Consensus in the Cloud: Paxos Systems Demystified

Ailidani Ailijiang*, Aleksey Charapko† and Murat Demirbas‡
Computer Science and Engineering
University at Buffalo, SUNY
Buffalo, NY 14260
*Email: *ailidani@buffalo.edu, †acharapk@buffalo.edu, ‡demirbas@buffalo.edu,*

Abstract—**Coordination and consensus play an important role in datacenter and cloud computing, particularly in leader election, group membership, cluster management, service discovery, resource/access management, and consistent replication of the master nodes in services. Paxos protocols and systems provide a fault-tolerant solution to the distributed consensus problem and have attracted significant attention as well as generating substantial confusion. In order to elucidate the correct use of distributed coordination systems, we compare and contrast popular Paxos protocols and Paxos systems and present advantages and disadvantages for each. We also categorize the coordination use-patterns in cloud, and examine Google and Facebook infrastructures, as well as Apache top-level projects to investigate how they use Paxos protocols and systems. Finally, we analyze tradeoffs in the distributed coordination domain and identify promising future directions for achieving more scalable distributed coordination systems.**

1. Introduction

Cloud computing deals mainly with big data storage, processing, and serving. While these are mostly embarrassingly parallel tasks, coordination still plays a major role in cloud computing systems. Coordination is needed for leader election, group membership, cluster management, service discovery, resource/access management, consistent replication of the master nodes in services, and finally for barrier-orchestration when running large analytic tasks.

The coordination problem has been studied by the theory of distributed systems extensively under the name "distributed consensus". This problem has been the subject of several impossibility results: while consensus is easy in the absence of faults, it becomes prone to intricate failure-scenarios in the presence of lossy channels, crashed participants, and violation of synchrony/timing assumptions. Several algorithms have been proposed to tackle the problem, however, Paxos introduced in 1989 [1] stood out from the pack as it provided a simple formally-proven algorithm to deal with the challenges of asynchrony, process crash/recovery, and message loss in an elegant and uniform manner.

Paxos's rise to fame had to wait until after the large-scale web services and datacenter computing took off in 2000s. Around that time Google was already running into the fault-induced corner cases that cause service downtimes. A fault-tolerant coordination service was needed for the Google File System (GFS), and Google adopted Paxos for implementing the GFS lock service, namely the Google Chubby [2]. The Google Chubby project boosted interest in the industry about using Paxos protocols and Paxos systems for fault-tolerant coordination.

An open-source implementation of the Google Chubby lock service was provided by the Apache ZooKeeper project [3]. ZooKeeper generalized the Chubby interface slightly and provided a general ready-to-use system for "coordination as a service". ZooKeeper used a Paxos-variant protocol Zab [4] for solving the distributed consensus problem. Since Zab is embedded in the ZooKeeper implementation, it remained obscure and did not get adopted as a generic Paxos consensus component. Instead of the Zab component, which required a lot of work for integrating to the application, ZooKeeper's ready-to-use file-system abstraction interface got popular and became the de facto coordination service for cloud computing applications. However, since the bar on using the ZooKeeper interface was so low, it has been abused/misused by many applications. When ZooKeeper is improperly used, it often constituted the bottleneck in performance of these applications and caused scalability problems.

Recently Paxos protocols and Paxos systems quickly grew in number adding further options to the choice of which consensus/coordination protocols/systems to use. Leveraging ZooKeeper, the BookKeeper [5] and Kafka [6] projects introduced log/stream replication services. The Raft protocol [7] went back to fundamentals and provided and open-source implementation of Paxos protocol as a reusable component. Despite the increased choices and specialization of Paxos protocols and Paxos systems, the confusion remains about the proper use cases of these systems and about which systems are more suitable for which tasks. A common pitfall has been to confuse the Paxos protocols with Paxos systems build on top of these protocols (see Figure 1). Paxos protocols (such as Zab and Raft) are useful for low-level components for server replication, whereas Paxos systems (such as ZooKeeper) have been often shoehorned to that task. The proper use case for Paxos systems is in highly-available/durable metadata management, under the

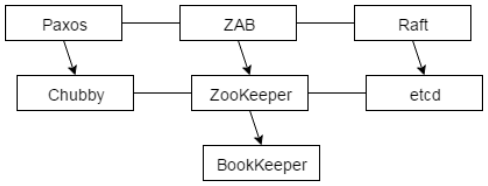


Figure 1. Paxos protocols versus Paxos systems

conditions that all metadata fit in main-memory and are not subject to very frequent changes.

Contributions of this paper.

1) We categorize and characterize consensus/coordination use patterns in the cloud, and analyze the needs/requirements of each use case.
2) We compare and contrast Paxos protocols and Paxos systems and present advantages and disadvantages for each. We present proper use criteria for Paxos systems.
3) We examine Google and Facebook infrastructure as well as Apache top-level projects to evaluate their use of Paxos protocols and systems.
4) Finally we analyze tradeoffs in the distributed coordination domain and identify promising future directions for achieving more scalable distributed coordination systems.

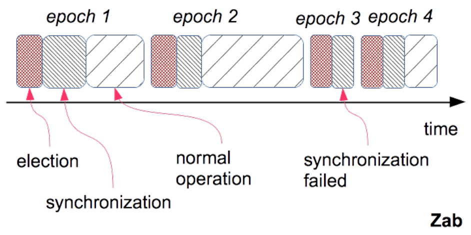
2. Paxos Protocols

In this section, we present Paxos protocol variants and compare and contrast their differences.

2.1. Similarities among Paxos protocols

The **original Paxos protocol**, detailed in [1], was developed for achieving fault-tolerant consensus and consequently for enabling fault-tolerant state machine replication (SMR) [8]. Paxos employs consensus to serialize operations at a leader and apply the operations at each replica in this exact serialized order dictated by the leader. The Multi-Paxos (a.k.a. multi-decree Paxos) flavor have extended the protocol to run efficiently with the same leader for multiple slots [9], [10], [11], [12], [13]. In particular, work by Van Renesse [14] presented a reconfigurable version of Multi-Paxos with a detailed and easy to implement operational specification of replicas, leader and acceptors.

Zab (ZooKeeper Atomic Broadcast) is the Paxos-variant consensus protocol that powers the core of ZooKeeper, a popular open-source Paxos system [3], [4]. Zab is referred to as an atomic broadcast protocol because it enables the nodes to deliver the same set of transactions (state updates) in the same order. *Atomic broadcast* or total order broadcast and consensus are equivalent problems [15], [16].



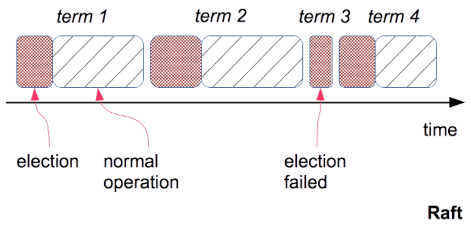


Figure 2. Phases in Zab and Raft

Raft [7] is a recent consensus protocol that was designed to enhance understandability of the Paxos protocol while maintaining its correctness and performance.

As shown in Figure 2, both Zab and Raft implement a dedicated phase to elect a distinguished primary leader. Both Zab and Raft decompose the consensus problem into independent subproblems: leader election, log replication, and safety and liveness. The distinguished primary leader approach provides a simpler foundation for building practical systems. A leader change is denoted by *epoch* $e \in \mathbb{N}$ and *term* $t \in \mathbb{N}$ in Zab and Raft, respectively. A new leader election will increase e or t, so all non-faulty nodes only accept the leader with higher epoch or term number. After leader election, in the normal operation, the leader proposes and serializes client's operations in total order at each replica.

In all Paxos protocols, every chosen value (i.e., proposed client operation) is a log entry, and each entry identifier z has two components denoted as *slot* and *ballot* number in Paxos, epoch and counter $\langle e, c\rangle$ in Zab, and as term and index in Raft. When the leader broadcasts a proposal for the current entry, a quorum of followers vote for the proposals and apply the corresponding operations after the leader commits. All Paxos protocols guarantee ordering, namely when command $\langle z, c\rangle$ is delivered, all commands $\langle z', c\rangle$ where $z' < z$ is delivered first, despite crashes of the leaders.

2.2. Differences among Paxos protocols

Leader election. Zab and Raft protocols differ from Paxos as they divide execution into phases (called epochs in Zab and terms in Raft), as shown in Figure 2 (redrawn from [7]). Each epoch begins with a new election, goes into the broadcast phase and ends with a leader failure. The phases are sequential because of the additional safety properties are provided by the *isLeader* predicate. The *isLeader()* predicate guarantees a single distinguished leader. That is, in Zab and Raft there can be at most one leader at any time. In

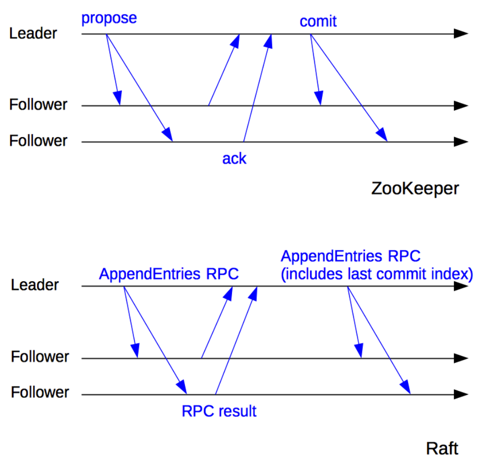


Figure 3. Messaging in Zab and Raft

contrast, Paxos does not provide this strong leader property. Since Paxos lacks a separate leader election phase, it can have multiple leaders coexisting, however it still ensures safety thanks to the ballot numbers and quorum concepts.

Zab algorithm has three phases and each node can be in one of these three phases at any given time. Discovery phase is where the leader election occurs, over current known configuration of the quorum. A process can only be elected if it has a higher epoch or if the epoch is same a higher committed transaction id. In the synchronization phase, the new leader synchronizes its initial history of previous epoch with all followers. The leader proceeds for the broadcast phase only after a quorum of followers acknowledged that they are synchronized with the leader. The broadcast phase is the normal operation mode, and the leader keeps proposing new client requests until it fails.

In contrast to Zab, there is no distinct synchronization phase in Raft: the leader stays synchronized with each follower in the normal operation phase by comparing the log index and term value of each entry. As shown in figure 2, lack of distinct synchronization phase simplifies Raft algorithmic states, but may result in longer recovery time in practice.

Communication with the replicas. Zab adopts a messaging model, where each update requires at least three messages: proposal, ack and commit as shown in Figure 3. In contrast Raft relies on an underlying RPC system. Raft also aims to minimize the state space and RPC types required in the protocol by reusing a few techniques repeatedly. For example, the AppendEntries RPCs are initiated by leader to both replicate log and perform heartbeat.

Dynamic reconfiguration. The original Paxos was limited as it assumed a static ensemble $2f + 1$ that can crash and recover but cannot expand or shrink. The ability to dynamically reconfigure the membership of consensus ensemble on the fly and while preserving data consistency provides an important extension for Paxos protocols. Dynamic reconfiguration in all Paxos protocols share the following basic approach. A client proposes a special *reconfig* command with a new configuration C_{new} which is decided in a log entry just like any other command. To ensure safety, C_{new} cannot be activated immediately and the configuration changes must go through two phases. Due to the different nature of the protocol, the reconfiguration algorithm differs in each Paxos protocol.

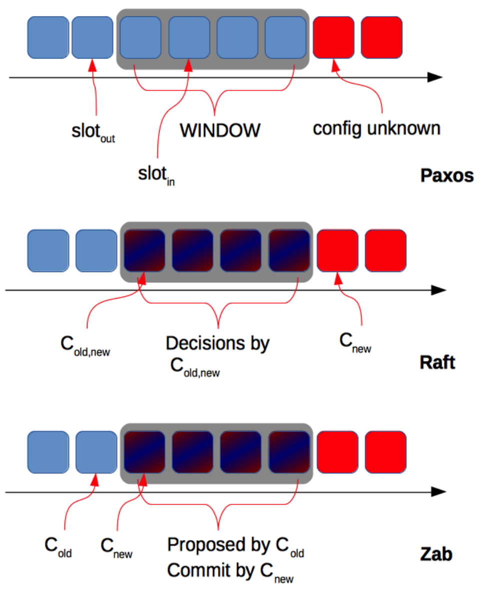


Figure 4. Dynamic reconfiguration in Paxos protocols

Dynamic reconfiguration approach in Paxos [14] introduces uncertainty of a slot's configuration, therefore, it imposes a bound on the concurrent processing of all commands. A process can only propose commands for slots with known configuration, $\forall \rho : \rho.slot_{in} < \rho.slot_{out} +$ WINDOW, as shown in Figure 4.

By exploiting primary order property provided by Zab and Raft, both protocols are able to implement their reconfiguration algorithms without limitations to normal operations or external services. Both Zab and Raft include a **pre-phase** where the new processes in C_{new} join the cluster as a non-voting members so that the leader in C_{old} could initialize their states by transferring currently committed prefix of updates. Once the new processes have caught up with the leader, the reconfiguration can proceed to schedule. The difference is that in Zab, the time between C_{new} proposed and committed, any commands received after reconfig is only scheduled but will not commit, as they are the responsibility of C_{new}. However, in Raft, the time interval is decided by quorum of $C_{old,new}$.

2.3. Extensions to the Paxos protocols

In the SMR approach, to further improve efficiency, under special cases a partial ordering of command sequence can be used instead of total ordering of chosen values: e.g., two *commutative* commands can be executed in any order since they produce the same state as the result. The resultant protocol called Generalized Paxos [11] is an extension of

TABLE 1. Latency and Consistency expectation

	λ	Read		Write		P(consistency)	0.999
		Median	99.9th percentile	Median	99.9th percentile		
N = 3	1.5	1.11 ms	6.27 ms	1.98 ms	6.92 ms	94.65%	2.5 ms
N = 5	1.5	1.11 ms	6.27 ms	2.1 ms	7.16 ms	94.79%	3 ms
N = 7	1.5	1.11 ms	6.27 ms	2.18 ms	7.27 ms	95.05%	3.5 ms
N = 9	1.5	1.11 ms	6.27 ms	2.19 ms	7.43 ms	95.52%	3.5 ms
N = 3	0.1	16.65 ms	89.7 ms	29.57 ms	110.15 ms	94.2%	30 ms
N = 5	0.1	16.65 ms	89.7 ms	31.78 ms	107.04 ms	95.03%	30 ms
N = 7	0.1	16.65 ms	89.7 ms	32.24 ms	107.42 ms	95.48%	35 ms
N = 9	0.1	16.65 ms	89.7 ms	32.76 ms	112.47 ms	95.51%	35 ms

Fast Paxos [10], and allows acceptors to vote for independent commands. Similarly EPaxos [13] is able to achieve lower latency because it allows nodes to commit conflict-free commands by checking the command dependency list. However, EPaxos adds significant complexity and extra effort to resolve the conflict if concurrent commands do not commute. In addition, from an engineer's perspective, the sketch algorithm descriptions in the literature are often underspecified, and lead to divergent interpretations and implementations. Building such system using Paxos consensus algorithm proved to be non-trivial [12].

Paxos users often face a trade-off between read latency and staleness. Although each write is serialized and synchronously replicated, such a write may only be applied to a quorum of replicas. Thus, another client reading at a replica where this write has not been replicated may still see the old version. Since the leader is the only process guaranteed to participate in all write quorums, stale reads can be avoided by reading from current leader with a consequent increase in latency.

The probability of stale reads is a function of the network. Inspired by the probabilistically bounded staleness (PBS[1]) [17], we modified the model to estimate the Zab/Raft-like primary ordered consensus protocol's read/write latency and $P(consistency)$. Our model adopts 6 different communication delays, *CR* (Client-Replica), *P* (Proposal), *A* (Ack), *C* (Commit), *R* (Read), and *S* (Response), in order to investigate possible read and write message reordering and resultant stale-reads. The simulation uses Monte Carlo method with each event drawn from a predefined distribution. For simplicity, we assume each channel latency fits in an exponential distribution characterized by λ and we assume message delays are symmetric, $CR = P = A = C = R = S = \lambda$ ($\lambda = 1.5$ means 0.66ms). In Table 1, $P(consistency)$ show the probability of consistent read of the last written version in different ensemble sizes, given that the clients read from the first responding replica.

3. Paxos Systems

In this section we compare and contrast three popular Paxos systems, ZooKeeper, Chubby, and etcd, and examine the features these systems provide to the clients. We also discuss proper usage criteria for these Paxos systems, a topic which has not received sufficient coverage.

1. http://pbs.cs.berkeley.edu/#demo

TABLE 2. Features of Paxos systems

	Systems		
Feature	**Chubby**	**ZooKeeper**	**etcd**
Filesystem API	✓	✓	✓
Watches	✓	✓	✓
Ephemeral Storage	✓	✓	✓
Local Reads	✓	✓	
Dynamic Reconfiguration		✓	✓
Observers	✓	✓	✓
Autoincremented Keys		✓	✓
Hidden Data			✓
Weighted Replicas		✓	

3.1. Similarities among Paxos systems

Chubby [2], ZooKeeper [3] and etcd [18] are consensus services designed specifically for loosely-coupled distributed systems. Chubby, originally a lock service used in Google productins, is the first service to provide consensus through a service, with ZooKeeper and others arriving later.

All three services hide the replicated state machine and log abstractions under a small data-store with filesystem-like API. Filesystem interface was chosen for its familiarity to the developers, reducing the learning curve. The interface enables developers to reason about consensus and coordination as if they were working with a filesystem on a local machine. ZooKeeper calls all data objects in the hierarchical structure as *znodes*. Each znode can act as both the file for storage and as a parent for other stored items.

An important feature common to these systems is the ability to set *watches* on the data objects allowing the clients to receive timely notifications of changes without requiring polling. Typically, these systems implement one-time watches, meaning that a system notifies the client only for the first change of the object. If a client application wants to continue receiving the updates, it must reinstitute the watch in the system.

All three systems support temporary or ephemeral storage that persists only while the client is alive and sending heartbeat messages. This mechanism allows the clients to use Paxos systems for failure detection and triggering reconfiguration upon addition or removal of clients in the application.

Both ZooKeeper and etcd provide the clients with the ability to create auto-incremented keys for the data items stored in a directory. This feature simplifies implementation of certain counting data-structures, such as queues.

All three Paxos systems adopt *observer* servers. Observer is a non-voting replica of an ensemble that learns the entire committed log but does not belong to a quorum set. Observers can serve reads with a consistent view of some point in the recent past. This way, observers improve system scalability and help disseminate data over a wide geographic area without impacting the write latency.

3.2. Differences among Paxos systems

Despite serving the same purpose, Chubby, ZooKeeper, and etcd have many differences both in terms of the feature sets and internal implementations. Chubby uses the Multi-Paxos algorithm to achieve linearizability, while Zab lies at the heart of ZooKeeper and provides not only linearizability, but also FIFO order for client requests, enabling the developers to build complex coordination primitives with ease. Raft is the consensus protocol behind the etcd system.

Unlike ZooKeeper and Chubby, etcd is stateless with respect to its clients. In other words, etcd system is oblivious to any clients using it and no client information is retained in the service. This allows etcd to use REST API as its communication interface, obviating the need for a special client software or library. Since etcd is stateless, it implements certain features very differently than ZooKeeper and Chubby. For instance, watches require a persistent connection with the client using HTTP long polling technique, while ephemeral storage requires clients to manually set time-to-live (TTL) on the data objects and update the TTL periodically.

Hidden data items is another interesting feature of etcd inspired by hidden files in conventional filesystems. With hidden object ability clients can write items that will not be listed by the system when requesting a file or directory listing, thus only clients who know the exact name of the data object are able to access it.

The original Zab algorithm, as well as many other consensus algorithms, only concern full replicas which contain all the write-ahead log and state machine entity and involve equally in voting process and in serving read requests. ZooKeeper extends Zab and introduces weighted replicas which can be assigned with different voting weights in the quorum, so the majority condition is converted to greater than half of the total weights. Replicas that have zero weight are discarded and not considered when forming quorums.

Table 2 summarizes the main similarities and differences among these Paxos systems. As the table shows, these systems provide expressive and comparable APIs to the clients, allowing the developers to utilize them for distributed coordination in many different use cases.

3.3. Proper use criteria for Paxos systems

The relative ease-of-use and generality of the client interfaces of these Paxos systems allow for great flexibility that sometimes leads to misuse. In order to prevent improper and inefficient utilization of Paxos systems, we propose the following criteria for proper use of Paxos systems. Violating any one of these criteria does not automatically disqualify the application from using a Paxos system, rather it calls for a more thorough examination of goals to be achieved and whether a better solution exists.

1) Paxos system should not be in the performance critical path of the application. Consensus is an expensive task, therefore a good use case tries to minimize performance degradation by keeping the Paxos system use away from the performance critical and frequently utilized path of the application.

2) Frequency of write operations to the Paxos system should be kept low. The first rule is especially important for write operations due to the costs associated with achieving consensus and consistently replicating data across the system.

3) Amount of data maintained in the Paxos system should be kept small. Systems like ZooKeeper and etcd are not designed as general-purpose data storage, so a proper adoption of these systems would keep the amount of data maintained/accumulated in the Paxos systems to a minimum. Preferably only small metadata should be stored/maintained in the Paxos system.

4) Application adopting the Paxos system should really require strong consistency. Some applications may erroneously adopt a Paxos system when the strong consistency level provided by the Paxos system may not be necessary for the application. The Paxos systems linearize all write operations, and such linearizability incurs performance degradation and must be avoided unless it is necessary for the application.

5) Application adopting the Paxos system should not be distributed over the Wide Area Network (WAN). In Paxos systems the leader and replicas are commonly located in the same datacenter so that the roundtrip times do not affect the performance very badly. Putting the replicas and application clients far away from the leader, e.g., across multiple datacenters and continents, would significantly degrade the performance.

6) The API abstraction should be fit the goal. Paxos systems provide filesystem-like API to the clients, but such an abstraction may not be suitable for all tasks. In some cases, such as the server replication discussed in the next section, dealing with the filesystem abstraction can be too cumbersome and error-prone that a different approach would serve better.

4. Paxos Use Patterns

In this section, we categorize and characterize the most common Paxos use patterns in the datacenter and cloud computing applications.

Server Replication (SR). Server replication via the state machine replication (SMR) approach is a canonical application for Paxos protocol. The SMR requires a state machine to be deterministic: multiple copies of the state machine begin in the start state and receive the same inputs in the same order and each replica will arrive at the same state having generated the same outputs. Paxos is used for serializing and replicating the operations to all nodes in order to ensure that states of the machines are identical and the same sequence of operations is applied. This use case becomes the most practical when the input operations causing the state changes are small compared to the large state maintained at each node. Pure Paxos, Zab and Raft protocols are better suited to achieve server replication than Paxos systems, since using Paxos systems like ZooKeeper introduces additional overhead of dealing with the filesystem API and maintaining a separate consensus system cluster in addition to the server to be replicated and its replicas.

Log Replication (LR). The objective of log replication is different than that of server replication. Log replication is applied in data integration systems that use the log abstraction to duplicate data across different nodes, while server replication is used in SMR to make copies of the server state. Since Paxos systems such as ZooKeeper have limited storage, they are not typically suitable for the data-centric/intensive task of log replication. Systems like BookKeeper and Kafka are a better fit for this use case as they remove consensus out of the critical path of data replication, and employ Paxos only for maintaining the configuration of a system.

Synchronization Service (SS). An important application of consensus is to provide synchronization. Traditionally, concurrent access to the shared data is controlled by some form of mutual exclusion through locks. However such approach requires applications to build their own failure detection and recovery mechanism, and a slow or blocked process can harm the overall performance. When the consensus protocol/system is decoupled from the application, the application not only gains fault tolerance of the shared data, but also achieves wait-free concurrent data access with guaranteed consistency.

Google Chubby [2] was originally designed to provide a distributed lock service intended for coarse-grained synchronization of activities through Multi-Paxos in its heart, but it found wider use in other cases such as name service and repository of configuration data. ZooKeeper [3] provides simple code recipes for exclusive locks, fair locks and shared locks. Since both Chubby and ZooKeeper expose a filesystem interface where each data node is accessed by a hierarchical path, the locks are represented by data nodes created by clients. The data nodes used as locks are usually *ephemeral* nodes which can be deleted explicitly or automatically by the system when a session that creates them terminates due to a failure. Since locks do not maintain other metadata within data nodes, all operations on locks are lightweight.

Barrier Orchestration (BO). Large-scale graph processing systems based on BSP (Bulk Synchronous Parallel) model like Google Pregel [19], Apache Giraph [20] and Apache Hama [21] use Paxos systems for coordination between computing processes. Since the graph systems process data in an iterative manner, a *double barrier* is used to synchronize the beginning and the end of each computation iteration across all nodes. Barrier thresholds may be reconfigured during each iteration as the number of units involved in the computation changes.

Of course, violating the proper use criteria of Paxos systems for this task can cause problems. Facebook Giraph paper [22] discusses the following example for this misuse pattern. Large-scale graph processing systems uses *aggregators* to provide shared state across vertices. Each vertex can send a value to an aggregator in superstep S, a master combines those values using a reduction operator, and the resulting value is made available to all vertices in superstep $S+1$. In early version of Giraph, aggregators were implemented using ZooKeeper, violating criteria 3. So this does not scale in the case of executing k-means clustering with millions of centroids, because it requires an equal amount of aggregator and tens of gigabytes of aggregator data coming to ZooKeeper from all vertices. To solve this issue, Giraph bypassed ZooKeeper and implemented shared aggregators, each randomly running on one of the workers. This solution lost durability/fault-tolerance of ZooKeeper, but achieved fast performance and scalability.

Configuration Management. Most Paxos systems provide the ability to store arbitrary data by exposing a filesystem or key-value abstraction to the systems. This gives the applications access to durable and consistent storage for small data items that can be used to maintain configuration metadata like connection details or feature flags. These metadata can be *watched* for changes, allowing applications to reconfigure themselves when configuration parameters are modified. *Leader election (LE), group membership (GM), service discovery (SD), and metadata management (MM)* are main use cases under configuration management, as they are important for cluster management in cloud computing systems.

Message Queues (Q). A common misuse pattern is to use the Paxos system to maintain a distributed queue, such as a publisher-subscriber message queue or a producer-consumer queue. In ZooKeeper, with the use of watchers, one can implement a message queue by letting all clients interested in a certain topic register a watch on the topic znode, and messages will be broadcast to all the clients by writing to that znode. Unfortunately, queues in production can contain many thousands of messages resulting in a large volume of write operations and potentially huge amounts of data going through the Paxos system, violating criteria 3. Moreover, in this case the Paxos system stands in the critical path of every queue operation (violating criteria 1 and 2), and this decreases the performance even further. Apache BookKeeper and Kafka projects properly address the message queue use case. Both of these distributed pub-sub messaging systems rely on ZooKeeper to manage the metadata aspect of the replication (partition information and memberships), and handle replicating the actual data separately. By removing consensus out of the critical path of data replication and using it only for configuration management, both systems achieve good throughput and scalable message replication.

Table 3 summarized and evaluates the Paxos use patterns in terms of usage criteria discussed in section 3. If a using pattern access the consensus system per-data operation, we consider it as high frequency, otherwise it is low frequency if application only access consensus system on rare events.

TABLE 3. EVALUATION OF PATTERNS

Patterns	Frequency	Data Volume	API	Paxos system use	Better substitute
SR	High	Large	Hard	Bad	Replication Protocol
LR	High	Medium	Hard	Bad	Replication Protocol
SS	Low	Small	Hard	OK	Distributed Locks
BO	Low	Depends	Easy	OK	
SD	Low	Small	Easy	Good	
GM	Low	Small	Easy	Good	
LE	Low	Small	Easy	Good	
MM	Medium	Medium	Easy	Good	Distributed Datastore
Q	High	Large / Medium	Hard	Bad	Kafka

Data volume describes the amount of data required by the pattern on average. The file-system API abstraction provided by most of the consensus system can be easy to utilize for some tasks, but hard to get it right without recipe for others. Overall, a using pattern is considered as a good or bad use for Paxos systems. Finally, a few substitutes of Paxos systems for particular task are also listed for reference.

5. Paxos Use in Production Systems

In this section, we examine the infrastructure stack of Google and Facebook, and Apache Foundation's top-level projects to determine where Paxos protocols and Paxos systems are used and what they are used for.

5.1. Google Stack

We reviewed Google's publications to gather information about the Google infrastructure stack. Each system related literature is the sole source of our analysis of 18 projects, among which 7 projects implement Paxos or directly depend on the consensus system.

At the bottom of the stack is coordination and cluster management systems. Borg [23] is Google's cluster manager that runs hundreds of thousands of jobs from many applications in multiple clusters. Borg uses Chubby as a consistent, highly available Paxos store to hold the metadata for each submitted job and running task. Borg uses Paxos in several places: (1) One usage is to write hostname and port of each task in Chubby to provide naming service; (2) Borgmaster implements Paxos for leader election and master state replication; (3) Borg uses Paxos as a queue of new submitted jobs which helps scheduling. Kubernetes [24] is a recent open source project from Google for Linux container cluster management. Kubernetes adopts etcd to keep state, such as resource server membership and cluster metadata.

The second layer of the stack contains several data storage components. The most important storage service is the Google File System (GFS) [25] (and the successor called Colossus [26]). GFS uses Chubby for master election, and master state replication. Bigtable [27] is Google's distributed storage system for structured data. It heavily relies on Chubby for a variety of tasks: (1) to ensure there is at most one active master at any time (leader election); (2) to store the bootstrap location of Bigtable data (metadata management); (3) to discover tablet servers and finalize tablet server deaths (group membership); (4) to store Bigtable schemas (configuration management). Megastore [28] is a cross datacenter database that provides ACID semantics within fine-grained and replicated partitions of data. Megastore is also the largest system deployed that uses Paxos to replicate primary user data across datacenters on every write. It extends Paxos to synchronously replicate multiple write-ahead logs, each governing its own partition of the data set. Google Spanner [26] is a globally distributed, multiversion database that shards data across many sets of Paxos state machines. Each spanserver implements a single Paxos state machine on top of each tablet for replication, and a set of replicas is collectively a Paxos group.

5.2. Facebook Stack

Facebook takes advantage of many open source projects to build a foundation of its distributed architecture. Storage layer is represented by both traditional relational systems and NoSQL databases. HDFS [29] serves as the basis for supporting large distributed key-value stores such as HBase [30]. HDFS is part of the Apache Hadoop project and uses ZooKeeper for all its coordination needs: resource manager state replication, leader election and configuration metadata replication. HBase is built on top of HDFS and utilizes ZooKeeper for managing configuration metadata, region server replication and shared resources concurrency control. Cassandra [31] is another distributed data store that relies on ZooKeeper for tasks like leader election, configuration metadata management and service discovery. MySQL is used as a relational backbone at Facebook with many higher level storage systems, such as Haystack and f4, interacting with it.

Data processing layer utilize the resources of the storage architecture, making Facebook rely on systems that integrate well with HDFS and HBase. Hadoop operates on top of HDFS and provides MapReduce infrastructure to various Facebook components. Hive [32] is an open source software created for data warehousing on top of Hadoop and HBase. It uses ZooKeeper as its consensus system for implementing metadata storage and lock service.

Facebook uses a modified and optimized version of ZooKeeper, called Zeus [33]. Currently Zeus is adopted in the Configerator project, an internal tool for managing configuration parameters of production systems. Zeus's role in the process is serializing and replicating configuration changes across a large number of servers located throughout Facebook's datacenters. It is likely that over time more of the company's systems are going to use Zeus.

5.3. Apache Projects

Apache foundation has many distributed systems that require a consensus algorithm for a variety of reasons. Zookeeper, being an Apache project itself, can be seen as the de facto coordination system for other applications under the Apache umbrella. Currently, about 31% of Apache projects in BigData, Cloud and Databse categories[2] directly use ZooKeeper, while many more applications relying on other projects that depend on Apache's consensus system. This makes ZooKeeper an integral part of open source distributed systems infrastructure. Below we briefly mention some of the more prominent systems adopting ZooKeeper as a consesnus service.

Apache Accumulo [34] is a distributed key-value store based on the Google's BigTable design. This project is similar to Apache HBase and uses Zookeeper in some of the identical ways: shared resource mutual exclusion, configuration metadata management and tasks serialization.

BookKeeper [5] project implements a durable replicated distributed log mechanism. It employs ZooKeeper for storing log metadata and keeping track of configuration changes, such as server failures or additions. Hedwig is a publish-subscribe message delivery system developed on top of BookKeeper. The system consists of a set of hubs that are responsible for handling various topics. Hedwig uses ZooKeeper to write hub metadata, such as topics served by each hub. Apache Kafka is another publish-subscribe messaging software, unlike Hedwig, it does not rely on BookKeeper, however it uses ZooKeeper for a number of tasks: keeping track of removal or additions of nodes, coordinating rebalancing of resources when system configuration changes and keeping track of consumed messages [6].

Apache Solr [35] search engine uses ZooKeeper for its distributed sharded deployments. Among other things, ZooKeeper is used for leader election, distributed data-structures such as queues and maps and storing various configuration parameters.

5.4. Evaluation of Paxos use

In order to evaluate the way Paxos protocols and Paxos systems are adopted by Google's and Facebook's software stacks and Apache top-level projects, we have classified the usage patterns into nine broad categories: server replication (SR), log replication (LR), synchronization service (SS), barrier orchestration (BO), service discovery (SD), group membership (GM), leader election (LE), metadata management (MM) and distributed queues (Q). The majority of the investigated systems have used consensus for tasks in more than one category. Table 4 summarizes various usage patterns observed in different systems.

Figure 5 shows the frequency of consensus systems being used for each of the task categories. As can be seen, metadata management stands for 27% of all usages and is the most popular adoption scenario of consensus systems, closely followed by the leader election. It is worth to keep in mind that metadata management is a rather broad category that encompasses many usage arrangements by the end systems including application configuration management or managing state of some internal objects or components. Synchronization service is another popular application for

2. https://projects.apache.org/projects.html?category

TABLE 4. PATTERNS OF PAXOS USE IN PROJECTS

		Usage Patterns								
Project	**Consensus System**	**SR**	**LR**	**SS**	**BO**	**SD**	**GM**	**LE**	**MM**	**Q**
GFS	Chubby			✓				✓	✓	
Borg	Chubby/Paxos	✓				✓		✓		
Kubernetes	etcd						✓		✓	
Megastore	Paxos		✓							
Spanner	Paxos	✓								
Bigtable	Chubby						✓	✓	✓	
Hadoop/HDFS	ZooKeeper	✓						✓		
HBase	ZooKeeper	✓		✓			✓		✓	
Hive	ZooKeeper			✓					✓	
Configerator	Zeus								✓	
Cassandra	ZooKeeper					✓		✓	✓	
Accumulo	ZooKeeper		✓	✓					✓	
BookKeeper	ZooKeeper						✓		✓	
Hedwig	ZooKeeper						✓		✓	
Kafka	ZooKeeper						✓	✓	✓	
Solr	ZooKeeper							✓	✓	✓
Giraph	ZooKeeper		✓		✓				✓	
Hama	ZooKeeper				✓					
Mesos	ZooKeeper							✓		
CoreOS	etcd					✓				
OpenStack	ZooKeeper					✓				
Neo4j	ZooKeeper			✓				✓		

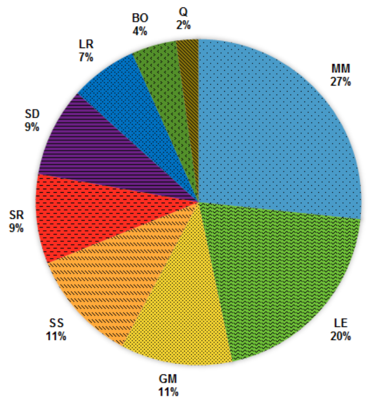


Figure 5. Relative Frequency of Consensus Systems Usage for Various Tasks

consensus protocols, since they greatly simplify the implementation of the distributed locks. Distributed queues, despite being one of the suggested ZooKeeper recipes, are used least frequently. This can be attributed to the fact that using consensus for managing large queues may negatively impact performance.

6. Concluding Remarks

We compared/contrasted popular Paxos-protocols and Paxos-systems and investigated how they are adopted by production systems at Google, Facebook, and other cloud computing domains. We find that while Paxos systems (and in particular ZooKeeper) are popularly adopted for coordination, they are often over-used and misused. Paxos systems are suitable as a durable metadata management service. However, the metadata should remain small in size, should not accumulate in size over time, and should not get updated frequently. When Paxos systems are improperly used, they constitute the bottleneck in performance and cause scalability problems. A major limitation on Paxos systems performance and scalability is that they fail to provide a way to denote/process commutative operations via the Generalized Paxos protocol. In other words, they force every operation to be serialized through a single master.

We conclude by identifying tradeoffs for coordination services and point out promising directions for achieving more scalable coordination in the clouds.

Trading off strong-consistency for fast performance. If strong-consistency is not the primary focus and requirement of the application, instead of a Paxos-protocol/Paxos-system solution, an eventually-consistent solution replicated via optimistic replication techniques [36] can be used for fast performance, often with good consistency guarantees. For example, in the Facebook's TAO system, which is built over a 2-level Memcached architecture, only 0.0004% of reads violate linearizability [37]. Adopting Paxos would have solved those linearizability violations, but providing better performance is more favorable than eliminating a tiny fraction of bad reads in the context of TAO usage.

Trading off performance for more expressivity. ZooKeeper provides a filesystem API as the coordination interface. This has shortcomings and is unnatural for many tasks. Tango [38] provides a richer interface, by working at a lower-layer. Tango uses Paxos to maintain a consistent and durable log of operations. Then Tango clients use this log to view-materialize different interfaces to serve. One Tango client can serve a B+tree interface by reading and materializing the current state from the log, another client a queue interface, yet another a filesystem interface. Unfortunately this requires the Tango clients to forward each read/write update towards getting a Paxos read/commit from the log, which degrades the performance.

Trading off expressivity with fast performance and scalability. In Paxos systems, the performance of update operations is a concern and can constitute bottlenecks. By using a less expressive interface, it is possible to improve this performance. If a plain distributed key-value store interface is sufficient for an application, a consistent-replication key-value store can be used based on chain replication [39]. Chain replication uses Paxos only for managing the configuration of the replicas, and replication is achieved with very good throughput without requiring Paxos commit for each operation. Furthermore, in this solution reads can be answered consistently by any replica. However, this solution forgoes the multi-item coordination in the Paxos systems API, since its API is restricted to put and get with key operations.

Along the same lines, an in-memory transactional key-value store can also be considered as a fast and scalable alternative. Sinfonia [40] provides transactions over an in-memory distributed key-value store. These transactions are minitransactions that are less expressive than general transactions, but they can be executed in one roundtrip. Recently RAMCloud system [41] showed how to implement the Sinfonia transactions in a fast and durable manner using a write-ahead replicated log structure. Since Paxos systems are limited by the size of one node's memory, RAMCloud/Sinfonia provides a scalable and fast alternative with minitransactions over multiple-records, with a slightly less expressive API.

References

[1] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[2] M. Burrows, "The chubby lock service for loosely-coupled distributed systems." in *OSDI*. USENIX Association, 2006, pp. 335–350.

[3] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, vol. 10, 2010.

[4] F. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Dependable Systems & Networks (DSN)*. IEEE, 2011, pp. 245–256.

[5] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," *ACM SIGOPS Operating Systems Review*, vol. 47, no. 1, pp. 9–15, 2013.

[6] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing." NetDB, 2011.

[7] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319.

[8] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," vol. 22, no. 4, pp. 299–319, Dec. 1990.

[9] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[10] ——, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[11] ——, "Generalized consensus and paxos," Technical Report MSR-TR-2005-33, Microsoft Research, Tech. Rep., 2005.

[12] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007, pp. 398–407.

[13] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 358–372.

[14] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, p. 42, 2015.

[15] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM (JACM)*, vol. 34, no. 1, pp. 77–97, 1987.

[16] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, 1996.

[17] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 776–787, 2012.

[18] "etcd," https://coreos.com/etcd/.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[20] "Apache giraph project," http://incubator.apache.org/giraph/.

[21] “Apache hama project,” http://hama.apache.org/.

[22] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: graph processing at facebook-scale,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[23] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[24] “Google kubernetes project,” http://kubernetes.io/.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *ACM SIGOPS Operating Systems Review*, vol. 37/5. ACM, 2003, pp. 29–43.

[26] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s globally-distributed database,” *Proceedings of OSDI*, 2012.

[27] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[28] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh, “Megastore: Providing scalable, highly available storage for interactive services,” *CIDR*, pp. 223–234, 2011.

[29] “Apache hadoop project,” http://hadoop.apache.org/.

[30] L. George, *HBase: The Definitive Guide*, 1st ed. O’Reilly Media, 2011. [Online]. Available: http://www.amazon.de/HBase-Definitive-Guide-Lars-George/dp/1449396100/ref=sr_1_1?ie=UTF8&qid=1317281653&sr=8-1

[31] A. Lakshman and P. Malik, “Cassandra: Structured storage system on a p2p network,” in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, ser. PODC ’09, 2009, pp. 5–5.

[32] “Apache hive project,” http://hive.apache.org/.

[33] C. Tang, T. Kooburat, P. Venkatachalam, A. Chandler, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, “Holistic configuration management at Facebook,” *Symposium on Operating Systems Principles (SOSP)*, pp. 328–343, 2015. [Online]. Available: http://sigops.org/sosp/sosp15/current/2015-Monterey/printable/008-tang.pdf

[34] “Apache accumulo project,” http://accumulo.apache.org/.

[35] “Apache solr,” http://lucene.apache.org/solr/.

[36] Y. Saito and M. Shapiro, “Optimistic replication,” *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005.

[37] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, “Existential consistency: Measuring and understanding consistency at Facebook,” *Symposium on Operating Systems Principles (SOSP)*, pp. 295–310, 2015. [Online]. Available: http://sigops.org/sosp/sosp15/current/2015-Monterey/printable/240-lu.pdf

[38] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, “Tango: Distributed data structures over a shared log,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 325–340.

[39] R. van Renesse and F. B. Schneider, “Chain replication for supporting high throughput and availability,” in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, vol. 6, 2004.

[40] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, “Sinfonia: a new paradigm for building scalable distributed systems,” in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 159–174.

[41] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, “Implementing linearizability at large scale and low latency,” in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 71–86.