space are owned by its peers in order to be able to route requests for those keys
    * Some nodes act as "seed" nodes which are known to all nodes. These nodes are typically well connected in the system and prevent logical partitions from being formed since all nodes will communicate with the seed nodes. Seed nodes are discovered via some configured or external method
    * Dynamo nodes only have a concept of local failure of other nodes (i.e. Node A might consider node B to be unhealthy if it is unresponsive even though node B is responsive to node C).
    * Dynamo doesn't need a globally consistent notion of failure because of the explicit node join and leave mechanism
- Bootstrapping New Nodes
    * New nodes that enter the system take over the keys of the tokens it has chosen in the key space
    * Older nodes that currently own these tokens detect the new node and offer to give the new node the keys it owns
    * Once the new node accepts the offer and receives the tokens, the older nodes can forget their tokens
    * Confirmation round ensures the new node doesn't receive any duplicate transfers for the same key range
- Implementation
    * 3 components: request coordination, membership and failure detection, and a local persistence engine
    * Request coordinator is built using an event-driven solution
    * Request handling is implemented as a state machine
    * When servicing a read, if a response from a replica node is received by the coordinator after it has sent a response to the client and the replica node's response indicates that the objects the coordinator returned were stale, the coordinator will do read repair and update the read replica nodes with the up to date version of the object
    * Write coordinators are typically chosen to be the node in the preference list that responded the quickest
- Experiences and Lessons Learned
    * Services either employ business logic or "last write wins" to handle conflict resolution
    * Configuring N, W, and R really effect availability, durability and performance. Most services use (3, 2, 2) for (N, R, W)

* An optimization for writes is to have each node maintain an in-memory cache of objects. Writes are stored in the cache and a writer thread periodically flushes the cache to persistent storage. It should be noted this trades durability for performance. Coordinator nodes can also choose one replica to do a durable write such that the node will only store the write in its persistent storage, this helps mitigate total loss of writes during failures
* Experience shows that divergent versions numbers are an extremely rare phenomenon and mostly occur when there are a large number of concurrent writes for the same object (normally initiated by an automated client)
* Another approach to request routing is to have it all be on the client side. A client could download membership information from a Dynamo node every 10 seconds and send requests to the correct node and bypass a hop to the load balancer or to the correct coordinator node (in the case of writes). Pull model is used because it scales better with many clients
* Nodes must be careful not to allow the scheduling of background tasks (hinted handoff, replica synchronization) to create resource contention that could slow down foreground tasks (put, get). A monitor task monitors the nodes resource utilization and reserves slices of time for the background tasks to run

3 Apache

3.1 Zookeeper: Wait-free coordination for Internet-scale systems

Zookeeper is a service for coordinating distributed processes

Zookeeper appears to be Chubby but without the *open* or *close* locking methods

4 Blockchain

4.1 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.
- 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

4.2 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

5 Consensus

5.1 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** $\rightarrow$ **P2A** $\rightarrow$ **P2**

5.2 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

6 Facebook

6.1 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.

7 Google

7.1 The Chubby lock service for loosely-coupled distributed systems

Chubby is a lock service that allows distributed processes to synchronize their actions and reach agreement on info about their environment

- Goals of Chubby are reliability, and availability, easy to understand semantics
- Uses Paxos to solve distributed consensus and leader election. Augments with clocks to achieve liveness

- Provides course-grained locking - locks are expected to be held by the primary for a long duration
- System Structure
  - Two main components: server and a library used by clients
  - A Chubby cell consists of 5 replicas that choose a master via a consensus protocol. The master is chosen by a majority vote between the replicas and is guaranteed by to be the only master for a short interval known as the master lease time.
  - Each replica maintains a simple database. Only the master coordinates reads and writes to this database. The replicas only copy updates from the master via the consensus protocol
  - A client trying to connect to the Chubby service first must locate the master of the Chubby cell. It will send a master identification request to the replicas listed in the DNS. Replicas respond to this query by returning the identity of the master. The client then directs all future queries to the master unless the master becomes unresponsive or indicates it is no longer the master
  - The master propagates write requests to all replicas and returns only once a majority of replicas have accepted the write request. The master services read requests itself since it is guaranteed to be the only master (note this is only safe if the master lease has not expired)
  - If a master fails, the replicas will select a new master once the master lease is up. This normally takes only a few seconds
  - If a replica fails for a long enough period of time, a monitoring recovery system will notice and start a new replica on a fresh machine. It will also update the DNS record and replace the IP address of the failed replica with that of the new one. The master periodically polls the DNS record and will learn of the new replica. The master will update the replica list in the database it maintains which will then get propagated to the other replicas via the consensus protocol. The new replica will eventually get a copy of this database via the consensus protocol but starts itself using backups of the database and information from the other replicas. The new replica is allowed to vote in the master election once it has responded to a write request from the current master
- Files, directories, and handles
  - Exposes a UNIX like filesystem containing a tree of files and directories
  - An example path is $/ls/foo/wombat/pouch$. ls (lock service) is the prefix for all Chubby names. foo is the name of a Chubby cell that resolves to a Chubby server via DNS. $local$ is the name of a special Chubby cell that represents the Chubby cell local to the client (normally in the same building)
  - In order to allow different Chubby masters to serve files in different directories, there isn't an operation to move a file between directories, directory modification time is not recorded, and file access is controlled by permissions on the file itself
  - The system does not expose file access times in order to make it easier to cache file metadata
  - Files and directories are called nodes (this is poor naming in my opinion)
  - No symbolic or hard links for a node within a Chubby cell
  - Nodes may be permanent or ephemeral. Nodes may be deleted and ephemeral nodes are also deleted if no client has them open.
  - Ephemeral files are temporary and used as a means for indicating that a client is still alive to other clients
  - Access to nodes is maintained by ACL files which reside in a well-kown part of the Chubby cell's namespace. Each node's metadata points to 3 different ACL files (readfile, writefil, ACL modification file).
  - ACL files contain names of users that are permitted to access the node in some way (read, write, modify ACL permissions)

  - Nodes' metadata also includes other fields: an instance number that is greater than the instance number of any previous node with the same name, a content generation number that increases when the files contents are written (not for directories), a lock generation number which increases when a node's lock transitions from $free$ to $held$, and and ACL generation number which increases when the node's ACL names are modified
  - A 64-bit file content checksum is also exposed so clients can tell if a file differs
  - Clients open nodes to obtain handles which are like file descriptors. Handles contain: check digits which prevent clients from forging handles (ACL permissions need only be enforced at handle creation), a sequence number which signals to the Chubby master if the handle was created by itself or a previous master, and mode information which is provided at handle creation which allows a newly restarted master to recreate its state if an old handle is presented to it

- Locks and sequencers
  - Each Chubby file or directory can act as a reader/writer lock
  - One client may exclusively hold the lock in write mode while many readers can share the lock in read mode
  - Locks are advisory, meaning clients need not hold any locks to interact with files. However; attempts to hold the same lock may conflict
  - Acquiring a lock in either mode requires write permission so that an unprivileged reader cannot stall a writer
  - A problem that can occur in a distributed environment that uses is locking is the following situation: Process A acquires lock L and sends request R but then fail after sending R. Process B then acquires L and sends message $R2$ which arrives before R because of delays. R will now act on data without the protection of L
  - Sequence numbers are introduced to all messages that make use of a lock in order to guard against message delays
  - A lock holder may request a *sequencer* - an opaque byte-string that describes the state of the lock after acquisition. The *sequencer* contains the name of the lock, lock generation number, and mode of the lock. The client passes the *sequencer* to other clients if it expects operations to be protected by the lock. The recipient clients must validate that the *sequencer* is valid (this can be done by checking its Chubby cache).
  - Chubby provides an (imperfect) additional guard against message delays. If a lock becomes free because of client failure, Chubby does not permit other clients from acquiring the lock for a configurable duration known as the *lockdelay*

- Events
  - Clients may subscribe to events when they create a handle: file contents modified (normally used to monitor the change in a service advertised via a file), child node add/removed modified, Chubby master fail over, a handle + lock has become invalid, lock acquired, conflicting lock requests
  -

- API
  - *Open*() takes a file or directory name relative to an existing directory handle (a valid handle to the '/' directory is always maintained by the client library) and returns a handle to that node. Client additionally specifies the mode to open the handle in, events to listen for, the $lock-delay$ time, and initial file contents + ACL list if the node to be opened will be newly created
  - *Close*() closes an open handle and never fails
  - *Poison*() invalidates any outstanding and future requests made using a handle without closing the handle
  - *GetContentsAndStat*() returns the full content + metadata for a file. Files are read atomically

- *GetStat*() returns just the metadata and *ReadDir*() returns children + metadata for a directory handle
- *SetContents*() writes data to the file handle. Files are written atomically. Client may optionally add a generation number. The contents will then only be written if the supplied generation number is more current. *SetACL*() does the analogous operation for ACL names
- *Delete*() deletes the node if it has no children
- *Acquire*(), *TryAcquire*(), and *Release*() all deal with locking
- *GetSequencer*() obtains a sequencer describing any lock held by this handle
- *SetSequencer*() associates a sequencer with a handle. All subsequent operations on the handle will fail if the sequencer becomes invalid.
- *CheckSequencer*() checks if a sequencer is valid
- Handles are only associated with a specific *instance* of a node rather than the node name itself. So operations on a handle for a node will fail if the node is deleted even if it is subsequently recreated
- ACL permissions are always enforced during *Open*() calls and may be enforced on subsequent calls
- To make calls asynchronous, all operations may take a callback
- Master election can be performed using Chubby's API as follows: all potential primaries for a service open the lock file and attempt to acquire the lock. One succeeds and becomes the leader. The leader writes its address in the lock file using *SetContents*(), so it can be discovered by the replicas and clients of the service. Clients + Replicas learn of the primary's address using *GetContentsAndStat*() (probably in response to a file modification event). Ideally, the primary creates a sequencer with *GetSequencer*() which it passes to all servers it communicates with. Servers should verify the primary is still the current master by verifying the sequencer using *CheckSequencer*().

- Caching
  - Since most Chubby traffic are read requests, it is desirable to employ caching in order to reduce the amount of read traffic
  - Chubby clients cache file data and node metadata in a consistent, write-through cache held in memory. The Chubby cell master keeps track of what each client *may* be caching and sends cache invalidations to clients in order to keep their caches consistent
  - When a write request for file data or node metadata comes into the Chubby master, it blocks the operation and sends (via KeepAlive RPC responses) cache invalidation events to all clients which may have cached the data to be changed. Once the Chubby master has received acknowledgement of the cache invalidation event from all of the clients it contacted (either through another KeepAlive RPC request or the expiry of the client's cache lease), it unblocks the write operation. Since writes are only a small portion of Chubby's traffic, its okay for them to be slower
  - While the Chubby master is waiting for clients to respond to cache invalidation acknowledgments for a node, it treats the node as uncachable, meaning clients sending read requests for the node will not cache the node. However, this means that reads requests never have to block
  - Clients acknowledge a cache invalidation event by removing the cached data and then making a KeepAlive RPC request
  - Client caching is simple - clients can only cache and invalidate objects. There is no updating cached objects. If updates were allowed, clients that accessed a file once and then never again could potentially get a large number of unnecessary cache update events
  - Clients can also cache open handles. If a client calls *Open*() on a node that it had previously opened, only the first RPC call will make it to the Chubby master. Clients may also cache locks

- Sessions and KeepAlives
  - Sessions between clients and Chubby cells are maintained via periodic KeepAlive RPCs. Sessions guarantee that all of the client's cached data, locks, and handles remain valid while the session is valid.
  - Clients request sessions by contacting the Chubby master. Sessions are ended either explicitly or when the client has idled for too long
  - Sessions have a lease time during which the Chubby master may not terminate the session. The master is free to extend this lease time
  - Session leases are extend in three circumstances: Upon session creation, when master failover occurs, and when a master responds to a KeepAlive RPC.
  - When a Chubby master receives a KeepAlive request, it blocks the call until right before the client's session lease is about to expire. It then returns a response to the client with the new session lease time. The client then immediately sends another KeepAlive RPC.
  - KeepAlive RPCs are also used by the master to send events and cache invalidations to the client. A master will return a KeepAlive RPC response early if such an event has occurred. This simplifies the Chubby protocol and allows client - master communication to work through firewalls which only allow establishing connections one way
  - The client maintains a local session lease timeout which is a conservative approximation of the master's lease timeout. This means the master's clock must not advance faster than the client's clock by no more than a constant factor
  - When a client's local lease time expires, it can't be sure if the master has terminated its sessions, so it enters into a *sessionjeopardy* mode. The client empties and disables its cache and tries to establish another KeepAlive RPC with the master with a timeout. If the client and master exchange a successful KeepAlive RPC, the client enables its cache again; otherwise, it assumes its session is terminated. This is so that the client does not block indefinitely waiting for RPCs to be returned from the Chubby master
  - The Chubby library will notify the application if its session goes into jeopardy with an event. The library will also send either a session $safe$ or *expired* event once the session either recovers or is confirmed to be ended.
- Fail-overs
  - When a Chubby master fails or loses mastership, it discards all of its in-memory state about sessions, handles, and locks. The timer for sessions leases also stops during this time since it only runs at the current master.
  - When a master failure or change occurs, clients session can potentially go into *jeopardy* depending on how long the mastership change takes
  - When a new Chubby master is elected and a client has contacted it, the new Chubby master must recreate the in-memory state of the old master. It does this by reading data on disc (obtained through the database replication protocol), by obtaining state from clients, and by conservative assumptions. The database records each session, held lock, and ephemeral files
  - New elected masters:
    1. Picks a new *epochnumber* which must be sent by clients on every call to the master. The master will reject client requests using an older *epochnumber*
    2. The master may respond to new master location requests but does not when first processing a session
    3. It builds the in-memory data the old master had for sessions and locks from the database. It also extends session lease
    4. The master now lets clients perform KeepAlive requests but no other session-related operations
    5. It emits a mastership change event to all sessions. This lets clients know to flush their caches and warn their applications of potentially lost messages

6. The master waits for all sessions to acknowledge the mastership change or let their session lease expire
7. The master allows all operations to proceed
8. If a client uses a handle created from a previous master (based on the sequence number of the handle), the master recreates the in-memory representation of the handle and accepts the requests. If a recreated handle is closed, then the master notes this so that it cannot be recreated in *this* (but possibly future) master epoch
9. After a short interval, the master deletes all ephemeral files that have no open file handles. Therefore, clients should promptly refresh handles on ephemeral files after a mastership change

- Database Implementation
  - Chubby initially used the replicated version of Berkley's DB
  - Google however chose to write their own replicated database using write-ahead logging and snapshotting similar Birell *etall*
- Backup
  - Every so often the master of a Chubby cell writes a snapshot of its data to a GFS file server in another building. Different buildings provide data redundancy in the case of building damage as well as eliminating cyclic dependencies between GFS cells and Chubby cells (since a GFS cell might use the Chubby cell in its building for mastership election)
  - Backups allow newly replaced replicas to be bootstrapped without placing load on existing replicas in the cell
- Mirroring
  - Chubby allows mirroring of files between cells.
  - Mirroring is most commonly used to deploy configuration files around the globe
  - A special Chubby cell called *global* has a subtree called */ls/global/master* which is mirrored at every Chubby cell under the subtree */ls/$cell/slave*. Each member of the *global* cell is spread out across the globe so that any client can contact this cell
  - The */ls/global/master* subtree contains Chubby's own ACL info as well as various files with locations of Chubby cell's where certain services (such as BigTable) are advertised
- Mechanisms for Scaling

8 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

8.1 MapReduce: Simplified Data Processing on Large Clusters

- MapReduce is a framework that allows programmers to automatically parallelize and distribute computations across a large cluster of commodity machines
- Programming Model
  - A MapReduce computation takes a set of input key/value pairs and produces a set of output key/value pairs by applying user-defined functions map and reduce to the dataset.
  - Map takes an input pair and produces an intermediate set of key/value pairs. the MapReduce framework then groups all intermediate keys with the same value and passes them to the reduce function

- The reduce function takes an intermediate key and a set of values for that key and produces a potentially smaller set of values for that key

- Implementation
  - Map invocations are distributed across machines by automatically partitioning the input dataset. The intermediate keys produced from map are further partitioned into R sets and are run throught the reduce funciton. Map, reduce, and R are supplied by the programmer
  - Execution order
    1. When MapReduce is first called, it splits the input data into partitions and forks a bunch of worker processes that will handle the map and reduce phase as well as a master processes for assigning partitions to workers
    2. Worker processes read their assigned input data partitions and run each parsed key/value pair throught the map function and buffer the intermediate key output in memory
    3. Map Workers periodically buffer the output intermediate key/value pairs to local disk and send these disk locations to the master process who is responsible for forwarding this info to the reduce workers
    4. Reduce workers are notified of the intermediate key/value pair disk locations on the map workers and read this data via RPCs. Once it has read all intermediate data, it sorts the data and groups similar keys together since many different keys may be mapped to the same reduce worker. An external sort is used if the data to be sorted is too large to fit into memory
    5. The reduce worker passes each unique key and all of the data for that key to the reduce function and saves the output to a file
    6. When all map and reduce workers are done, the master wakes the user program and returns control back to the user code
  - When a MapReduce job is completed the output is consolidated into R output files which the user may access
  - The master process stores several pieces of information for each worker process. It stores the state (idle, in-progress, or completed) and the identity of the worker machine (for non-idle tasks). The master also stores the R disk locations of the intermediate output emitted by the map tasks. this information is pushed incrementally to in-progress reduce workers
  - To detect and handle worker failure, the master sends a healthcheck ping in small time intervals. If a worker is marked as failed, all map tasks it has completed and all map and reduce tasks that are in-progress are reset to an idle state and rexecuted by another worker. Completed map tasks must be reexecuted because the intermediate results are stored on the local disk of workers. On the other hand, completed reduce tasks need not be recomputed since they are stored in a shared global file system. Reduce workers are also notified of map worker failures and are told the new worker disk location to read the task data from
  - Master workers can take checkpoints of the state of its data structures periodically. Master failure is unlikely since there is only a single master process. The paper implementation of MapReduce simple aborts the entire MapReduce job if there is a master failure
  - To try and save on network bandwidth, the master process tries to schedule map tasks on workers that already have the partition of the data it needs on local disk
  - Typically the initial M partitions of the input data is chosen to be much larger than the number of worker machines while the R output files are chosen to be a much smaller multiple of M
  - "straggler" tasks, tasks where the worker is taking an unusually long amount of time to complete a task, are a common reason why MapReduce jobs can take a long time to finish. To alleviate this, the master node will schedule backup tasks of all in-progress tasks on different machines when a MapReduce job is close to completion. The master marks the tasks as complete whenever either the original or the backup tasks complete

- Refinements
  - A useful extension is to allow users to supply the partitioning function that creates the R partition of output files
  - Intermediate key/value pairs are guaranteed to be processed in increasing order. This is useful when the output files need to be accessed in efficient random order
  - A combiner function can be supplied by the user to save on network bandwidth. A combiner function is essentially just a reduce function that gets run on the map task output on the map worker before being sent to a reducer worker
  - MapReduce allows for users to implement a reader interface for reading in input data for the MapReduce job and a writer interface for outputing data
  - Users can create auxiliary, temporary files during a map or reduce function; however, it is the responsibility of the user to make operations on these files atomic
  - MapReduce allows optional skipping of bad records that make jobs deterministically fail. When a record causes a worker to fail, a signal handler sends a UDP packet containing the record id that caused it to fail to the master. If the master sees multiple failures across different workers for a record, it indicates that the record should be skipped
  - The master exposes an HTTP server with the state of all tasks and workers in the MapReduce computation
  - MapReduce allows users to create statistic and diagnostic counters. Counters are named and are incremented in the map or reduce function. Tasks increment counters during execution and periodically send the counter values to the master for aggregation (this is done within the healthcheck ping response). The master displays counter values on the status page and returns the values to the user code after the MapReduce job has been completed. The master also handles duplicate counters that arise from task rescheduling or backup tasks
- Performance

9 Replication

9.1 Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial

Author: F.B. Schneider

- State Machines
  - A state machine consists of a set of variables which encode the state of the machine and a set of commands which deterministicly transform the state of the machine.
  - Clients interact with state machines by issuing requests to execute commands.
  - Because state machines execute requests one at a time in the order they were received, clients can assume: O1) Requests issued by a single client to a given state machine are processed in the order they were issued. O2) If a request r made by client c causes a request r' to be issued by client c', then r is processed by r' (I believe this is a causality condition outlined in one of Lamport's papers)
  - The defining characteristic of a state machine is that it is a deterministic process that reads in a stream of requests and occasionally produces output.
  - Outputs of a state machine should be completely independent of time and other system activity and should only be determined by the sequence of input requests. This means that state machine commands should not implement any sort of time varying functionality.
  - State machines implement procedures and clients implement procedure calls
- Fault Tolerance

- Component failures occur in practice when a component no longer follows its specification.
- Two classes of failures are considered: Byzantine failures and Fail-stop failures
- Byzantine failures occur when a component exhibits arbitrary or malicious behavior, sometimes colluding with other faulty components (Note: Byzantine failures are problematic for permissionless systems such as the blockchain).
- Fail-stop failures occur when a component switches to a failure state that is detectable by other components
- Most applications can assume just fail-stop failures will occur, but critical systems should assume that Byzantine failures may also occur
- To measure fault tolerance, the notion of *t fault tolerance* is introduced. A system of distinct components is *t fault tolerant* if it can continue to operate according to its spec if no more than t of those components fail.
- Statistical methods (such as the mean time between failures and the probability of failures in a given time interval) can also be used to measure the fault tolerance of a system. However, Schneider argues that using *t fault tolerance* as a measurement for system fault tolerance is advantageous because it makes explicit the assumptions needed for a system to work and is not tied to the reliability of the components of the system (as statistical measurements normally are). The value of t is normally computed via statistical methods

- Fault-tolerant State Machines
  - A $tfaulttolerant$ state machine can be implemented by running replicas of the state machine on different processors in a distributed system
  - If Byzantine failures are assumed in a $tfaulttolerant$ system, then an ensemble of state machine replicas must have at least $2t + 1$ replicas to survive t failures. The output of the replicated state machine is the output of the majority of the replicas.
  - If fail-stop failures are assumed in a t fault tolerant system, then the ensemble must have $t + 1$ replicas. The output of the ensemble is any of the outputs of the replicas
  - Replica coordination is needed to implement $tfaulttolerant$ systems and can be accomplished by making sure that every non-faulty replica receives every request (Agreement) and processes requests in the same relative order (Order).
  - If we assume only fail-stop failures, then the Agreement condition can be relaxed for read-only requests since they can be sent to any non-faulty replica (instead of all replicas).
  - The Order condition can be relaxed for requests that commute - i.e. requests whose order do not change the final state of the state machine.
  - Agreement
    - Agreement can be obtained by any protocol that allows a designated process, called the transmitter, to broadcast a value to the other replicas such that the following are satisfied: IC1: All non-faulty replicas agree on the same value. IC2: If the transmitter is non-faulty, then all non-faulty replicas use its value as the one on which they agree
    - These types of protocols are often called Byzantine agreement protocols, reliable broadcast protocols, agreement protocols (consensus protocols?).
    - The transmitter can either be the client sending the request or a special replica. If the client is not the transmitter, then special care must be taken to make sure that the transmitter does not lose or corupt the request (this can be done by including the client in the set of processors in the agreement protocol).
  - Order
    - In order to satisfy the Order requirement for Replica Coordination, the replicas must process all requests in some total order. This can be accomplished by assigning a unique identifier to each request and only processing requests once they become

stable (i.e. once a replica is guaranteed that no request from any correct client with a lower identifier can be received). Therefore, replicas only processes the next stable request with the lowest identifier

* Note that the method for assigning unique identifiers to requests is constrained by O1 and O2, meaning that if request r is causaly related to request r' (r processed before r'), then $uid(r) < uid(r')$
* One way we can create a total ordering of requests is to use Logical Clocks (as described by Lamport in his seminal paper). Each processor p starts with a clock value T_p which is incremented accordingly: LC1) T_p is incremented $after$ each event at p. LC2) Upon receipt of a message with timestamp m, $T_p = max(m, T_p) + 1$.
* Next we need to formalize a stability condition. It should be noted that when assuming Byzantine failures it is impossible to gurantee with a deterministic protocol that agreement can be reached. Therefore, a stability condition can only be created for environments with synchronized clocks or environments where fail-stop failures can only occur but there are no message time bounds.
* If we assume that fail-stop failures can only occur with no message time bounds and if we use logical clocks to generate sequence numbers for requests, then we can show that two properties of communication channels hold: **FIFO)** Messages between pairs of processors are delivered in the order in which they were sent. **Failure Detection Assumption)** A processor p only detects that a processor q has become faulty after the last message sent by q is received by p. With these assumptions, we can create the **Logical Clock Stability Test Tolerating Fail-stop Failures:** Every client periodically makes requests (with some being no-op requests) to the state machine. A message r is stable once a state machine has received a message with a larger timestamp from *every* non-faulty client.
* Proof of the above: The **FIFO Assumption** of communication channels guarantees that clients will always send requests with monotonically increasing timestamps; therefores, once a request r is received by state machine sm from client c, no other r' will be received by sm from c with a lower timestamp than r. Once sm has received messages from all non-faulty clients with timestamps greater than r, then sm will not receive lower timestamps from those clients. For clients that can fail, the **Failure Detection Assumption** guarantees that those clients will stop sending messages at some time; therefore, sm can be assured that it will not receive a message with a timestamp lower than r from faulty clients after they have been detected to have failed.
* **Note that I'm skipping synchronized clocks case because I don't fully get it. Need to read more about what bounded clock synchronization actually looks like**
* A third method of generating unique identifiers for requests is to have the replicas agree on the identifier for a request as part of the agreement protocol that satisfies the Agreement requirement. An advantage to this approach is that all processors in the system no longer need to always be communicating (this was needed to gurantee message stability or clock synchronization in the previous two approaches).
* We can implement such a unique identifier system like so: When a replica sm_i first receives a request r, they propose a candidate unique identifier $cuid(sm_i, r)$. At this point the replica is said to have $seenrequestr$. A replica has $accepted$ request r once it knows $uid(r)$ - the final accepted unique identifier for r.
* To gurantee that identifiers generated in this manner are totally ordered, we must add two constraints: **UID1)** $cuid(sm_i, r) \leq uid(r)$. **UID2)** If a replica sees request r' after accepting request r, then $uid(r) < cuid(sm_i, r')$
* **Replica-Generated Identifiers Stability Test)** A request r that has been accepted by state machine sm_i is stable provided there is no request r' that has been seen by sm_i, not been accepted yet by sm_i and for which $cuid(sm_i, r') \leq uid(r)$.
* Proof of the above: Assume that a request r has been accepted by state machine sm_i, we must show that it is not possible for sm_i to accept a request with a lower uid than r. For the case of requests r' that have not yet been seen by sm_i, it follows from $UID2$ that $uid(r) < cuid(sm_i, r')$; therefore, it follows transitively from $UID1$

that $uid(r) < cuid(sm_i, r') \leq uid(r') \rightarrow uid(r) < uid(r')$. For requests that have been seen by sm_i but not yet accepted yet, it follows from our assumption that r is stable that $uid(r) < cuid(sm_i, r')$ and (using the previous logic) $uid(r) < uid(r')$.

* Note that $uid's$ generated by replica sets do not necessarily satisfy $O1$ and $O2$ (consider the case where a client sends a request r and then sends a message to another client which triggers request r', it is possible that $uid(r') < uid(r)$ even though $r < r'$). Thus, we must contraint clients so that once they make a request r to the replica set, they may not make any further communication until r has been accepted (this seems rather limiting...).
* To complete our Order implementation, we must guarantee that our protocol satisfies: 1) $UID1$ and $UID2$ 2) $r \neq r' \rightarrow uid(r) \neq uid(r')$ 3) Every request that is seen is eventually accepted.