Responsive Consistency in User-Centric Dynamic Network Environments

A prospectus submitted in partial fulfillment of the degree of Doctor of Philosophy

Preliminary Oral Examination for
Benjamin Bengfort

Advisor:
Dr. Peter J. Keleher

Committee Members:
Dr. Amol Deshpande
Dr. James A. Reggia
Dr. Dave Levin

Department of Computer Science
University of Maryland, College Park, MD 20742
December 6, 2016

Abstract

Data replication has largely become the central technique in creating durable, highly available, multi-user systems and as a result the fault-tolerant coordination of replicas for correctness and consistency has become an increasingly important research area. As these systems have grown in scale and been deployed in geographically distributed clouds, the research focus has shifted away from networked file systems toward data-centric, cloud-managed database systems. One motivation for this shift has been the need to support an increasing number of mobile devices and another to provide scalability in collaborative workloads. However, with the advent of the Internet of Things and the increased use of multiple devices and local collaboration, we expect that local, user-centric clouds of heterogenous devices and mobile, dynamic networks will become increasingly important.

We contend that current consistency models are highly dependent on the network environment and that existing protocols can fail to achieve progress or provide sufficient consistency for user applications in many environments outside of the data center. Weak or eventual consistency models rely on the fast propagation of updates and therefore consistency is dependent entirely on message latency. Strong, sequential consistency provided by consensus protocols require multiple messages passed through a leader and though it can be shown that such protocols are correct, leadership introduces a bottleneck and partitions can make progress impossible. In user-centric dynamic clouds, responsive consistency models must be developed to take advantage of the changing network environment and respond to user requirements.

We hypothesize that we can provide responsive consistency in heterogenous, mobile topologies through two mechanisms: the federation of multiple consistency models depending on the environment and the load balancing of leadership costs in a quorum through hierarchical consensus. Federated consistency allows heterogenous devices to select a variety of consistency polices that are integrated through a core consensus group. By allowing flexibility locally, devices can make progress through consistency models that take advantage of the current network environment, but still have global guarantees. Hierarchical consensus improves the core consensus group, making it highly available and scalable by balancing leadership to specific decision spaces based on access patterns. We propose that federated consistency and hierarchical consensus together will produce a responsive consistency model that provides strong enough correctness guarantees and high enough availability to implement an effective multi-user distributed file system.

Contents

List of Figures

1 Introduction

The rise of virtualized, on-demand computing resources and cloud computing has dramatically shifted the focus of research on consistency in distributed storage systems away from networked file systems toward geographically distributed database management systems, particularly key-value storage. These types of data systems benefit from low-latency, high-bandwidth connections where failure is common due to the magnitude of resources rather than inherent instability. In a low latency environment, "eventually consistent" (EC) [84] systems are said to be consistent enough for most workloads given some probabilistic bounding of staleness [14, 16, 13]. This has led consistency research [9, 11, 64, 6, 86] towards investigating stronger forms of eventual consistency, primarily building upon Dynamo [30] and Cassandra [50] as reference implementations. Alternatively, research into ensuring strong, sequential consistency (SC) [52] or linearizability (LIN) [38] in this environment has led to systems like Spanner [29] which uses atomic and GPS clocks to ensure "TrueTime" for sequential ordering or a small number of master nodes that implement consensus algorithms [54, 73] for locking or ordering [47].

The focus on data center consistency has led to a centralized approach to data storage, forcing a modality where devices must connect to the cloud for file replication even when files exist in the local area [32]. Although this allows systems to ignore annoyances such as local network configuration and decentralized protocols it does present unnecessary overhead in terms of cost and latency. More pernicious, perhaps, is that users must now buy-in and store their data with a single provider that could go out of business or be hacked, which has lead to research in replicating local data with multiple untrusted cloud stores [87, 34]. There are more devices than ever before connecting to storage applications, partially because users have multiple, heterogenous devices from wearables to workstations and partially because of the advent of the Internet of Things [67]. Cloud storage tends to be application specific, but with more devices and more users, *generalized distributed storage* is required for collaboration and sharing that is not siloed and can be easily accessed by a variety of new platforms. Therefore we see a strong motivation to turn attention back to user-oriented networks of mobile heterogenous devices that do not have the benefit of data center level connectivity.

We propose to explore consistency mechanisms in *user-centric dynamic clouds*: a multi-user tapestry of mobile, heterogenous devices connected via variable-latency and partition-prone networks that change over time. These types of clouds are interesting because they describe a much wider range of use cases than a data center environment, for example search and rescue clouds, mobile sensor clouds, or even transportation clouds. We believe that the unique challenges of this network environment require specialized consistency models in order to be effective. Because of the user-centric nature of this research, the requirement for collaboration and sharing, and in order to generalize distributed storage in this type of cloud we further propose the study of a file system as the primary data storage application. File systems must be highly available such that a user does not notice any delay due to coordination but must also have strong consistency such that any conflicts are presented to the user as soon as possible. These two requirements together provide a challenge for any single consistency model, particularly when the devices in the file system are mobile, can be turned off, and may have a variety of resource constraints.

Our core contention is that *consistency depends on the "environment"*. In a file system, the two primary symptoms of inconsistencies are version forks and stale reads. "Weak" (EC) systems depend on disseminating

new writes quickly in order to prevent forks and stale reads. A "strong" (Paxos [53] consensus) system without caching might be provably correct, but might fail to make progress as network conditions deteriorate. Furthermore, file caching is essential to providing usable performance for any file system. Caching (without expensive validations) leads to stale reads. Forks can happen even without caching unless locking is used, however locks are too expensive for data center environments [47, 29], much less user-centric dynamic clouds. Hence, a strong (Paxos) system does not necessarily provide strong consistency guarantees on individual application-level file accesses. Further, the extent to which the system diverges from SC increases with decreasing network quality.

In order to provide any consistency guarantees in a dynamic environment the consistency model must be *responsive*, providing flexibility when the network is unavailable or laggy and providing strong consistency guarantees when stable connections exist. Our approach focuses on two primary techniques to provide responsiveness: the *federation* of weak but available consistency mechanisms with strong but strict coordination and the use of *hierarchical* consensus to scale consensus groups beyond a handful of devices. "Federated consistency" allows devices to switch local consistency models depending on the use case or to adapt consistency protocols automatically as the network environment changes, optimizing for strong consistency or availability as required. Availability in partition prone networks is provided by eventual consistency, and federation allows users to flexibly adapt to changing environments. "Hierarchical consensus" increases the availability of a consensus group by creating decision localities of interest based on accesses to particular objects by a particular group of devices. Coordination with consensus is required for strong file system semantics and we propose that a hierarchical model makes consensus flexible enough for a file system. Both models are flexible locally, but can provide global system guarantees, guarantees we propose to explore in this research.

We hypothesize that the integration of multi-modal consistency with hierarchical centralized leadership via consensus will lead to higher availability than strong consistency models but provide stronger guarantees than eventually consistent systems. We will quantitatively demonstrate the efficacy of such a system through simulation and by implementing a file system, FlowFS, and testing it under real-life workloads. By comparing FlowFS to homogenous systems with similar sizes and topologies and measuring the number of inconsistencies and amount of latency in both simulation and a real world implementation we will show that responsive, flexible consistency protocols are more available and provide stronger guarantees than their homogenous counterparts in user-centric dynamic personal cloud topologies.

The rest of the proposal is organized as follows: first, we will present background and related work on consistency models, consensus protocols, and replication. We will then describe our preliminary work: network simulations of a variety of consistency protocols and an exploration of the effect of environment on consistency. We will then proceed to describe federated consistency in detail, work that we have also already investigated in simulation. Next, we will present our initial thoughts on hierarchical consensus and extensions on both federation and hierarchical consensus to improve performance such as automatic adaptation. We conclude with a proposed timeline and set of research goals towards a dissertation.

2 Background and Related Work

Our work is primarily grounded in data-centric consistency models. These models previously thought of as discrete levels, e.g. weak and strong, have more recently been viewed as a continuum [85, 1]. To provide background to consistency mechanisms in dynamic environments, we will start by defining a generalized log-based consistency model then extend it to our more file-system specific, primary metric: *forks*. We will then explore various consistency levels on the consistency continuum: weak, eventual, causal, sequential, and linearizable expressed both in terms of the generalized model as well as how they perform in a file system context. This grounding primarily provides background to our first responsive approach: federated consistency.

Federated consistency is a form of hybrid consistency where several consistency protocols are integrated in a single system. Because our preliminary work has focused on federation, we will present a detailed review of related work on hybridization and consistency rationing. We hypothesize that by integrating both eventual and sequential consistency such that there is a strong central core, similar to the architecture presented by Oceanstore [48], federated consistency will be able to provide stronger guarantees and centralized conflict resolution than other weaker consistency models, while providing flexibility and availability in variable latency environments.

The strong central core must provide at least sequential consistency, a consistency level that is generally coordinated via a consensus protocol. Therefore, in the second part of the background, we will present consensus protocols, starting by a review of Paxos [53, 54] and its various flavors. We have selected Raft [73] as our consensus protocol of choice, and therefore describe its operation in particular detail. Our review of consensus is intended to present the motivation and discussion of hierarchical consensus – an extension of Raft that load balances the leader across decision spaces (groups of related objects) based on access patterns. Finally, in the last section of the background, we will review topics in replication that are critical to our work and present related systems.

2.1 Consistency

Our consistency model is a *data-centric* model, as opposed to a *client-centric* model [15]. Client-centric models view the system as a black box and consistency is described as guarantees made to processes or applications interacting with the system such as "read your writes" or "write follows read" [84]. Data-centric consistency on the other hand is concerned about the ordering of operations applied to a replica and generally considers the problem of how those operations relate to each other in a per-replica, append-only log. This generalization is different than a file-system specific view of consistency, but gives a method for comparing different consistency protocols by increasing strictness of ordering and staleness.

We can generalize data-centric consistency as follows: each replica is a state machine that applies commands in response to client requests or messages from other replicas. Each command is appended to a log that records the time ordered sequence of commands such that an entry at the end of the log happened before the previous entry. Replicas are locally consistent if they are in a similar state expressed by the relative condition of their logs (note that this generalization will translate well to a discussion of consensus in the next section); they are globally consistent if they are in a similar state to some abstract global ordering that

meets one or more criteria. In either case, consistency guarantees can be described along two dimensions: staleness and ordering as follows:

1. *Staleness* refers to how far behind the latest global state a local log is and can be either expressed by the visibility latency of distributing a command to all replicas or simply by how many updates the log is behind by.

2. *Ordering* refers to how closely individual logs adhere to an abstract chronological ordering of commands. A strict ordering requires all logs or some prefix of the log to be identical, whereas weaker ordering allows some divergence in the order updates are applied to the log.

Most data-centric consistency models refer to the strictness of ordering guarantees and the method by which updates are applied to the state of the replica [79]. This is primarily because enforcing ordering strictness leads to increased staleness in two ways: increased coordination introduces delays between the issuance of a command and its local application and through dependencies that require dependent commands to be applied before the command in question. While staleness is easy to quantify in terms of time (t-visibility) and versions (k-staleness) [14], ordering is very difficult to quantify. Therefore in order to generalize to consistency models beyond eventual consistency, our model considers the specific symptoms of ordering inconsistency in a file system context: *forks* and *stale reads*.

2.1.1 File System Consistency

We define a file system as a hierarchical namespace of object *versions* where a version represents the immutable state of a particular object at a specific time [37]. The *view* of a file system is the set of locally cached versions, usually the latest version, for each object. Accesses to specific objects, e.g. read and write, update the local view of the file system and must be replicated in order for the file system to be consistent across replicas.

We propose a file-system specific consistency model that extends the generalized log ordering consistency method. In this model, commands are file accesses to a single object and each replica maintains a single, ordered log of accesses. Write accesses update the state of the file system (or view) by creating new versions of objects; read accesses update the state by determining the most recent version of the object either via the local cache or via a remote access. A *stale read* is defined as a read to the local cache that returns a version of the object that is behind the latest global version. The ordering of reads and writes in the log determines the chronological sequence of all accesses to the file system and therefore defines the version history.

Each *version* is a piece of meta data that describes an object state at a particular time and is identified by a monotonically increasing, conflict-free version number [76]. Versions can contain dependency information, and in particular one dependency is required for all versions: the version of the parent upon whose access created the resulting version. An object's version history is therefore a tree, though the desired history is a single sequence or chain of unbroken updates to the object. Inconsistency in a file system is therefore a divergence in the linear history of the set of objects in the namespace and can be measured in terms of *forks*.

Fork A fork occurs when two replicas concurrently write a new version to the same parent object as shown

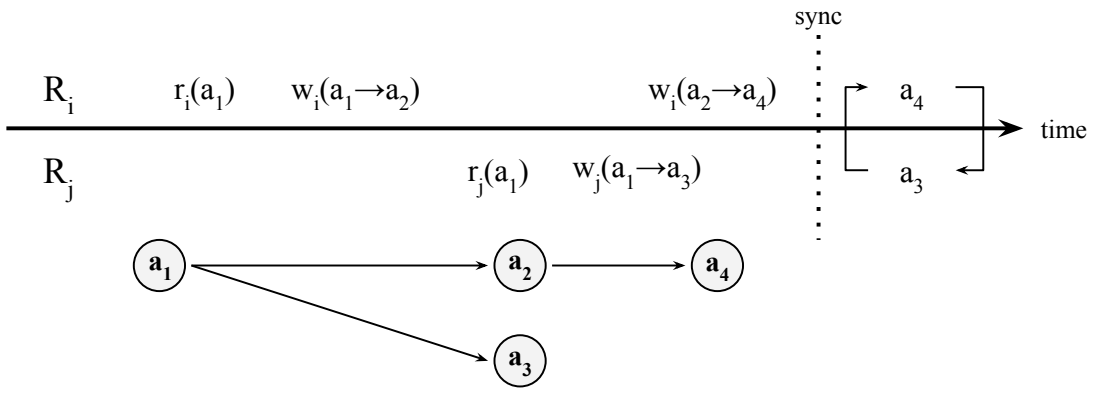

Figure 1: An *object fork* ocurs when a single version is the parent (last read version) of multiple new versions. Replicas R_i and R_j each read and then write object a in isolation. R_j's read was stale and causes an object fork, a_4 vs a_3, discoverable only after synchronization.'

> in Figure 1. Forks introduce inconsistency because there are now two potential orderings of updates to the log, but forks are primarily the symptom of staleness; e.g. the second writer wrote to a stale version of the object.

There are two primary causes of forks: concurrent accesses by multiple replicas (a true conflict between multiple writers) and stale reads. The former is distinguished from the latter only if the possibility of synchronization (a process to bring replica logs to the same state) could have occurred. From the point of view of the system, they are identical causes. Forks can branch to arbitrary lengths as replicas continue to write to their latest local copies, however when synchronization occurs a decision as to which ordering of writes is correct must occur. Note that conflict in this consistency model refers to the concurrent access of the same or dependent objects. In fact, it is possible that the concurrent access to two independent objects causes two different, valid orderings – however because it is difficult to implicitly define objects that are independent from each other, consistency definitions do not take this conflict into account.

2.1.2 Discrete Consistency Levels

In this section we will identify several discrete consistency levels in order of increasing strictness of the ordering of their logs as defined by the generalized log consistency model presented by Bermbach and Kuhlenkamp [15]. The relationship between these consistency models indicates that there is a continuum of consistency based on the strictness of ordering. We will also note, where possible, the effect of the consistency model in terms of file-system specific consistency and forks.

Weak consistency (WC) is the least strict consistency model. A weakly consistent system makes no guarantees whatsoever about the relationship of local to remote writes and whether or not any given update will become visible [84]. Weak consistency is often described as "replicas might get lucky and become consistent" and in fact a weakly consistent implementation may not have a synchronization protocol whatsoever [15]. For this reason, we do not consider weak consistency in general except to identify it as a baseline.

Eventual consistency (EC) is primarily concerned with the final state of all logs in the system given some

period of quiescence that allows the system to converge. In this case, all replicas, no matter their log ordering, should have identical final versions for all objects in the namespace. This suggests that eventual consistency requires some *anti-entropy* mechanism to propagate writes and a policy to handle convergence [81]. Eventual consistency is very popular for NoSQL databases and hosted distributed storage services [30, 50] because it allows an optimistic approach to consistency: conflicts are rare in most systems, if something does go wrong, conflict resolution is passed to the application layer. In practice, most applications can handle some inconsistency and moreover the small inconsistency windows due to low latencies in cloud data centers make such conflicts rare and short-lived enough to be worth the risk [14].

Eventual consistency implemented by a *last writer wins* policy simply accepts all writes so long as they are more recent than the latest local versions. Eventual reads and writes are always performed on local caches and are therefore highly available since they return a response to accesses immediately. Eventual consistency does allow forks to occur and moreover allows individual replica logs to have wholly different orderings so long as the last version for each object is in the same given no accesses have occurred for a long enough period of time. As a result, the latest version of an object may alternate between writes to competing forks (a fairly weak semantic) and in this case, it is up to the application to detect the inconsistency. However, EC logs do have one important property - for each object, every write in the log is ordered in a monotonically increasing fashion.

Causal Consistency (CC) ensures that all writes which have causal relationships have those dependencies satisfied (inserted into the log) before the write can become visible [3]. Therefore, even though a write might have been propagated to another replica server it cannot be read until all of its dependencies have also been propagated. Causal consistency can increase staleness particularly when implicit or potential causality creates large dependency graphs that must be resolved before writes can be applied [63]. This can be managed by allowing the application to explicitly specify the dependencies for each write [10]. Causal consistency is often referred to as the "strongest form of consistency for highly available systems", however CC does not require replica convergence [36] though with a few modifications including a convergence mechanism, it can easily be bolted on to an eventual system [12].

Eventual and causal consistency are both weak forms of consistency that are designed for high availability; that is they can respond to requests immediately from a local cache without coordination. Conflict detection and resolution are therefore critical processes asynchronous with the accesses that cause the conflict. These types of consistency are well suited for network environments that are partitioned or such that effective coordination is not available at the time of the access; e.g. for a user working on an airplane with no connectivity. The following strong consistency models, on the other hand, require coordination at the time of the access, detecting and eliminating conflict synchronously.

Sequential consistency (SC) is a strong consistency model that requires that all replicas have the exact same ordering of their logs, such that all writes by all clients are appended in the same exact order [8]. Sequential consistency is not strict in that it does not make guarantees about staleness (or the ordering of reads) but does require that all writes become visible in the same order [15]. Sequentially consistency is typically implemented with consensus algorithms such as Paxos [56] or Raft [73] that coordinate logs by defining a transitive, global ordering for all conflicts. Alternatively, sequential consistency and can be implemented with warranties – time based assertions about a group of objects that must be met on all replicas before the assertions expired [62].

Linearizability (LIN) is the strongest form of consistency; not only must all write operations occur in sequence, but all operations including reads must be ordered chronologically [38]. A consensus algorithm alone cannot implement linearizability and instead some distributed locking mechanism is required. For example a consensus algorithm can be adapted to instead of making decisions about the total ordering of conflicting writes, granting or releasing locks from requestors, however this opens up the potential for deadlock and extremely poor performance, defeating the purposes of replication in the first place! Data center environments that don't have to deal with issues of clock skew by using super precise atomic and GPS clocks can use precise time measurements to enable a distributed two phase commit protocol [29], however every replica is required to have such a time piece, which is not practical for heterogenous topologies.

2.1.3 Consistency Rationing and Hybridization

Federated consistency attempts to integrate multiple consistency levels as described in the previous section into a hybrid consistency model such that there is a separation between the local availability and consistency trade-off and global consistency guarantees. For example, the hybridization of an eventual consistency cloud of devices with a strong sequential core allows eventual nodes to make progress, while minimizing the effect of convergence delay on consistency. In this section, we will review work related to this style hybridization.

One of the earliest attempts to hybridize weak and strong consistency was a model for parallel programming on shared memory systems by Agrawal et al [2]. This model allowed programmers to relax strong consistency in certain contexts with causal memory or pipelined random access in order to improve parallel performance of applications. Per-operation consistency was extended to distributed storage by the RedBlue consistency model of Li et al [60]. Here, replication operations are broken down into small, commutative suboperations that are classified as red (must be executed in the same order on all replicas) or blue (execution order can vary from site to site), so long as the dependencies of each suboperation are maintained. The consistency model is therefore global, specified by the red/blue ordering and can be adapted by redefining the ratio of red to blue operations, e.g. all blue operations is an eventually consistent system and all red is sequential.

The next level above per-operation consistency hybridization is called *consistency rationing* wherein individual objects or groups of objects have different consistency levels applied to them to create a global quality of service guarantee. Kraska et al. [46] initially proposed consistency rationing be on a per-transaction basis by classifying objects in three tiers: eventual, adaptable, and linearizable. Objects in the first and last groups were automatically assigned transaction semantics that maintained that level of consistency; however objects assigned the adaptable categorization had their consistency policies switched at runtime based on a cost function that either minimized time or write costs depending on user preference. This allowed consistency in the adaptable tier to be flexible and responsive to usage.

Chihoub et al. extended the idea of consistency rationing and proposed limiting the number of stale reads or the automatic minimization of some consistency cost metric by using reporting and consistency levels already established in existing databases [26, 27]. Here multiple consistency levels are being utilized, but only one consistency model is employed at any given time for all objects, relaxing or strengthening depending on observed costs. By utilizing all possible consistency semantics in the database, this model allows a greater spectrum of consistency guarantees that adapt at runtime.

Al-Ekram and Holt [4] propose a middleware based scheme to allow multiple consistency models in a single distributed storage system. They identify a similar range of consistency models, but use a middleware layer to forward client requests to an available replica that maintains consistency at the lowest required criteria by the client. However, although their work can be extended to deploying several consistency models in one system, they still expect a homogenous consistency model that can be swapped out on demand as client requirements change. Additionally their view of the ordering of updates of a system is from one versioned state to another and they apply their consistency reasoning to the divergence of a local replica's state version and the global version. Similar to SUNDR, proposed by Li et al. [61], an inconsistency is a fork in the global ordering of reads and writes (a "history fork"). Our consistency model instead considers object forks, a more granular level that allows concurrent access to different objects without conflict while still ensuring that no history forks can happen.

2.2 Consensus

Consensus algorithms are generalized as replicated state machines where a quorum of replicas must coordinate to decide on the application of a command that will change the local state of the replica [78]. By keeping a log of all applied commands, consensus is fault-tolerant because an offline node that rejoins the quorum can replay the commands to return to the same state as the other replicas. Moreover, so long as a majority of nodes are online, the quorum can be said to be available, in that it will respond to requests that access the state. Because accesses can be seen as commands modifying the visible state of the object namespace, and because the command log is identical to the consistency logs described in the last section, we can say that consensus algorithms can be used to provide strong consistency at the cost of multiple coordination messages per access.

Most consensus algorithms in the literature are variations of the Paxos algorithm [53, 54], and more specifically the Multi-Paxos implementation [21, 66, 23]. Paxos proposes three phases to safely apply a command to the state machine: *prepare*, *propose*, and *accept*. All phases are two stages, the first a vote and the second a broadcast of the results of the vote. In the prepare phase, a replica attempts to establish the master replica for a specific command by broadcasting a ballot number and getting majority agreement that the ballot (the entry at the log at that position) is owned by the master. The second phase broadcasts the value and receives a majority vote if that state can be successfully applied. The third phase accepts (commits) the result to the state machine. The primary variation, Multi-Paxos, bundles multiple requests by reserving ballot numbers ahead of time to eliminate the *prepare* phase (which is why most descriptions of Paxos generally identify only two phases). One way we can think of this is as the election of a primary leader in the quorum.

Because of the multiple round-trips necessary to achieve consensus, many variations of Paxos have been proposed to improve performance. For example S-Paxos [18] attempts to distribute the workload of the leader, Fast Paxos [56] allows multiple leaders per round, Flexible Paxos [39] allows for multiple quorum intersections, and Egalitarian Paxos [68, 69] allow fast and slow track voting. These variations tend to be optimistic in that conflict occurs rarely and that it can be detected by the consensus algorithm, allowing for repair if necessary.

2.2.1 Raft

Paxos has been shown to be very difficult to correctly implement and verify [23] and although attempts have been made to highlight the practicality of a subset of the Paxos guarantees [66] it has been shown that Paxos is not easily understandable. To that end, we have used the Raft consensus protocol [73, 40] as our primary consensus implementation in our preliminary investigations. The protocol is described briefly as follows.

Every Raft node can be in one of three states: *follower*, *candidate*, and *leader* and are initialized in the follower state. The system has two primary timing parameters: the election timeout and the heartbeat interval. When in follower and candidate mode, a timeout is randomly selected from the election timeout range, and if the timeout occurs the node will become a candidate and start an election to become leader. If a majority of nodes vote yes to that candidate then the node switches to the leader state and begins sending `AppendEntries` messages at the rate of one at least every heartbeat interval. If a follower or a candidate receives a valid `AppendEntries` message, it resets its election timeout by selecting a new random timeout from the election timeout range. The relationship between the heartbeat interval and the election timeout must be such that at least two heartbeats can arrive before the node switches to candidacy. The random selection of a timeout prevents thrashing if all nodes are started at the same time.

Once elected leader, all accesses are forwarded to that node. Leaders maintain a term identification, a monotonically increasing global number. If a message comes in with a remote term higher than the local, then that node must switch to being a follower and update their local term. Accesses and their terms are sent to all followers in `AppendEntries` messages. Followers compare their terms to the access term, and if a majority of followers accept the access, then the leader sends a commit message in the following `AppendEntries` RPC message.

2.3 Replication

Consistency and consensus are generally separate issues from the actual replication of both metadata and data in a distributed storage system. In fact, consistency only requires the replication of *metadata* because it is the metadata that defines the view or state of the system [24]. In fact, given that data can be replicated as immutable chunks or blocks identified by a hash function, the problem of replicating data is one of service availability [83] or data locality [49] not one of locality. Therefore for the purposes of this research, we focus on the consistent replication of metadata.

We have explored two primary methods of replication in a distributed file system: gossip and broadcast protocols. As a rule, limiting the number of messages sent during replication is important as the number of messages is a good metric for resource usage and constraints such as available bandwidth or message processing capability. Gossip protocols are generally used as anti-entropy to disseminate information across the network in an epidemic fashion. Because gossip protocols use peer-to-peer connections, there are only as many messages per gossip interval as there are pairwise associations. Broadcast replication, on the other hand, can quickly overwhelm a system if all nodes are broadcasting a message to all other nodes, but is used well in centralized systems such as the Raft protocol where only a leader broadcasts and followers respond to the leader.

Gossip protocols (often called rumor spreading) are a form of epidemic information spreading that do not require central coordination [45, 43]. On a routine interval, specified by the `anti-entropy delay` timing parameter, a replica will randomly select one of the other replicas in the system and exchange information. Because all nodes are randomly selecting another node at every interval, information travels quickly through the network in an exponential fashion and provide high fault-tolerance and even automatic stabilization. Generally speaking there are two types of gossip: *push* and *pull*. Push methods simply forward all locally updated information to the remote node on the anti-entropy interval. Pull methods require two phases: a request for information after a certain period and the response from the remote node.

Several replication protocols have directly influenced or inspired our work. Bayou [81, 80] is an eventually consistent system that implements anti-entropy as its replication mechanism and conflict resolution by some primary or trusted centralized server. SUNDR [61] is a secure file system that integrates local and cloud storage and detects inconsistencies by identifying forks, our primary inconsistency metric. The Ori File System [65] replicates file history and allows the merge and grafting of histories similar to Git trees. As a result branches and forks are also a big part of the Ori methodology as are complex data structures for grafting and dealing with conflict. Finally, CalvinFS [82] and GlobalFS [75] both implement strong consistency in file systems across a wide area using consensus algorithms (namely Paxos) but take advantage of data center performance and small quorum sizes.

3 Preliminary Work

We have begun our investigation into the relationship between network environment and consistency by building a discrete event simulator (DES) that allows us to easily characterize networks and consistency protocols. The simulator was implemented in Python using the SimPy simulation package, allowing for rapid prototyping and implementation of a variety of processes and flexibility to put together a series of experimental simulations. For our preliminary work, we have utilized the simulator to investigate *federated consistency* and have made progress by obtaining experimental results. In this section, we will describe the simulator and the investigation that led to the core proposal of federated consistency and hierarchical consensus. In the proposed work section, we will explore federated consistency in greater detail.

The simulator has two primary components: workload generators and device processes. Workload generators act as clients (users) that issue read and write accesses to named objects and are generally associated with a single device process such that the access is associated with a local replica. Device processes represent, for the most part, replica servers that are connected to each other through a specific topology. Devices respond to accesses, generate replication messages, and respond to messages from other devices. The input to a simulator therefore is a topology that defines the devices, their specific properties and connections as well as a workload trace or set of parameters to generate random, realistic workloads. Experiments are conducted by running multiple simulations in parallel with different combinations of the above parameters.

The primary input to a simulation is the topology of devices and their connections. Topologies allow us to vary connectivity and to simulate a range of network environments, exploring the effect of a variety of consistency protocols in user-centric dynamic clouds. Our standard model of user-centric dynamic clouds is described by multiple wide-area locations with variable network connectivity between and within each

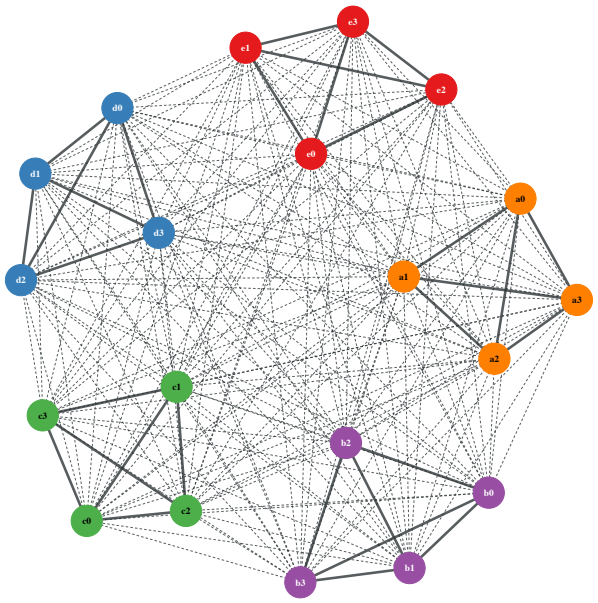

Figure 2: Proposed Network Topology.

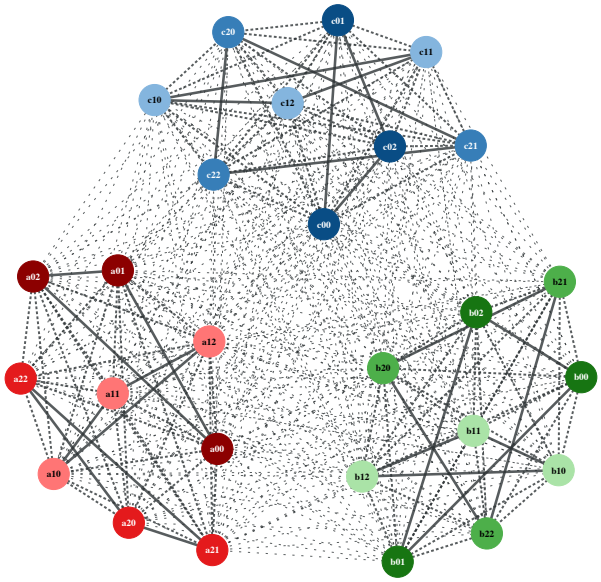

Figure 3: Proposed Network Hierarchy.

location. We propose to investigate a fully connected topology of replica devices each assigned a geographic region as shown in Figure 2. Within each region, replica nodes enjoy stable, low-latency connections with their neighbors. However, across regions the latency is higher and the connections variable, meaning that out of order messages and delays are more common across the wide area than in the local area. This topology can be further generalized to tiers of locations as shown in Figure 3, such that groups of nodes are grouped hierarchically in tiers of increasing latency and variability.

User-centric, dynamic networks are susceptible to routine partitions, events that cause replication messages to be delayed or dropped. We define two types of communication failure events in our topology: node failure and network partitions. *Node failure* occurs when a single node is shut off or stops responding to messages. *Network partitions* occur when it is not possible for messages to be sent or received from a single geographic region. In both cases, two conditions must be dealt with by the replication protocol in order to satisfy correctness criteria: resending lost messages and bringing nodes that are behind up to date. Introducing these events in our simulation is future work, but is necessary to a complete investigation of consistency in variable network environments.

Because most research on gossip protocols and consensus algorithms specify small topologies of a size that provide a minimum level of fault tolerance, our initial questions related to the scalability of consistency protocols to topologies with increasing numbers of nodes. In order to explore these questions, we designed an experiment with 46 independent simulations: 23 differently sized topologies from 5 to 225 devices and two homogenous consistency protocols. This example serves to demonstrate the methodology behind our simulations as well as highlight a number of the tunable knobs that allow us to explore our proposal in greater detail.

Each simulation implements either eventual consistency via bilateral gossip and a latest-writer wins policy or sequential consistency via Raft consensus. The simulation is specified by a topology with a variable number

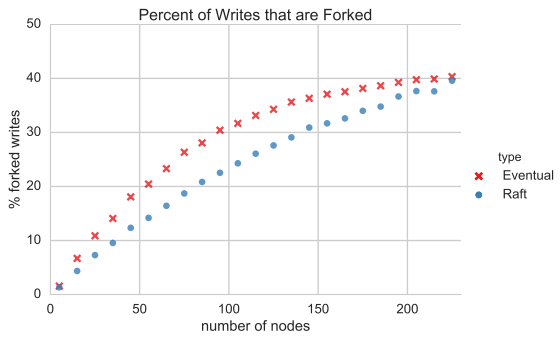

Figure 4: Increasing forks in homogenous systems as topology size increases.

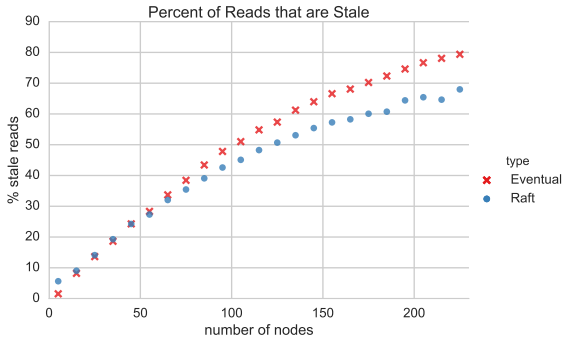

Figure 5: Increasing stale reads in homogenous systems as topology size increases.

of devices evenly divided between five locations. Devices in the same location are said to be in the local area; local area latencies were normally distributed with mean latency, $\lambda_\mu = 30$ms and standard deviation of latency, $\lambda_\sigma = 5$ms. Across locations, devices were said to be communicating in the wide area, such that latencies are higher and more variable: $\lambda_\mu = 300$ms and $\lambda_\sigma = 50$ms.

Both EC and SC implementations rely on timing parameters for messaging such as the gossip interval and election timeouts. In order to define a relationship between consistency models, these timing parameters are defined by network latency. We first computed a conservative "tick" parameter given as $T = 10\lambda_\mu$ based on the wide area latency mean. The "tick" can then be used to specify all other timing parameters. The anti-entropy interval for Eventual consistency is given as $\frac{T}{4} = 750$ms. For Raft, the heartbeat interval is $\frac{T}{2} = 1500$ms and the election timeout is a uniform random selection in the range $U(T, 2T) = U(3000, 6000)$. These conservative timing parameters ensure that it is rare that messages arrive out of order, even in highly variable connections.

The simulation workload was defined as a static trace of accesses such that all simulations received the same exact accesses. Each device implemented a workload of read and write accesses on a local namespace of 15 objects. Conflicts were introduced to the simulation by overlapping a subset of the local namespace of each device with the other device namespaces, specified by a probability of conflict, P_c. In this simulation, $P_c = 0.3$ meaning that 30% of the objects accessed on each device would also be accessed on another device. Accesses were issued at intervals normally distributed with the access interval mean, $A_\mu = 3000$ms and the access interval standard deviation, $A_\sigma = 380$ms such that accesses were roughly related to the timing parameters.

As the number of nodes increases, the number of conflicts also increases – but not the likelihood of a conflict. Our system primarily measures inconsistencies as *forks* and *stale reads*. As shown in Figure 4, the number of conflicts that create forks in Raft is lower than in eventual. This is because eventual consistency relies on fast propagation of updates to minimize forks, which becomes increasingly slow for larger topologies; however Raft does eventually catch up to eventual when the number of nodes causing conflict is too much even for broadcast style distribution. In terms of stale reads, eventual consistency initially outperforms Raft for smaller topologies as shown in Figure 5. There is a cross-over point where anti-entropy is faster at distribution then the broadcast mechanism takes over; this point is defined by the relationship of the anti-entropy interval with the heartbeat interval.

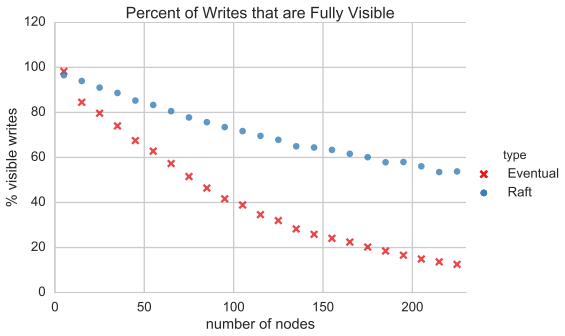

Figure 6: Percent of fully visible writes in homogenous eventual and sequential consistency systems.

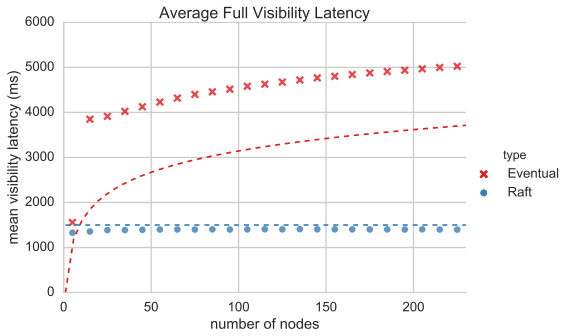

Figure 7: Latency of full visibility in homogenous eventual and sequential consistency systems.

Both of types of inconsistencies are directly related to how updates become fully visible in either system. In fact as the number of nodes increases, both systems start to degrade in the percent of writes that become fully visible, though EC far more than raft as shown in Figure 6. For Eventual, this is because the time to visibility is related to the number of pairwise anti-entropy sessions required to propagate a write to all nodes as shown in Figure 7, where the red line represents perfect convergence (not likely given uniform random neighbor selection). If an update becomes outdated before it can become fully replicated, it gets "stomped", meaning that the later update is propagated over the earlier one. Raft on the other hand only stops writes from becoming fully visible if they are forks, dropping them and rejecting the access to the client who must retry.

These figures seem to suggest that Raft is a much better replication protocol and that there doesn't seem to be any reason to use eventual consistency. However, eventual consistency allows more inconsistencies to occur in order to provide more flexibility and availability. Raft suffers from write and commit latencies and a bottleneck as all communication must pass through the leader. EC always accesses locally and therefore has no write latency. Raft also suffers from partitions and node failure, and cannot make progress if no leader exists; EC will always respond no matter the current network condition. Moreover, in these simulations, we have artificially constrained the anti-entropy rate by relating it to the "tick" parameter. As such, both protocols send approximately the same number of messages. However, without message limits, exponential anti-entropy can converge more quickly than Raft broadcast propagation, which must be sequential.

We believe that these initial investigations show a critical opportunity: that we can blend the high availability of an eventually consistent system and leverage the best parts of anti-entropy and eventual consistency in network environments where messages cannot get through along with the stability and increased visibility of a strongly consistent core, similar to the central core and flexible outer shell proposed by Gray and Oceanstore [35, 48]: we call this *federated consistency*. Furthermore, if we can find a way to allocate decision space to a smaller number of nodes, we should be able to scale Raft to greater number of nodes without as steep a curve: we investigate this in *hierarchical consensus*. Finally, as suggested by the T "tick" parameter (though not explicitly covered), the timing measures and performance of consistency protocols are related to the network environment, which is dynamic. Potentially we can improve performance by monitoring the environment and adapting timing parameters to minimize the number of inconsistencies: investigated in *Adaptive Consistency*.

4 Proposed Work

We propose two mechanisms that together will provide responsive, flexible consistency in user-centric dynamic clouds: federated consistency and hierarchical consensus. We hypothesize that the integration of federated consistency models and scalable consensus through these two methodologies will lead to a system that is quantitatively more available (experiences lower aggregate read, write, commit, and visibility latencies) than a homogenous implementation of a sequentially consistent system implemented by Raft. Further we hypothesize that such a system will have stronger consistency guarantees (fewer forks and stale reads) than a homogenous implementation of an eventually consistent file system implemented with bilateral gossip anti-entropy.

Federated consistency and hierarchical consensus provide different opportunities to scale and handle variable environments, each enhancing the other. Federated consistency allows us to create a flexible, heterogenous distributed system, allowing different replica servers to maintain different consistency levels based on need, but ensuring that global consistency guarantees are met. Hierarchical consensus extends the Raft consensus protocol to use consensus partitioning as a means of achieving scale and high availability among replicated logs, without sacrificing sequential consistency. We hope to further stretch our proposal to include the investigation of real time optimization and adaptation in response to changing environments. The stretch goal, Adaptive Consistency proposes to utilize both heuristic methods of steering configuration change as well as active, machine learning mechanisms for online optimization.

We propose to confirm our hypotheses in two ways: through the simulation of a variety of network environments and conditions and through real world experimentation on a fully implemented system. The simulation, as discussed in the preliminary work section will allow us to easily explore a variety of environments at a low cost of time and effort. We propose to extend our preliminary work to account for outages and to provide visualizations of consistency protocols. The full implementation will be a distributed file system called *FlowFS* whose architecture is given in Figure 16. By exploring real world workloads in real network environments we will be able to make constructive claims about how consistency needs to evolve as user-centric dynamic clouds become increasingly important.

In this section, we present details of our proposed work. We start with a brief description of federated consistency, preliminary work that has already been investigated through use of our simulation. We will then propose a methodology towards hierarchical consensus. Finally, we will propose stretch goals for integrating federated consistency and hierarchical consensus to become adaptive, automatically responding to changes in the environment. We will conclude with a timeline and goals for paper publications on these topics.

4.1 Federated Consistency

Heterogenous topologies with multiple users mean a variety of requirements for both availability and correctness. For example, consider a user working on a non-critical document on a train with limited cellular connectivity; the requirement here is probably high availability and progress rather than strong replication. On the other hand, during collaborative document editing, users might want to ensure strong sequential consistency and are willing to accept minimal delays such that the document is always in a consistent state. However, it is not possible to maintain a single, global consistency level that meets both of these require-

ments, leading us to the question: can consistency be adapted or tuned at runtime in such a way as to provide more availability in low connectivity situations or for low conflict objects and strong consistency and correctness in optimal network conditions or for critical or high priority workloads?

Our preliminary solution is a hybridization of discrete consistency models as discussed earlier, which we have already begun exploring via the simulation as introduced in the preliminary work. The federated consistency model allows individual replicas to select their own local consistency policies and engage in replication according to the mechanism specified by the policy. Each replica maintains its own local state which is modified in response to local accesses as well as the receipt of messages from remote replicas. Each replica sends messages to other nodes in order to propagate the latest writes as well as to perform housekeeping. Therefore every replica can be seen as an event handler that responds to local access events as well as remote messages and generates more events (sent messages) in return. Simply put, so long as every federated replica has an event handler for all types of RPC messages, federation only has to be defined at the *consistency boundaries*, that is when replicas of one consistency type send messages to that of another.

Given the consistency models discussed in the previous section, we will omit weak consistency as being too simplistic and linearizability as being too performance restrictive. Instead we will focus on the federation of eventual consistency, implemented with latest-writer wins gossip based anti-entropy, and sequential consistency implemented with the Raft consensus algorithm.

A federated consistency protocol finds a middle ground in the trade-off between performance and consistency, particularly between an eventually consistent system implemented via gossip-based anti-entropy [45] and a sequential consistency model implemented by the Raft consensus protocol [73]. By exploring these two extremes in the consistency spectrum we have observed in simulation that the overall number of inconsistencies in the system is reduced from the homogenous eventual system and that the access latency is decreased from the homogenous sequential system. Moreover, because the global consistency of the system is topology-dependent, it can be said to have flexible or dynamic consistency. We have found that large systems with variable latency in different geographic regions can perform well by allowing most nodes to operate in an optimistic fashion, but maintain a strong central quorum to reduce the amount of global conflict.

4.1.1 Gossip and Anti-Entropy

Eventual consistency allows replicas to operate in a highly available fashion at the risk of encountering some conflict (in the form of forks) that must be resolved in the future. We take the common approach of using periodic *anti-entropy* sessions to converge replicas (e.g. reducing entropy, the divergence between the states of individual replicas) via a gossip protocol [45]. Each replica periodically selects a random partner and sends a `Gossip` message containing the latest version of all objects in the replica's local log. On receipt of the `Gossip` message, the remote replica will compare the RPC object versions with those in its local log. If the RPC versions are later, it will append the later versions of the object to the log (*last-writer wins*). However if the remote object version is later it will send that version back to the originating node in a `GossipResponse` message. As a result, our anti-entropy implementation is *bilateral*.

Eventual consistency replicas read and write locally, resulting in essentially zero read and write latency. Forks are caused by the *visibility latency*, i.e., the amount of time needed to propagate a write to the rest of

the system. The visibility latency is dependent on the number of nodes in the topology and the anti-entropy interval, but because of the bilateral nature of our gossip protocol, has a logarithmic relationship to the number of nodes rather than a linear one.

The relationship between the anti-entropy delay and the size of the network relative to the mean time between accesses potentially means that eventual consistency would not cause forks. Said another way, nothing beats eventual consistency for availability and correctness given a fast enough network. Unfortunately, real-world networks are not ideal and conflicts in an eventual consistent system can lead to behaviors that are especially challenging in file systems. If a partition occurs and an object is forked, both sides of the eventually consistent system will continue allowing the branch to be extended, resulting in a difficult merge. Updates can also be "stomped", meaning that if they are not fully replicated they will not continue to be propagated if a later version exists. In order to provide stronger file system semantics, we propose to implement a core strong consistency consensus group that will improve the overall correctness of the system.

4.1.2 Strong Core Consensus Group

We implement sequentially consistent replicated logs via the Raft consensus algorithm [73]. However, consensus alone does not ensure that sequentially consistent file system accesses are guaranteed and a number of policy decisions about how Raft followers read, write and interact with the leader must be discussed. In order to present the possibilities for policies in our system, we must understand the background of the Raft protocol's implementation of sequential consistency.

The Raft leader has the primary responsibility of coordinating all other Raft replicas and therefore is also primarily responsible for ensuring a sequentially consistent system that maintains file system consistency invariants. A write access that originates at a follower must be sent as a `RemoteWrite` to the leader. The leader accepts writes in the order that they are received, and if the leader detects a fork – that is that a write has a parent version who already has a child version in the log – Raft will simply reject (drop) the write. In order to minimize the number of messages that Raft sends, Raft will aggregate all writes into the next `AppendEntries` message and send them together.

The aggregation of writes and latency of coordination messages means that there is a delay between leadership decisions and follower actions. As a result, write accesses are dependent on the response time of the leader but could also be dropped, requiring the accessor to retry or to handle the conflict somehow. In order to make progress, followers must decide how to read a parent version in order to make a write. There are several options, each providing a varying level of safety, minimizing the risk of a stale read:

1. *READ COMMITTED* Raft replicas will only read the latest committed version of an object, guaranteeing that the write will not be rolled back in the case of an outage. However, this read mode introduces the potential for a lot of staleness and therefore forks.
2. *READ LATEST* Raft replicas will read the latest version of the object in their log, even if it hasn't been committed. Moreover, replicas will read their own local writes rather than waiting for an `AppendEntries` to return their write.
3. *REMOTE READ* Rather than read locally, simply request the latest version from the leader. This

introduces the potential for additional latency, but may be faster if the expected message latency is less than the heartbeat interval.

Each of these options has critical implications for the likelihood of stale reads and forks in the system. Replicas would choose read committed if the network was highly partition prone and messages from the leader were unstable and prone to being rolled back. Remote read servers replicas well when the average message latency is far lower than the heartbeat interval, though this could be improved by making the heartbeat interval similar to the network latency. For this reason, we propose *read latest* as the most likely scenario for a file system implementing sequential consistency with Raft.

4.1.3 Integration

A key requirement of federated consistency is the opportunity to create heterogeneous systems with no performance cost, e.g. a homogenous eventual cloud and a homogenous Raft cloud will continue to perform equivalently whether or not they participate in a federated cloud. However, we posit that an eventual cloud should benefit in lowered data staleness and in fork frequency from being connected to a strong, central consensus group. Similarly, Raft nodes should be able to use anti-entropy mechanisms to replicate data and continue writing even if the leader is unavailable and no consensus can be reached to elect a leader. The question is therefore how to integrate the eventual consistency via gossip and sequential consistency via consensus in a non-invasive way.

The central problem is that eventual and Raft clouds choose the "winner" of a fork in exactly opposite ways. Eventual clouds choose the last of a set of conflicting writes through a latest-writer wins policy, whereas Raft clouds effectively choose the first by dropping any write that conflicts with any previously seen writes. Our insight is that if the strong central quorum can somehow make an accepted version "more recent" than a dropped fork, then the propagation of the fork would cease in the eventual cloud, reducing the possibility of continued forks.

In order to federated multiple consistency models, there are two integration points: communication and consistency. Our initial approach was to integrate communication at the Raft nodes, by allowing Raft nodes to participate in anti-entropy with the eventual cloud (but not other Raft nodes). Eventual nodes therefore "synchronize" with a local Raft node (modified by some synchronization probability) by exchanging `Gossip` messages with the Raft nodes. A slight increase in the synchronization probability balanced the amount of synchronization with the amount of communication in the eventual cloud given the imbalance in the ratio of eventual nodes to Raft nodes. In order to manage the communications delay between the anti-entropy timeout and leadership coordination, Raft nodes must keep local caches of forked or dropped writes so as to not propagate them back to the eventual cloud or replay them to the leader. However, we have observed that this was not enough to stop the eventual cloud from propagating a fork around Raft, causing further inconsistencies.

Our approach to integrate consistency is to extend each version number with an additional monotonically increasing counter called the *forte* (strong) version that can only be incremented by the leader of the Raft quorum. Because the Raft leader dropped forks or any version that was not more recent than the latest version, incrementing the forte number on commit ensures that only consistent versions are marked. In order

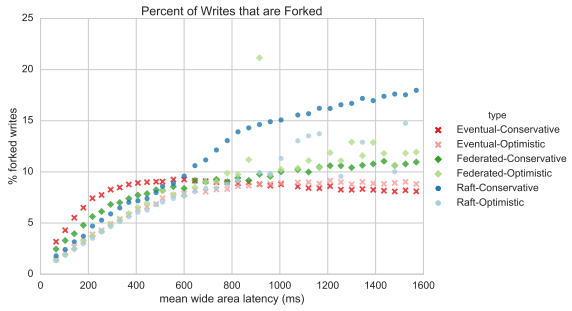

Figure 8: Federated forks as latency and variability increase.

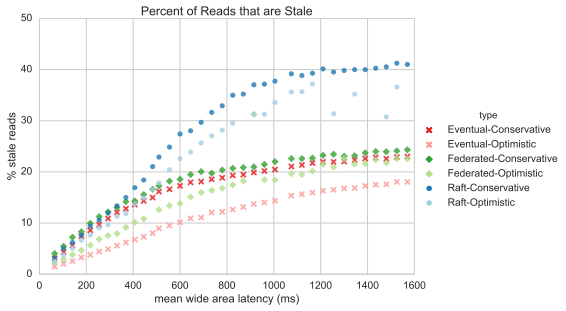

Figure 9: Federated stale reads as latency and variability increase.

to determine the latest version, the forte number is compared first, then the version number, allowing Raft to "bump" the consistent version to a more recent version. In order to prevent that version's children from becoming less recent, on receipt of a version with a higher forte than the local, eventual nodes must search for the forte entry in their local log, find all children of the update, and set the child version's forte equal to that of the parent.

Our preliminary investigations show that this integration works well to curb inconsistencies due to anti-entropy delays as well as those introduced by communication integration. There are, however, quite a few knobs to turn in the system described above. Further investigation into smoothing integration points between consistency protocols and the policies that each define may lead to smoother messaging with fewer resource constraints. We have identified a current bottleneck in the system however: the leadership of the central quorum, through which every single write must pass, no matter the size of the cloud. In order to address this, we propose an adaptation to the central consensus group such that it can provide hierarchical consensus.

4.1.4 Preliminary Results

We have already begun to investigate federated consistency, utilizing the simulation described in Section 3 and plan to submit these results to ICDCS 2017 in December. Our approach focused on two primary experiments: isolating the effect of increasing and more variable latency on consistency protocols and exploring the relationship of conflict likelihood to consistency. In both experiments we fixed as many variables as possible. Each simulation received identical topologies as shown in Figure 2: five locations with 4 replica servers in each. Federated consistency specifies a topology of one Raft replica server per location and all other devices eventually consistent. Each simulation also received identical workloads totaling approximately 28,000 access events across all devices (both reads and writes) to a namespace of 20 objects per replica server. Timing parameters for each protocol – homogenous Raft, eventual consistency, and federated consistency – were computed with a "tick" parameter, T, to preserve the relationship between each protocol and the network environment.

The first experiment consisted of 12 latency ranges with 3 variabilities: low, medium, and high, resulting in 36 normal latency distributions. For each of these distributions we ran simulations for our three protocols with two different "tick" parameters: *conservative* and *optimistic*, where the conservative tick parameter guarantees no out of order messages and the optimistic tick parameter is more performant but provides

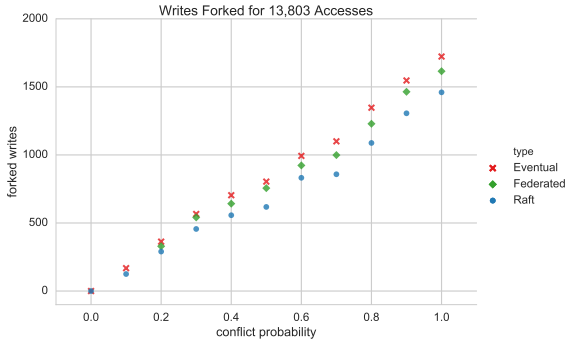

Figure 10: Federated forks as conflict on object accesses increases.

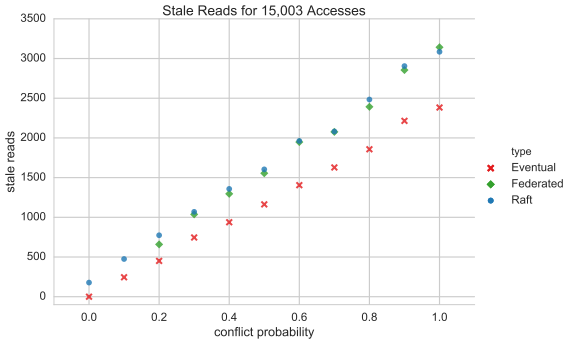

Figure 11: Federated stale reads as conflict on object accesses increases.

no such guarantee. Roughly, the conservative parameter was specified in the original Raft paper and is used routinely in Raft implementations. The optimistic parameter was proposed by [40] to make Raft more available. In total, the latency variation experiments consisted of 216 individual experiments.

The results of the latency variation experiment shows that network environment has a strong influence on consistency; moreover it shows that federated consistency balances the inconsistencies between eventual and Raft, providing more stable consistency guarantees even in variable environments. Both Figures 8 and 9 show a cross-over point where at the lower latency and variability, Raft performs better in terms of percent of forks and stale reads, then around 600ms in Figure 8, eventual starts to outperform Raft. However, no matter the latency distribution, Federated performs equivalently with the better performing protocol, particularly for the conservative timing parameter.

The second experiment fixed the latency distribution and instead varied the probability of conflict, e.g. at $P_c = 0.0$ each device accessed their own, independent namespace and at $P_c = 1.0$ each device shared the exact same namespace. Note that at $P_c = 0.0$ forks are impossible, but stale reads are in the case of Raft: a device can read a stale version of it's own write while awaiting a commit or response from the leader depending on the read policy described in Section 4.1.2. As shown in Figure 10, Federated has fewer forks than an eventually consistent system, thanks to the core consensus group. Federated also performs no worse than Raft in terms of stale reads as shown in Figure 11. When considering inconsistencies, Raft only improves the performance of eventual, but only provides performance benefits to the Raft core group, this is why Federated would benefit from a more available core consensus group, which we propose can be implemented via hierarchical consensus.

4.2 Hierarchical Consensus

Federated consistency presents the opportunity to extend distributed storage systems to medium to large scale networks comprised of dozens of replica servers even in the face of high variability in unstable, mobile network environments. By allowing replicas to participate in eventual consistency replication if required to maintain a minimum quality of service, the system becomes flexible enough to handle variability. The key to Federation, however, is the strong central quorum that coordinates partitions in the eventual cloud, handling conflicts and pruning forked branches, minimizing overall inconsistency in the system.

Given the critical nature of the central quorum, a natural question arises: can the quorum scale to meet the availability needs of the Federated system as the topology grows? At a minimum, members of the quorum must be distributed geographically to provide high availability to anti-entropy synchronization requests, requiring at least one quorum node per location. As topologies scale, not only do the number of devices making requests to the leader increase, but so to do the number of objects that must be managed. Finally, as previously shown in Figure 6 the number of clients making requests increases, the overall visibility of new versions decreases unacceptably. Larger quorums provide more availability but additional resource requirements.

Raft and Paxos optimize the three phase consensus procedure (propose, accept, commit) of quorums by nominating a dedicated proposer, often called the leader or coordinator, who is solely allowed to make voting proposals thus staging the propose phase in advance of any consensus decisions via election. However, the leader is also a single point of failure, and most work in consensus protocols regards correctness and fault tolerance of a system given partitions that remove the leader from connecting to a majority of nodes. Additionally, in order to achieve linearizablity, every single access (including reads) must go through the leader, making the leader a bottleneck whose response performance creates a floor for the overall performance of the system. In order to scale consensus to larger systems, leadership must be addressed.

There are two primary approaches to increasing the availability of the leader in the literature: optimistic "paths" and multiple leaders. The former method, implemented in Fast Paxos [56], Egalatarian Paxos [68], and S-Paxos [18] allows clients to directly contact acceptors (followers) with proposals, bypassing the leader and distributing leadership activities in a so called optimistic "fast path". However, in order to maintain correctness some conflict detection method is required; for example Fast Paxos requires larger quorums for fast path, and Egalitarian Paxos adds dependencies that are checked at "execution", where a dependency failure requires a fall back to the "classic (slow) path". The latter method allocates the decision space to multiple coordinators either by load balancing multiple quorums as in Multi-Paxos [22] or by utilizing a per-tablet (a grouping of objects) quorum as in BigTable [25]. Per-tablet quorums also reduce the likelihood of conflicts introducing the possibility of a hybrid approach: optimistic fast paths on multiple quorums with conflict detection and slow path resolution as in MDCC [47].

Unfortunately these approaches, while working well on smaller scale quorum sizes (3-7 nodes), do not scale well in heterogenous and variable-latency environments. Fast-path requires optimism that conflict is relatively rare, otherwise it performs worse than simple classic-path methods since it must implement conflict detection. In larger networks, conflict is more likely, both because of the increased number of writers and owing to message delay. Allocating multiple, small quorums to different decision spaces requires that objects in different tablets be independent of each other; that is, there is no way to order writes sequentially to objects in tablets maintained by separate quorums. Federated consistency, however, requires a single, sequential ordering of all writes to maintain the eventual cloud and many applications of personal clouds that have implicit dependencies not easily embedded as application-level invariants, such as those specified database schemas.

We propose hierarchical consensus as a solution to scaling consensus to larger networks while still maintaining a sequential ordering across all objects such that dependencies between objects are determined at runtime. Our approach uses a tiered structure of leadership where the object namespace is governed by a root quorum that partitions the namespace to smaller subquorums via consensus decisions. The set of

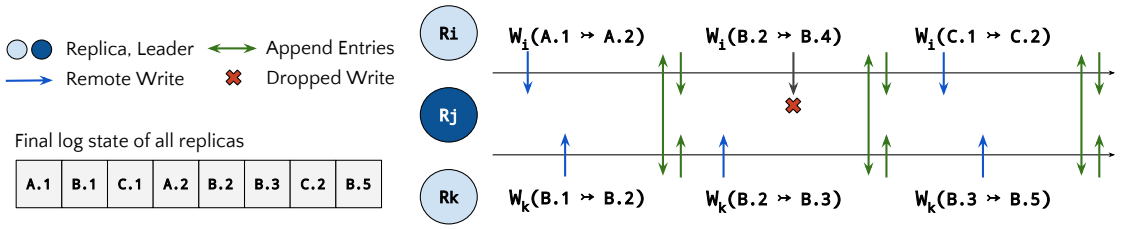

Figure 12: All writes must be forwarded to the leader of a quorum, which must then determine the ordering of those writes, based on a first-writer wins policy. Any inconsistencies such as a fork detected at the leader will be dropped; the remote node can then retry the write or must recover.

allocation decisions creates an ordered set of epochs where each epoch defines a specific mapping of objects to subquorums. Accesses on all objects maintained by a single quorum are dependent on each other and *on no other subquorum*. Every access that occurs in a single epoch is said to have *happened before* every access in previous epochs; every write within a subquorum is ordered with respect to that quorum's decisions, and every write between quorums within a single epoch is said to be concurrent.

Hierarchical consensus provides *sequential consistency* on the entire namespace, flexibly allocating consensus decisions to object dependencies which may change over time. Hierarchical consensus is more available due to the use of multiple, smaller quorums and the localization of leadership to where accesses are occurring.

4.2.1 Consensus Consistency

In order to implement a sequentially consistent file system with consensus, we must translate the generalized consensus model to a specific consistency model. Generalized consensus protocols do not natively implement any particular consistency and only consider the application of ordered commands to a state machine. Commands generally have no dependencies and agreement considers no invariants beyond whether or not accepting the command would violate global ordering. Therefore to describe the use of a consensus protocol for consistency we must describe approaches such that a consensus decision maintains a consistency invariant. The most obvious is to use consensus to grant locks to replica, object pairs and unlocks once the write has been fully replicated. So long as read/write locks are observed, this system implements linearizeability and is equivalent to two phase commit, though such a system suffers from extremely poor performance.

The approach we take is to make accesses the commands, such that the state machine becomes an ordered series of accesses. If both read and write accesses are applied to the log (meaning that both a read and a write must be remote through the leader) then the system remains linearizable. However, in order to improve performance, we allow reads to occur to the local log of each follower, introducing the possibility of a fork if the local read is behind the global state and a write is submitted that modifies the stale read. Forks are the natural consequence of conflicting accesses and while exacerbated by staleness, are a product of versioning in a file system; local caching only makes the problem slightly worse. In order to maintain the no-forks invariant, leaders must therefore check writes to detect forks and drop those that are.

Consider the simple example shown in Figure 12, implemented using the Raft protocol where writes are aggregated in `AppendEntries` (shown as green lines that occur routinely every heartbeat interval). In this

example the object namespace is the set $\{A, B, C\}$ and each version is represented as the object name dot annotated with a monotonically increasing version number (we presume that version 1 of each object has already been written to the log). A write is conducted by reading the latest version of the object and writing the new version, therefore a write by replica i that reads version n and writes to version m on object O is given as follows: $W_i(O.n \rightarrowtail O.m)$. Writes are ordered with respect to their arrival at the leader (R_j), are appended to the logs of the followers on `AppendEntries`, are committed by the leader when a majority of followers responds affirmative to the append RPC, and are marked as committed by followers on the subsequent `AppendEntries`. The leader must reject $W_i(B.2 \rightarrowtail B.4)$ in order to maintain consistency because that write would cause a fork to occur in the log. The final log is identically ordered on all replicas for all objects, therefore we can say we have achieved sequential consistency such that every write that appears in log position i happens before ($\rightarrow$) the log entry at $i - 1$.

In this example, we have only nominated one *explicit* dependency on each write, the parent version, and enforced a no-forks invariant as a policy on this dependency at the leader. A consequence of this style consistency is the *implicit* causality of all prior writes to a given write. For example, $W_i(C.1 \rightarrowtail C.2)$ implies implicit dependencies on $A.2$ and $B.3$, e.g. any versions that could have possibly been read prior to the write (and also the reason that reads must be logged in order to achieve linearizability). Similarly $W_k(B.3 \rightarrowtail B.5)$ has implicit dependencies on $A.2$ and $C.1$, more closely inspecting the log we can see that $C.2 \rightarrow B.5$ and $C.1 \rightarrow C.2$ therefore by transitivity, $C.1 \rightarrow B.5$, a correct interpretation of the log and implicit dependencies.

While implicit dependencies are critical to consistency, we observe that it is unlikely that a write is truly dependent on every historical write but is rather dependent on a local, recent subset of the namespace [10]. Hierarchical consensus therefore makes use of *explicit* causality to allocate the namespace to subquorums in order to balance the leader workload. This has several implications including allowing the ability to lock the ownership of a series of critical files, to specify particular replicas as stable storage for high value data, or to maintain policies and invariants that go beyond ordering; for example spatial polices (what data gets stored), temporal policies (what versions are stored) and synchronization policies (how and when replication occurs).

4.2.2 Namespace Allocation

An open question for our research is how to automatically allocate the namespace such that leadership of a subset of the namespace is local to the accesses and that members of the quorum are distributed to provide wide area durability and availability. The goal of allocation, is shown in Figure 13 - a hierarchy of quorums such that the children of each tier manage consensus decisions on a non-overlapping portion of the namespace.

Consider a motivating example using the notion of *sessions*. Starting from a quiescent state (no accesses), a node or a group of nodes begins reading and writing to a set of objects (perhaps collaborative editing of a document, or a series of financial transactions); we expect that after some finite amount of time, the access pattern will change or cease. We can therefore state that all objects involved in a single session are explicitly causally related to each other and to no other objects in the namespace. Whether we describe sessions as a fixed, sliding window or as something more variable, some automatic determination that those objects should be coordinated together is necessary. To that end we will define a *tag* as a time-annotated subset of

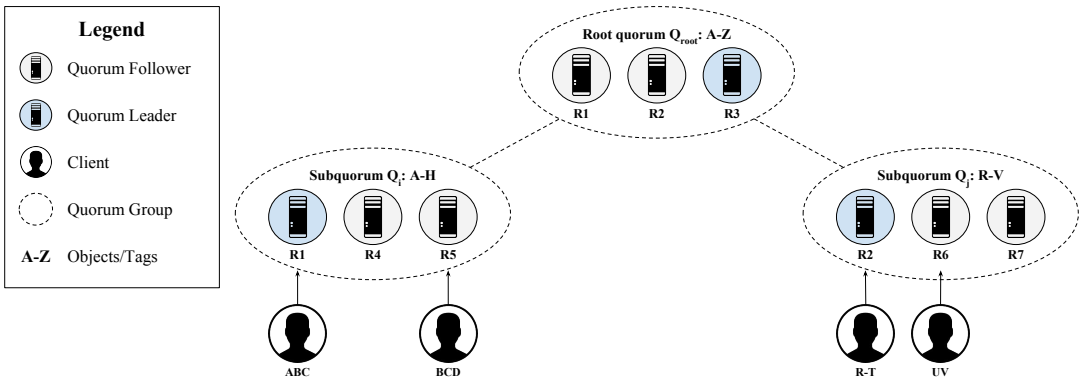

Figure 13: The root partitions the object namespace across one or more subquorums. Each subquorum serializes all accesses to the objects in its tag; accesses in distinct subquorums are, by definition, concurrent.

the namespace and a *tagspace* as the set of non-overlapping tags that compose the namespace for a given time period.

Hierarchical consensus starts with a root quorum whose responsibility is to govern the entire namespace. The root quorum does so by maintaining a root *epoch*, a monotonically increasing counter which identifies the current *tagspace*. In addition to consensus decisions related to leader election, accesses, and membership changes, decisions that modify the tagspace also require consensus. We define two primary operations: *split* creates a new subquorum, splitting a larger tag from the previous tagspace and *join* removes a subquorum, joining two smaller tags from the previous tagspace. Any decision that modifies the tagspace (thus creating a new tagspace) requires an increment of the *epoch*.

The fundamental relationship between accesses in epochs is as follows: any access that happens in epoch i happened before every access in epoch $i-1$; accesses in different tags but in the same epoch happen concurrently from the global perspective, but are ordered locally in the quorum that governs that tag. As a result, every log has a determined, sequentially consistent and correct ordering, though the logs of two individual replicas may differ. Hierarchical consensus cannot provide linearizability, but does provide sequential consistency.

The root consensus group must coordinate all tagspace changes. Let's consider the example where we want to exchange part of a tag between two subquorums, e.g. we want to transform a tagset $\{ABC\} \rightarrowtail \{ABGH\}$ and $\{DEFGH\} \rightarrowtail \{CDEF\}$. Both subquorums have a portion of the tag that the want to give up and a portion they want to receive. We propose, initially, that this is a two phase process. Both subquorums make their requests to the leader, who may aggregate several namespace changes into a single one. While the root quorum gets consensus to make the epoch change, subquorums can continue operating on their own tags. Once the root quorum updates the epoch, it communicates the change to the subquorums. The subquorum then increments its epoch and acknowledges its change to the root quorum. At this point, the subquorum can operate on the portion of the tag it owned before, *but not the portion of the tag that is being modified*. Once the root quorum gets confirmations of epoch update from all subquorums, it notifies the subquorums that it can begin operating on the complete tag from the new epoch.

Only the subquorums that are involved in the tagspace changes need be notified and involved in the two

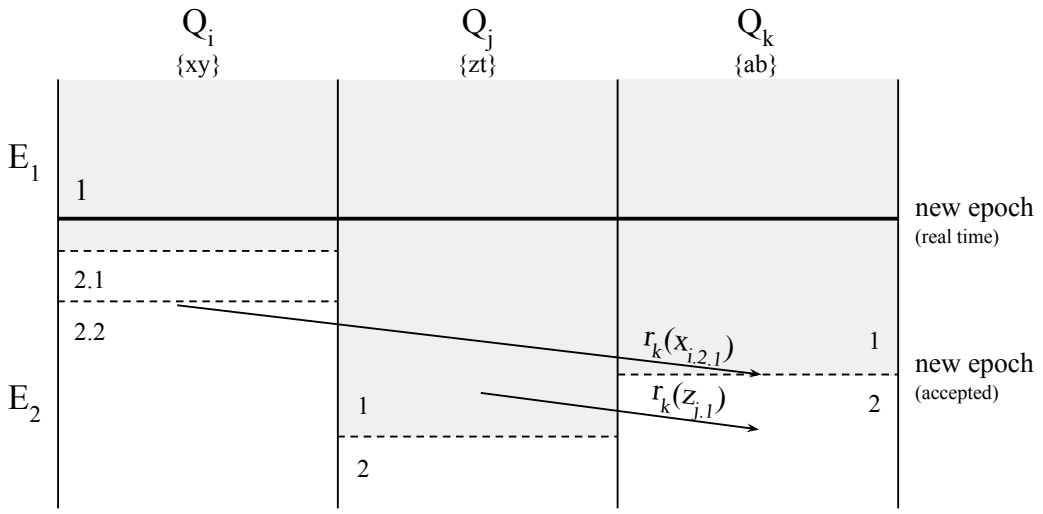

Figure 14: The gray region shows the "fuzzy" boundary between epochs E_1 and E_2. None of the subquorums advance to the new epoch immediately. Q_i does so first, subsequent reads of Q_i's data by Q_k forces Q_k to advance, Q_i serializes the remote access by creating subepoch $E_{i.2.2}$

phase epoch change coordinated by the root epoch. Other subquorums can update their epoch number at no cost when they see the new epoch number from remote requests from other subquorums or when they are notified by the root epoch. In this way, non-responding subquorums do not block tagspace changes for other quorums. Because the tag is left behind in the previous epoch, all writes in that tag will be ordered before writes in the next epoch. However, because no writes in the next epoch depend on these writes safety is still guaranteed. This means that subquorums can have "fuzzy epochs" as shown in Figure 14, wherein subquorums are behind others; this property gives hierarchical consensus fault tolerance. It does, however, also present a challenge: what if the subquorum with a tag never comes back online and another subquorum wishes to access that part of the tagspace? We propose to investigate mechanisms of restricting long running accesses to a single tagspace that may involve conflict, such as periodic tag renewals.

4.2.3 Operation

The namespace allocation creates a tiered structure of leadership such that the root quorum is responsible for the entire namespace, subquorums are responsible for tags, and leaves are responsible for handling accesses to their tag. Any access to an object must be forwarded as a remote access to the leader of the quorum that handles the tag that encapsulates the object. The only exception is the optimization made for members of the quorum who can read locally any object in their tag space. By optimizing the locality of access and spreading the workload to multiple leaders, we hope to show that the overall performance of the system increases.

When a member of a quorum wishes to access an object that is not in that quorum's tag, one of two things must happen: either a tag decision must be made to reallocate the tag space such that it is local to the new accessor or some mechanism must allow remote accesses between quorums. The former case is expensive,

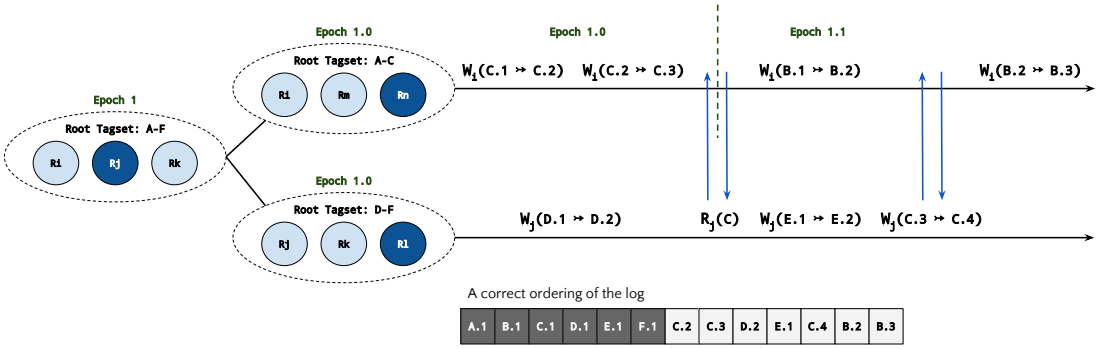

Figure 15: Remote reads and writes may be cheaper than changing epochs frequently. In order to order remote accesses correctly, subepochs are used to show ordering dependencies between accesses in two tag spaces. Here the remote read from the D-F subquorum causes the A-C quorum to create a new subepoch.

but if the accesses to that object become routine then the upfront cost will pay for future accesses. However, for non-routine accesses some mechanism for remote access is required.

The issue is that a remote access from one tag to another creates an implicit dependency between all accesses that happened before the remote access and all those that follow the local one. Correctness is maintained by the ordering of epochs, so some local split is required. Consider the example in Figure 15: the remote read access, $R_j(C)$ implies any writes to tag $D-F$ following the read depends an all writes in tag $A-C$ that occurred before the read. Furthermore, the remote write $W_j(C.3 \rightarrowtail C.4)$ is a non-conflicting write, depends on all writes on tag $D-F$ that happened before *and* depends on all writes in tag $A-C$ that occurred before the remote read, and occurs concurrently with any writes that happen after so long as there is no conflict.

Our proposed solution is similar to the epoch solution: each tag quorum maintains a per-epoch, monotonically increasing sub-epoch counter. In the case of a remote access, that counter is incremented to demarcate the ordering of all accesses in the tag that happened before the remote access and all those that follow. This counter must be replicated in addition to the version number, and can therefore be seen as an explicit dependency applied to all remote writes. As a result, any quorum that receives these ordering indicators can appropriately order their logs.

This mechanism also leads us to suspect that generalizing the consensus hierarchy to arbitrarily deep levels is possible; particularly since the notion of sub-epoch numbers already exists. Clearly some coordination is required to ensure that tag space changes filter down to all leaf nodes, and that is the primary subject of our future work related to this proposal.

In addition to correctness, we also propose to show failure tolerance by utilize Raft-style quorum mechanics. There are two distinct kinds of failure in a hierarchical system: replica failures (a single node crashes and dies) and partitions, where parts of the network are cut off from receiving messages. We believe that we can model failure tolerance from both of these perspectives through a formulation of quorum size, minimal intersection set between quorums, and the depth/breadth of the hierarchy. We propose to show correctness in the face of failure in the least and will stretch to show inherent robustness due to the structure of the hierarchy.

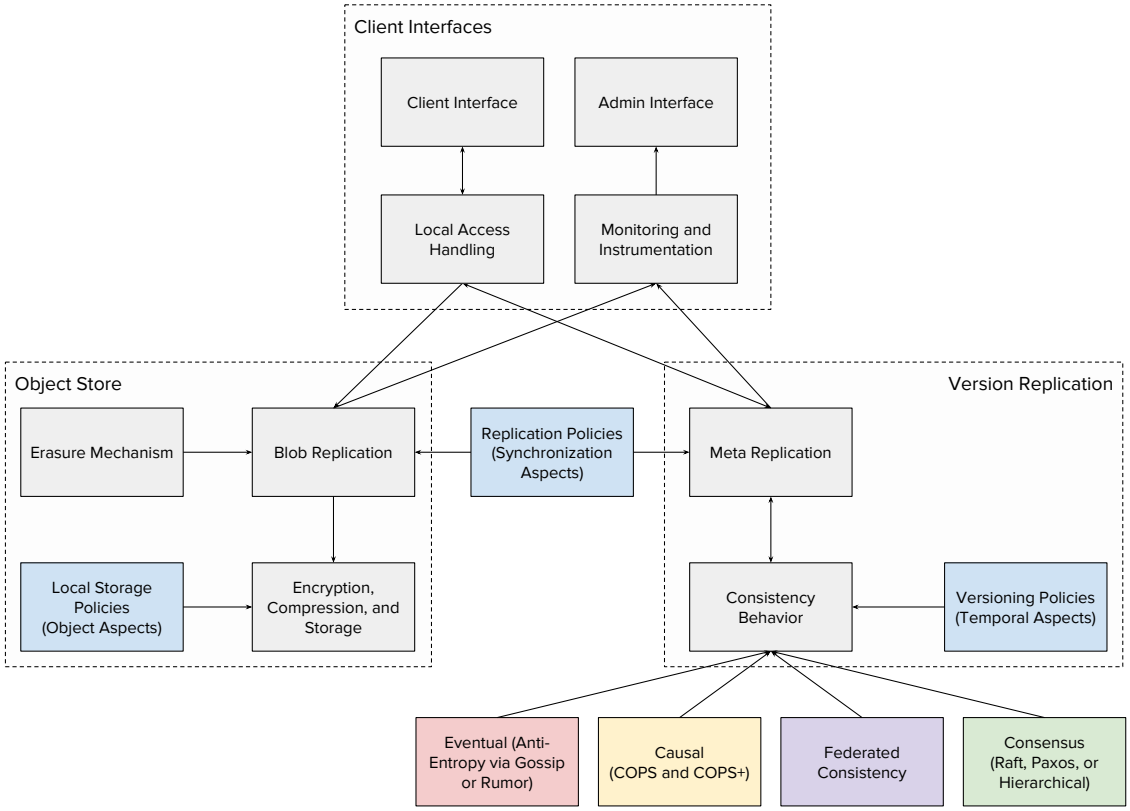

Figure 16: We propose to implement a file system called FlowFS in the Go programming language. This component model of our architecture shows a decoupling between version and object replication. Consistency models only apply to version replication because they define the view of the file system. Object replication can be done optimistically with weak consistency protocols.

4.3 System Description

Experimentation by simulation is essential to give us the flexibility to describe a wide variety of environments, parameters, and use-cases. However we propose to implement a distributed file system called FlowFS to more completely explore the use of federated consistency and hierarchical consensus. FlowFS, implemented in the Go programming language, will allow us to quantitatively describe real-world environments and utilization and to show how our proposed consistency model is experienced by users. In this section we will describe our proposed implementation, described by the component architecture in Figure 16.

FlowFS aggregates individual accesses into *Close-To-Open* (CTO) [41, 70, 65, 61, 75] consistency such that read and write accesses are "whole file" [71]. Furthermore, with respect to local accesses we guarantee that a read returns the last write (given no remote updates, *Read Your Writes Consistency*) and that writes are atomic with respect to each other (*Monotonic Write Consistency*) [15]. These client-oriented consistency guarantees are necessary to provide file system semantics and are enabled by local cacheing such that intermediate sync and flush operations are written to disk and replication only occurs when a file is closed. This however, does present an increased opportunity for inconsistency as the editing latency of a

file increases the likelihood of concurrent accesses. Two concurrent write accesses create forks in the version history, a conflict that must be resolved in order to maintain file system consistency. However, in order to describe conflict resolution (both manual and automatic), we first must describe how objects and their versions are defined.

The file system is defined as a hierarchical namespace, which could be described as an object store with complex keys similar to Amazon S3 [16]. Each object has a unique name which is associated with a piece of metadata called a *version*. Versions are identified by a monotonically increasing version number, implemented either as a vector clock [76] (or a simple Lamport scaler) in the case of a fixed topology or as a vector stamp [5] in the case of dynamic topologies. Version numbers are utilized to provide fork detection and identify potential conflicts. Versions are related to each other with dependency information and all versions have at least one dependency: the parent version that it was derived from. As such, the namespace identifies a *object history* and the most recent versions local to a replica server define the *view* of the file system.

When a file is closed after editing, the data associated with the file is chunked into a series of unique, fixed-length blobs identified by a hashing function applied to the data. The version created by the write access to the file specifies the blobs and their ordering that make up the file. However because blobs are unique, we assert that consistent replication can be decoupled from blob replication. Every device in the system therefore has a complete view of the file system because all version meta data is completely replicated without the storage resource constraints of fully replicated blobs. In fact, every device can specify distinct local storage policies (object aspects) that define what blobs reside on their local disks. The protocol and timing used to replicate both versions and objects can also be modified locally to specify a message budget or other resource restrictions (synchronization aspects). Techniques like hoarding [49] and TCP layer replication [83] can improve blob replication, optimistically colocating blobs with likely read accesses. However, if a blob is not available locally a remote access to a storage device with the blob is all that is required.

Decoupling blobs from versions also provides a unique conflict detection and resolution mechanism that can reduce the impact of the increased likelihood of concurrent access due to CTO consistency. Similar to Git and OriFS [65] version trees can be implemented with data structures that perform deeper conflict detection down to the blob level. If two different blobs have been modified, automatic conflict resolution can be applied similar to Git merge. If the same blobs have been modified a manual intervention is required; on close, a diff can immediately be shown to the user so that conflicts are handled close to the time that they occur. This mechanism works best for files that are edited in multiple locations, such as documents, and poorly for files that are completely modified no matter the size of the change, such as binary files. We assert that binary files, primarily images, video, binary executables, etc. are generally created once and updated rarely and therefore generally have a low likelihood of conflict, requiring no special conflict resolution mechanism.

System consistency depends only on the replication of version information since a version defines what is visible on each replica to be read. When a file is opened, a read access occurs, and at close a write access may occur if the user has modified the file. Consistency protocols only deal with version metadata and therefore the size of messages is much smaller, allowing for high throughput. Temporal aspects can allow local policies regarding how much history per version is stored on the local replica, further decreasing storage requirements.

Consistency is implemented such that each replica server maintains a log of versions applied to their local

view. A read access to a particular object simply looks up the latest local version of that object. Because dependency information can be embedded into a write, it is not necessary to include read accesses in the log. For example, in order to create a transaction that reads from objects X and Y, performs a computation then updates objects Y and then Z: the write to Y would include as a dependency the earlier version of Y and the read version of X and the write to Z would include the updated version of Y and the read version of X. Other notions of dependencies include implicit session dependencies, e.g. all writes are dependent on any access that occur within a minimum time threshold of each other, or explicit dependencies that are added by the application. Defining additional dependencies is the subject of future work and stretch goals related to this dissertation.

Finally, a note on security in FlowFS. Although security is not a specific focus of our research, we believe that security should not be an afterthought. Because devices must be added to the system by joint consensus or through a trusted administrator, our research focuses on non-byzantine behavior. Messages between replica servers will be authenticated and signed by per-client tokens that are generated when devices are added to the topology. All communications will be secured by encrypted transport mechanisms such as TLS or SSL. The data itself can be secured by encrypting individual blobs with user-specific keys; blob replication to non-owner controlled device does not necessarily mean that the blobs can be read by all users. These steps are by no means comprehensive, but provide a baseline minimum security guarantee which could be explored further in future research.

4.4 Adaptive Consistency

Both the size and volatility of a user-centric dynamic cloud require the distributed file system to be as responsive and adaptive as possible. Because there are many users and devices each with individual resources and requirements, direct administration of policies and configurations is difficult; managing those policies and configurations in response to changing environments is impossible. Together, federated consistency and hierarchical consensus provide mechanisms that deal with unstable network environments and the unique challenges a multi-user, multi-device system may encounter. However, neither protocol specifically deals with the dynamic or mobile nature of the network; and while they provide responsiveness at the user layer, cannot take advantage of boosts in bandwidth or respond to gaps our outages.

Therefore we propose the final step in the study of user-centric dynamic clouds is the real time adaptation of the network according to observed latency values. We have observed that eventually consistent nodes can change their timing parameters at will to take advantage of lower latencies or to save work in sparse network conditions. While joint consensus may be required to adapt the parameters of a consensus group, lowering the timing parameter dramatically improves throughput and commit latency and in the face of increasing latency, ensuring that the timing parameters are conservative enough such that messages aren't received out of order or such that "leader thrashing" occurs as much delayed heartbeat messages cause candidacies.

By equipping the system with online monitoring of local network conditions and access patterns, we propose that decentralized administration through machine learning techniques will allow the system to be as responsive as possible, involving users in *active learning* [42, 74] while taking advantage of immediate local optimizations [72]. The application of supervised and unsupervised models based on historical training data after comprehensive feature analysis of both human and machine semantic data will allow the system to

automatically make administrative decisions for a wide space of instances based on historical data. Per-user, and per-object models can also be trained and applied as ensembles so that different parts of larger systems behave differently in response to specific desired behaviors. Reinforcement learning allows for online performance tuning, anomaly detection, and recovery.

We formulate the machine learning problem such that each replica behaves as an independent cognitive agent with the goal of maximizing *local utility*, which can be defined specifically to the device: resource management, minimization of forks, lower latency messages, improved availability, etc. [72]. With this formulation, our first ML approach is to use cooperative reinforcement learning [59, 58], a technique that has been applied to packet routing in dynamic networks [20] and more recently to cognitive radio networks [28]. Similar to these approaches, each agent (a replica) will modify its local timing parameters, e.g. the rate at which it sends replication messages, the timeout before message retry, whether or not it aggregates messages, etc. in response to its observations about its local environment.

An example of configuration optimization through online bandit algorithms [19] involves the replication of data blobs. Because blob replication is decoupled from consistency, eventually consistent replication via anti-entropy is appropriate. The timely propagation of a blob to all nodes in a large network depends on the random selection of a neighbor to perform anti-entropy with. Instead of simply implementing uniform random selection, a multi-armed bandit can be used such that neighbors that are "better" to gossip with are selected. Reinforcement should prefer neighbors that have not seen updates and whose latency is as low as possible. In this way, an anti-entropy topology can be learned to maximize propagation.

Not all configuration changes can be made independently, however, because consensus parameters and quorum membership require coordination. In this case, supervised learning algorithms can be used to classify the current environment of a quorum, which can then be used to set policies. For example, multi-class classifiers [77] can use features such as the observed distribution of latency, number of IP address changes, number of users, time online, etc. to classify the current quorum as "stable", "moderate", or "unstable". Quorums in stable environments may pursue more aggressive messaging to minimize forks and strong consistency, while those in unstable environments take a deliberate approach to ensure that progress can be made. Anomaly detection algorithms [7, 31] can be used to determine if an unusual environment exists, which can then be used to guide voting policies.

Policies themselves can be learned or induced based on rule learning. Decision tree induction has long been used in manufacturing environments for control purposes [33]. Similarly, policies can be selected using induced rules based on observed states of the system. Such rules will be trained from historical performance data.

Finally similarity mechanisms [17] can be used to apply policies to versions, objects, or even replica configurations. If users set policies for a type of file (for example binary files), similarity mechanisms can be used to propose the application of that policy to another group of files. Similar files can be replicated together and otherwise globally managed together. When new replicas join the topology, then can be automatically configured by their characteristics. Unsupervised clustering mechanisms allow the broad application of a variety of aspects across the entire network without the individual management of replicas or file types.

5 Timeline

In order to meet the requirements of this proposal and provide substantive results for a dissertation, we propose to undertake the following projects, along with their given priority and timeline.

Project	Priority	Time Estimate
Simulation of Federated Consistency	△	1 months
Simulation of Hierarchical Consensus	△	2 months
Implementation of the FlowFS	△	3-4 months
Evaluation of FlowFS on real workloads	△	2-3 months
Proof of correctness and consistency	△	1 month
Heuristics to optimize and allocate FlowFS	▽	1 month
Online optimization and consistency adaptation	▽	2-3 months
TOTAL		12-16 months

After investigations and evaluations of federated consistency and hierarchical consensus in simulation, I propose to implement a file system utilizing both mechanisms and verify the system meets consistency requirements on large networks in real world workflows. Goals marked with △ indicate necessary checkpoints to the successful completion of the dissertation, including some proof of correctness. I further propose the stretch goals (marked with ▽) of real time adaptation and online optimization, investigating the application of both heuristics and active machine learning techniques to learn usage patterns and improve consistency and performance.

Based on preliminary work done in simulation as well as the above timeline, I propose the following paper and conference submission goals:

Paper	Title	Location	RFP	Date
Federated Consistency	IEEE ICDCS 2017	Atlanta, GA	Dec 5, 2016	Jun 5-8 2017
User-Centric Dynamic Clouds	ACM HotStorage 2017	Santa Clara, CA	N/A	Jul 10-11, 2017
Hierarchical Consensus	ACM PODC 2017	Washington DC	Feb 10, 2017	July 2017
Hierarchical FlowFS	ACM SOSP 2017	Shanghai, China	Apr 21, 2017	Oct 29-31, 2017
Responsive Consistency	USENIX NSDI 2018	N/A	N/A	N/A

6 Conclusion

In this proposal we have presented two novel approaches to provide responsiveness in user-centric dynamic clouds: federated consistency and hierarchical consensus, as well as stretch proposals related to automatic Adaptive Consistency. Research into these approaches is important because the number of computing devices per person is increasing and new devices are coming online via the Internet of Things. Current approaches of centralized, cloud distributed data storage provide mobile availability and durability, but do not lend themselves well to multi-user, collaborative environments. We believe that as a result, user-centric dynamic personal clouds will become increasingly important to support the increase in the number of local devices.

We have grounded our proposal in consistency, consensus, and replication. A generalized consistency model shows that consistency is a scale along dimensions of ordering and staleness; we have extended this model to account for file-system specific semantics and specifically measure forks and stale reads, symptoms of inconsistent ordering. Consensus allows the strong coordination of a distributed system, and we specifically consider the Raft consensus protocol. Finally, replication via anti-entropy gossip and broadcast protocols must consider resource constraints. By applying consistency only to the metadata of objects and by replicating immutable data blocks, we believe that a global, scalable network file system is possible.

We propose to show through simulation and through a file system implementation that federated consistency and hierarchical consensus together will lead to a highly available system with strong consistency guarantees. Federation will make use of a strong-central quorum to synchronize an eventual cloud, minimizing overall inconsistency in the system, while allowing eventual nodes to make progress with reduced coordination requirements. The central quorum needs to be able to scale to dozens or hundreds of nodes and so we propose hierarchical consensus as a means of partitioning consensus decisions to particular objects across the namespace to different leaders. This load balancing will increase throughput but also increase correctness. Together and with potential future research into online optimization we believe that this research will lead to a high quality dissertation.

A Reading List

A.1 Consistency

1) Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978

2) David Bermbach and Jörn Kuhlenkamp. Consistency in distributed storage systems. In *Networked Systems*, pages 175–189. Springer, 2013

3) Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM*, 29, 1995

4) Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, 23(2):279–302, 2014

5) Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007

6) Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011

7) Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012

8) Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994

9) Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. *Proc. of the 11th USENIX NSDI*, 2014

10) James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013

A.2 Consensus

1) Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001

2) Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM, 2007

3) Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006

4) Leslie Lamport. Generalized consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005

5) Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013

6) Martin Biely, Zoran Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 111–120. IEEE, 2012

7) H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. *ArXiv e-prints*, August 2016

8) Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014

9) Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21, 2015

10) Butler W. Lampson. How to build a highly available system using consensus. In *Distributed Algorithms*, pages 1–17. Springer, 1996

A.3 Replication

1) John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, and others. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000

2) Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM, 1996

3) Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 565–574. IEEE, 2000

4) David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491. IEEE, 2003

5) Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013

6) Jinyuan Li, Maxwell N. Krohn, David Mazieres, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, volume 4, pages 9–9, 2004

7) Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, 2015

8) Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazieres. Replication, history, and grafting in the Ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 151–166. ACM, 2013

9) Peter J Keleher. Decentralized replicated-object protocols. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 143–151. ACM, 1999

10) Geoffrey H. Kuenning and Gerald J. Popek. *Automated Hoarding for Mobile Computers*, volume 31. ACM, 1997

References

[1] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

[2] Divyakant Agrawal, Manhoi Choy, Hong Va Leong, and Ambuj K. Singh. Mixed consistency: A model for parallel programming. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 101–110. ACM, 1994.

[3] Mustaque Ahamad, Gil Neiger, Prince Kohli, James Burns, Phil Hutto, and T. E. Anderson. Causal memory: Definitions, implementation and programming. *IEEE Transactions on Parallel and Distributed Systems*, 1:6–16, 1990.

[4] Raihan Al-Ekram and Ric Holt. Multi-consistency data replication. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 568–577. IEEE, 2010.

[5] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps-decentralized version vectors. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 544–551. IEEE, 2002.

[6] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M Hellerstein. Consistency without borders. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, page 23. ACM, 2013.

[7] Fabrizio Angiulli and Clara Pizzuti. Fast Outlier Detection in High Dimensional Spaces. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 15–27. Springer, 2002.

[8] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[9] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations (Extended Version). *arXiv preprint arXiv:1302.0309*, 2013.

[10] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012.

[11] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination-avoiding database systems. *arXiv preprint arXiv:1402.2237*, 2014.

[12] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 International Conference on Management of Data*, pages 761–772. ACM, 2013.

[13] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[14] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, 23(2):279–302, 2014.

[15] David Bermbach and Jörn Kuhlenkamp. Consistency in distributed storage systems. In *Networked Systems*, pages 175–189. Springer, 2013.

[16] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, page 1. ACM, 2011.

[17] Indrajit Bhattacharya and Lise Getoor. A Latent Dirichlet Model for Unsupervised Entity Resolution. In *SDM*, volume 5, page 59. SIAM, 2006.

[18] Martin Biely, Zoran Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 111–120. IEEE, 2012.

[19] Djallel Bouneffouf, Romain Laroche, Tanguy Urvoy, Raphael Féraud, and Robin Allesiardo. Contextual Bandit for Active Learning: Active Thompson Sampling. In *International Conference on Neural Information Processing*, pages 405–412. Springer, 2014.

[20] Justin A. Boyan and Michael L. Littman. Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach. *Advances in neural information processing systems*, pages 671–671, 1994.

[21] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association, 2006.

[22] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 316–317. ACM, 2007.

[23] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM, 2007.

[24] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[26] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301. IEEE, 2012.

[27] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Consistency in the cloud: When money does matter! In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 352–359. IEEE, 2013.

[28] Charles Clancy, Joe Hecker, Erich Stuntebeck, and Tim O'Shea. Applications of Machine Learning to Cognitive Radio Networks. *IEEE Wireless Communications*, 14(4):47–52, 2007.

[29] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[30] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[31] Paul Dokas, Levent Ertoz, Vipin Kumar, Aleksandar Lazarevic, Jaideep Srivastava, and Pang-Ning Tan. Data Mining for Network Intrusion Detection. In *Proc. NSF Workshop on Next Generation Data Mining*, pages 21–30, 2002.

[32] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: Understanding personal cloud storage services. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, pages 481–494. ACM, 2012.

[33] Bob Evans and Doug Fisher. Overcoming process delays with decision tree induction. *IEEE expert*, 9(1):60–66, 1994.

[34] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In *OSDI*, volume 10, pages 337–350, 2010.

[35] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM, 1996.

[36] Rachid Guerraoui and Corine Hari. On the consistency problem in mobile distributed computing. In *Proceedings of the Second ACM International Workshop on Principles of Mobile Computing*, pages 51–57. ACM, 2002.

[37] Pat Helland. Immutability changes everything. *Queue*, 13(9):40, 2015.

[38] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[39] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. *ArXiv e-prints*, August 2016.

[40] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21, 2015.

[41] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[42] Adam Kalai and Santosh Vempala. Efficient Algorithms for Online Decision Problems. *Journal of Computer and System Sciences*, 71(3):291–307, 2005.

[43] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 565–574. IEEE, 2000.

[44] Peter J Keleher. Decentralized replicated-object protocols. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 143–151. ACM, 1999.

[45] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491. IEEE, 2003.

[46] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.

[47] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[48] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, and others. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[49] Geoffrey H. Kuenning and Gerald J. Popek. *Automated Hoarding for Mobile Computers*, volume 31. ACM, 1997.

[50] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[51] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[52] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[53] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[54] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[55] Leslie Lamport. Generalized consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[56] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[57] Butler W. Lampson. How to build a highly available system using consensus. In *Distributed Algorithms*, pages 1–17. Springer, 1996.

[58] John Langford and Tong Zhang. The Epoch-Greedy Algorithm for Multi-Armed Bandits with Side Information. In *Advances in Neural Information Processing Systems*, pages 817–824, 2008.

[59] Martin Lauer and Martin Riedmiller. An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer, 2000.

[60] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

[61] Jinyuan Li, Maxwell N. Krohn, David Mazieres, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, volume 4, pages 9–9, 2004.

[62] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. *Proc. of the 11th USENIX NSDI*, 2014.

[63] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[64] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[65] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazieres. Replication, history, and grafting in the Ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 151–166. ACM, 2013.

[66] David Mazieres. Paxos made practical. *Unpublished manuscript*, 2007.

[67] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[68] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*, 2012.

[69] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[70] Lily B Mummert, Maria R Ebling, and Mahadev Satyanarayanan. *Exploiting Weak Connectivity for Mobile File Access*, volume 29. ACM, 1995.

[71] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[72] Martin J. Oates and David Corne. Investigating Evolutionary Approaches to Adaptive Database Management against Various Quality of Service Metrics. In *International Conference on Parallel Problem Solving from Nature*, pages 775–784. Springer, 1998.

[73] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[74] Thomas Osugi, Deng Kim, and Stephen Scott. Balancing Exploration and Exploitation: A New Algorithm for Active Machine Learning. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 8–pp. IEEE, 2005.

[75] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Riviere, and Pascal Felber. GlobalFS: A Strongly Consistent Multi-Site File System.

[76] D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering*, (3):240–247, 1983.

[77] Huimin Qian, Yaobin Mao, Wenbo Xiang, and Zhiquan Wang. Recognition of human activities using SVM multi-class classifier. *Pattern Recognition Letters*, 31(2):100–111, 2010.

[78] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[79] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems*. Prentice-Hall, 2007.

[80] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149. IEEE, 1994.

[81] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM*, 29, 1995.

[82] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, 2015.

[83] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. Operating system support for massive replication. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, pages 227–230. ACM, 2002.

[84] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[85] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239–282, 2002.

[86] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, pages 75–87. ACM, 2015.

[87] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ViewBox: Integrating local file systems with cloud storage services. In *FAST*, pages 119–132, 2014.