Dynamic Reconfiguration of Primary/Backup Clusters

Alexander Shraer *Benjamin Reed*
Yahoo! Research
{*shralex, breed*}*@yahoo-inc.com*

Dahlia Malkhi
Microsoft Research
dalia@microsoft.com

Flavio Junqueira
Yahoo! Research
fpj@yahoo-inc.com

Abstract

Dynamically changing (reconfiguring) the membership of a replicated distributed system while preserving data consistency and system availability is a challenging problem. In this paper, we show that reconfiguration can be simplified by taking advantage of certain properties commonly provided by Primary/Backup systems. We describe a new reconfiguration protocol, recently implemented in Apache Zookeeper. It fully automates configuration changes and minimizes any interruption in service to clients while maintaining data consistency. By leveraging the properties already provided by Zookeeper our protocol is considerably simpler than state of the art.

1 Introduction

The ability to reconfigure systems is critical to cope with the dynamics of deployed applications. Servers permanently crash or become obsolete, user load fluctuates over time, new features impose different constraints; these are all reasons to reconfigure an application to use a different group of servers, and to shift roles and balance within a service. We refer to this ability of a system to dynamically adapt to a changing set of machines or processes as *elasticity*.

Cloud computing has intensified the need for elastic long lived distributed systems. For example, some applications such as sports and shopping are seasonal with heavy workload bursts during championship games or peak shopping days. Such workloads mean that elasticity is not a matter of slowly growing a cluster; it may mean that a cluster grows by an order of magnitude only to shrink by the same order of magnitude shortly after.

Unfortunately, at the back-end of today's cloud services, one frequently finds a coordination service which itself is not elastic, such as ZooKeeper [12]. Companies such as Facebook, LinkedIn, Netflix, Twitter, Yahoo!, and many others, use Zookeeper to track failures and configuration changes of distributed applications; application developers just need to react to events sent to them by the coordination service. However, Zookeeper users have been asking repeatedly since 2008 to facilitate reconfiguration of the service itself, and thus far, the road to elasticity has been error prone and hazardous: Presently, servers cannot be added to or removed from a running ZooKeeper cluster and similarly no other configuration parameter (such as server roles, network addresses and ports, or the quorum system) can be changed dynamically. A cluster can be taken down, reconfigured, and restarted, but (as we explain further in Section 2) this process is manually intensive, error prone and hard to execute correctly even for expert ZooKeeper users. Data corruption and split-brain[1] caused by misconfiguration of Zookeeper has happened in production[2]. In fact, configuration errors are a primary cause of failures in production systems [22]. Furthermore, service interruptions are currently inevitable during reconfigurations. These negative side-effects cause operators to avoid reconfigurations as much as possible. In fact, operators often prefer to over-provision a Zookeeper cluster than to reconfigure it with changing load. Over-provisioning (such as adding many more replicas) wastes resources and adds to the management overhead.

Our work provides a reconfiguration capability using ZooKeeper as our primary case-study. Our experience with ZooKeeper in production over the past years has lead us to the following requirements: first, ZooKeeper is a mature product that we do not want to destabilize; a solution to the dynamic reconfiguration problem should not require major changes, such as limiting concurrency or introducing additional system components. Second, as many Zookeeper-based systems are online, service disruptions during a reconfiguration should be minimized and happen only in rare circumstances. Third, even if there are failures during reconfiguration, data integrity, consistency or service availability must not be compromised, for instance, split-brain or loss of service due to partial configuration propagation should never be possible. Finally, we must support a vast number of clients who seamlessly migrate between configurations.

We use the Zookeeper service itself for reconfiguration, but we ruled out several straw-man approaches. First, we could have used an external coordination ser-

[1] In a *split-brain* scenario, servers form multiple groups, each independently processing client requests, hence causing contradictory state changes to occur.

[2] `http://search-hadoop.com/m/ek5ej2dOQsB`

vice, such as another ZooKeeper cluster, to coordinate the reconfiguration, but this would simply push the reconfiguration problems to another system and add extra management complexity. Another naïve solution would be to store configuration information as a replicated object in Zookeeper. When a ZooKeeper server instance comes up, it looks at its replica of the state to obtain the configuration from the designated object. While this solution is simple and elegant, it is prone to inconsistencies. Some replicas may be behind others, which means they could have different configuration states. In a fixed configuration, a consistent view of the system can be obtained by contacting a quorum of the servers. A reconfiguration, however, changes the set of servers and therefore guaranteeing a consistent view requires additional care. Consequently, reading the configuration from an object in Zookeeper may lead to unavailability or, even worse, corrupt data and split-brain.

Indeed, dynamically reconfiguring a replicated distributed system while preserving data consistency and system availability is a challenging problem. We found, however, that high-level properties provided by Zookeeper simplify this task. Specifically, ZooKeeper employs a primary/backup replication scheme where a single dynamically elected primary executes all operations that change the state of the service and broadcasts state-updates to backups. This method of operation requires that replicas apply state changes according to the order of primaries over time, guaranteeing a property called *primary order* [13]. Interestingly, this property is preserved by many other primary/backup systems, such as Chubby [5], GFS [8], Boxwood [19], PacificA [21] and Chain-Replication [20] (see Section 6). These systems, however, resort to an external service for reconfiguration. In this work we show that leveraging primary order simplifies reconfiguration. By exploiting primary order we are able to implement reconfiguration without using an external service and with minimal changes to ZooKeeper (in fact, reconfigurations are pipelined with other operations and treated similarly) while guaranteeing minimal disruption to the operation of a running system. We believe that our methods may be applied to efficiently reconfigure any Primary/Backup system satisfying primary order.

Previous reconfiguration approaches, such as the one proposed by Lamport [15], may violate primary order, cause service disruption during reconfiguration, as well as impose a bound on the concurrent processing of *all* operations due to uncertainty created by the ability to reconfigure (see Section 2). Similar to our approach, FRAPPE [4] imposes no such bounds, but requires rollback support and complex management of speculative execution paths, not needed in our solution.

Our reconfiguration protocol also encompasses the clients. As the service configuration changes, clients should stay connected to the service. Literature rarely mentions the client side of reconfiguration, usually stating the need for a name-service (such as DNS), which is of course necessary. However, its also crucial to re-balance client connections across new configuration servers and at the same time prevent unnecessary client migration which may overload servers, severely degrading performance. We propose a probabilistic load-balancing scheme to move as few clients as possible and still maintain an even distribution of clients across servers. When clients detect a change, they each apply a migration policy in a distributed fashion to decide whether to move to a new server, and if so, which server they should move to.

In summary, this paper makes the following contributions:

- An observation that primary order allows for simple and efficient dynamic reconfiguration.
- A new reconfiguration protocol for Primary/Backup replication systems preserving primary order. Unlike all previous reconfiguration protocols, our new algorithm does not limit concurrency, does not require client operations to be stopped during reconfigurations, and does not incur a complicated management overhead or any added complexity to normal client operation.
- A decentralized, client-driven protocol that rebalances client connections across servers in the presence of service reconfiguration. The protocol achieves a proven uniform distribution of clients across servers while minimizing client migration.
- Implementation of our reconfiguration and load-balancing protocols in Zookeeper (being contributed to Zookeeper codebase) and analysis of their performance.

2 Background

This section provides the necessary background on ZooKeeper, its way of implementing the primary/backup approach, and the challenges of reconfiguration.

Zookeeper. Zookeeper totally orders all writes to its database. In addition, to enable some of the most common use-cases, it executes requests of every client in FIFO order. Zookeeper uses a primary/backup scheme in which the primary executes all write operations and broadcasts state changes to the backups using an atomic broadcast protocol called Zab [13]. ZooKeeper replicas process read requests locally. Figure 1 shows a write operation received by a primary. The primary executes the write and broadcasts a state change that corresponds to the result of the execution to the backups. Zab uses quo-

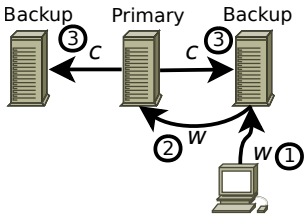

Figure 1: The processing of a write request by a primary. 1. a backup receives the request, w; 2. the backup forwards w to the primary; 3. the primary broadcasts the new state change, c, that resulted from the execution of w.

rums to commit state changes. As long as a quorum of servers are available, Zab can broadcast messages and ZooKeeper remains available.

Primary/Backup replication a la Zab. Zab is very similar to Paxos [15], with one crucial difference – the agreement is reached on full history prefixes rather than on individual operations. This difference allows Zab to preserve primary order, which may be violated by Paxos (as shown in [13]). We now present an overview of the protocol executed by the primary. Note that the protocol in this section is abstract and excludes many details irrelevant to this paper. The protocol has two parts, each involving an interaction with a quorum: A startup procedure, which is performed only once, and through which a new leader[3] determines the latest state of the system[4]; and a steady-state procedure for committing updates, which is executed in a loop.

Zab refers to the period of time that a leader is active as an *epoch*. Because there is only one leader active at a time, these epochs form a sequence, and each new epoch can be assigned a monotonically increasing integer called the epoch number. Specifically, each backup maintains two epoch identifiers: the highest epoch that it received from any primary in a startup phase, $e_{prepare}$, and the highest epoch of a primary whose history it adopted in steady-state, e_{accept}.

Startup: A candidate leader b chooses a *unique* epoch e and sends a PREPARE message to the backups. A backup receiving a PREPARE message acts as follows:

- If $e \geq e_{prepare}$, it records the newly seen epoch by setting $e_{prepare}$ to e and then responds with an ACK message back to the candidate.
- The ACK includes a history prefix H consisting of state-updates previously acknowledged by the backup, as well as the epoch e_{accept}.

When b collects a quorum of ACK messages, it adopts a history H received with the highest e_{accept} value, breaking ties by preferring a longer H.

Steady-state: For every client request op, the primary b applies op to its update history H and sends an ACCEPT message to the backups containing e and the adopted history H; in practice, only a delta-increment of H is sent each time. When a backup receives an ACCEPT message, if $e \geq e_{prepare}$, it adopts H and sets both $e_{prepare}$ and e_{accept} to e. It then sends an acknowledgment back to b. Once a quorum of followers have acknowledged the ACCEPT message, and hence the history prefix, b commits it by sending a COMMIT message to the backups.

Primary order. Because the primary server broadcasts state changes, Zab must ensure that they are received in order. Specifically, if state change c is received by a backup from a primary, all changes that precede c from that primary must also have been received by the backup. Zab refers to this ordering guarantee as *local primary order*. The local primary order property, however, is not sufficient to guarantee order when primaries can crash. It is also necessary that a new primary replacing a previous primary guarantees that once it broadcasts new updates, it has received all changes of previous primaries that have been delivered or that will be delivered. The new primary must guarantee that no state changes from previous primaries succeed its own state changes in the order of delivered state changes. Zab refers to this ordering guarantee as *global primary order*.

The term *primary order* refers to an ordering that satisfies both local and global primary orders. While the discussion above has been in the context of ZooKeeper and Zab, any primary/backup system in which a primary executes operations and broadcasts state changes to backups will need primary order. The importance of this property has already been highlighted in [13, 3]. Here, we further exploit this property to simplify system reconfiguration.

Configurations in Zookeeper. A ZooKeeper deployment currently uses a static configuration S for both clients and servers, which comprises a set of servers, with network address information, and a quorum system. Each server can be defined as a *participant*, in which case it participates in Zab as a primary or as backup, or an *observer*, which means that it does not participate in Zab and only learns of state updates once they are committed. For consistent operation each server needs to have the same configuration S, and clients need to have a configuration that includes some subset of S.

Performing changes to a ZooKeeper configuration is currently a tricky task. Suppose, for example, that we are to add three new servers to a cluster of two servers. The

[3]For the sake of readers familiar with Zookeeper and its terminology, in the context of Zookeeper and Zab we use the term "leader" for "primary" and "follower" for "backup" (with no difference in meaning).

[4]Zab contains a preparatory step that optimistically chooses a candidate-leader that already has the up-to-date history, eliminating the need to copy the latest history from one of the backups during startup.

two original members of the cluster hold the latest state, so we want one of them to be elected leader of the new cluster. If one of the three new servers is elected leader, the data stored by the two original members will be lost. (This could happen if the three new servers start up, form a quorum, and elect a leader before the two older servers start up.) Currently, membership changes are done using a "rolling restart" – a procedure whereby servers are shut down and restarted in a particular order so that any quorum of the currently running servers includes at least one server with the latest state. To preserve this invariant, some reconfigurations (in particular, the ones in which quorums from the old and the new configurations do not intersect) require restarting servers multiple times. Service interruptions are unavoidable, as all servers must be restarted at least once. Rolling restart is manually intensive, error prone, and hard to execute correctly even for expert ZooKeeper users (especially if failures happen during reconfiguration). Furthermore, this procedure gives no insight on how clients can discover or react to membership changes.

The protocol we propose in this paper overcomes such problems and enables dynamic changes to the configuration without restarting servers or interrupting the service.

Reconfiguring a state-machine. Primary/backup replication is a special instance of a more general problem, *state-machine replication* (SMR). With SMR, all replicas start from the same state and process the same sequence of operations. Agreement on each operation in the sequence is reached using a consensus protocol such as Paxos [15]. Similarly to our algorithm, most existing SMR approaches use the state-machine itself to change system configuration, that is, the reconfiguration is interjected as any other operation in the sequence of state-machine commands [16]. The details of implementing this in a real system are complex, as pointed out in a keynote describing the implementation of Paxos developed at Google [6]. One of the core difficulties is that a reconfiguration is very different from other SMR commands, in that it changes the consensus algorithm used to agree on the subsequent operations in the sequence.

To better understand the issue, notice that in SMR there is no dependency among operations and thus separate consensus decisions are made for the different "slots" in the history sequence. Thus, if operations 1 through 100 are proposed by some server, it is possible that first operation 1 is committed, then 80, then 20, and so on. It is also possible that an operation proposed by a different server is chosen for slot number 2. Suppose now that a server proposes *reconfiguration* for slot 50. If the proposal achieves a consensus decision, it is most natural to expect that it changes the set of servers that need to execute the consensus algorithm on subsequent slots (51 and onward). Unfortunately, above we stated that we already committed slot number 80 using the current configuration; this could lead to inconsistency (a split brain scenario). We must therefore delay the consensus decisions on a slot until we know the configuration in which it should be executed, i.e., after all previous slots have been decided. As a remedy, Lamport proposed to execute the configuration change α slots in the future, which then allows the consensus algorithms on slots n through $n + \alpha - 1$ to execute simultaneously with slot n. In this manner, we can maintain a 'pipeline' of operations, albeit bounded by α.

Thus, standard SMR reconfiguration approaches limit the concurrent processing of *all* operations, because of the uncertainty introduced by the ability to reconfigure. We use a different approach that overcomes this limitation by exploiting primary order. Our reconfiguration algorithm speculatively executes any number of operations concurrently.

3 Primary/Backup Reconfiguration

We start with a high level description of our reconfiguration protocol. In general, in order for the system to correctly move from a configuration S to a configuration S' we must take the following steps [3], illustrated in Figure 2:

1. persist information about S' on stable storage at a quorum of S (more precisely, a consensus decision must be reached in S regarding the "move" to S');
2. deactivate S, that is, make sure that no further operations can be committed in S;
3. identify and transfer all committed (and potentially committed) state from S to S', persisting it on stable storage at a quorum of S' (a consensus decision in S' regarding its initial state).
4. activate S', so that it can independently operate and process client operations.

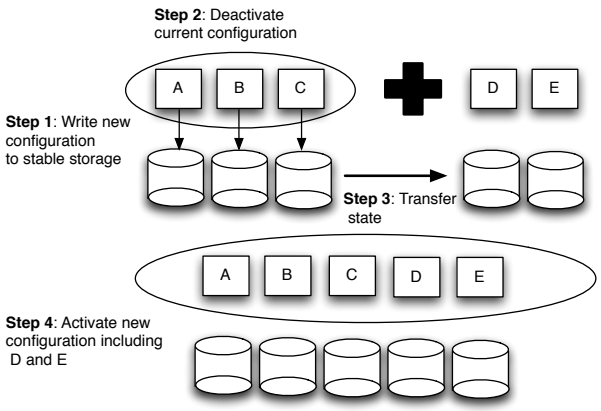

Figure 2: The generic approach to reconfiguration: adding servers D and E to a cluster of three servers A, B and C.

Note that steps 1 and 2 are necessary to avoid split brain. Steps 3 and 4 make sure that no state is lost when moving to S'. The division into four steps is logical and somewhat arbitrary – some of these steps are often executed together.

In a primary/backup system many of the steps above can be simplified by taking advantage of properties already provided by the system. In such systems, the primary is the only one executing operations, producing state-updates which are relative to its current state. Thus, each state-update only makes sense in the context of all previous updates. For this reason, such systems reach agreement on the prefix of updates and not on individual operations. In other words, a new update can be committed only after all previous updates commit. This does not, however, limit concurrency: a primary can execute and send out any number of state-updates speculatively to the backups, however updates are always committed in order and an uncommitted suffix of updates may later be revoked from a backup's log if the primary fails without persisting the update to a sufficient number of replicas (a quorum). Reconfiguration fits this framework well – we interject a configuration update operation, *cop*, in the stream of normal state-updates, which causes a reconfiguration after previously scheduled updates are committed (in state-machine terminology, $\alpha = 1$). Thus, a reconfiguration is persisted to stable storage in the old configuration S just like any other operation in S (this corresponds to step 1 above). At the same time, there is no need to explicitly deactivate S – step 2 follows from the speculative nature of the execution. Just like with any other state-update, the primary may execute any number of subsequent operations, speculatively assuming that *cop* commits. Primary order then makes sure that such operations are committed only after the entire prefix up to the operation (including the configuration change *cop*) is committed, i.e., they can only be committed in the new configuration as required by step 2.

Since the primary is the only one executing operations, its local log includes all state changes that may have been committed; hence, in step 3 there is no need to copy state from other servers. Moreover, we start state transfer ahead of time, to avoid delaying the primary's pipeline. When processing the reconfiguration operation *cop*, the primary only makes sure that state transfer is complete, namely that a quorum of S' has persisted all operations scheduled up to and including *cop*. Finally, in step 4, the primary activates S'.

If the primary of S fails during reconfiguration, a candidate primary in S must discover possible decisions made in step 1. If a new configuration S' is discovered at this stage, the candidate primary must first take steps to commit the stream of commands up to (and including) the operation proposing S', and then it must repeat steps 2–4 in order to transition to S'. Unlike the original primary, the new candidate primary needs to perform a startup-phase in S' and discover the potential actions of a previous primary in S' as well. This presented an interesting challenge in the Zab realm, since a primary in Zab usually has the most up-to-date prefix of commands, and enforces it on the backups. However, a new primary elected from S might have a staler state compared to servers in S'. We must therefore make sure that no committed updates are lost without introducing significant changes to Zab. Below (in Section 3.1), we describe the solution we chose for this pragmatic issue and the *Activation Property* it induces.

We now dive into the details of our protocol. Due to space limitations, we omit the formal proofs here and focus on the intuition behind our algorithm.

3.1 Stable primary

We start by discussing the simpler case, where the primary P of the current configuration S does not fail and continues to lead the next configuration. Figure 3 depicts the flow of the protocol.

pre-step: In order to overlap state-transfer with normal activity, backups in S' connect to the current primary, who initializes their state by transferring its currently committed prefix of updates H. With Zab, such state-transfer happens automatically once backups connect to the primary, and they continue receiving from P all subsequent commands (e.g., op_1 and op_2 in Figure 3), making the transition to S' smooth.

step 1: The primary p schedules *cop*, the reconfiguration command, at the tail of the normal stream of updates. It sends an ACCEPT message containing *cop* to all the backups connected to it (a backup may belong to S and/or to S') and waits for acknowledgments. Consensus on the next configuration is reached once a quorum of S acknowledges *cop*.

step 2: The primary does not stall operations it receives after *cop*. Instead, they are executed immediately and scheduled after *cop*. In principle, all updates following *cop* are the responsibility of S'.

step 3: Transfer of commands has already been initiated in the pre-step; now, p waits for acknowledgement for *cop* and the history of commands which precede it from a quorum of S'.

step 4: Once *cop* is acknowledged by both S and S', the primary commits *cop* and activates S' by sending an ACTIVATE message to backups. Similarly to an ACCEPT, ACTIVATE includes the primary's epoch e and processed by a backup only if e is greater or equal to this backup's $e_{prepare}$.

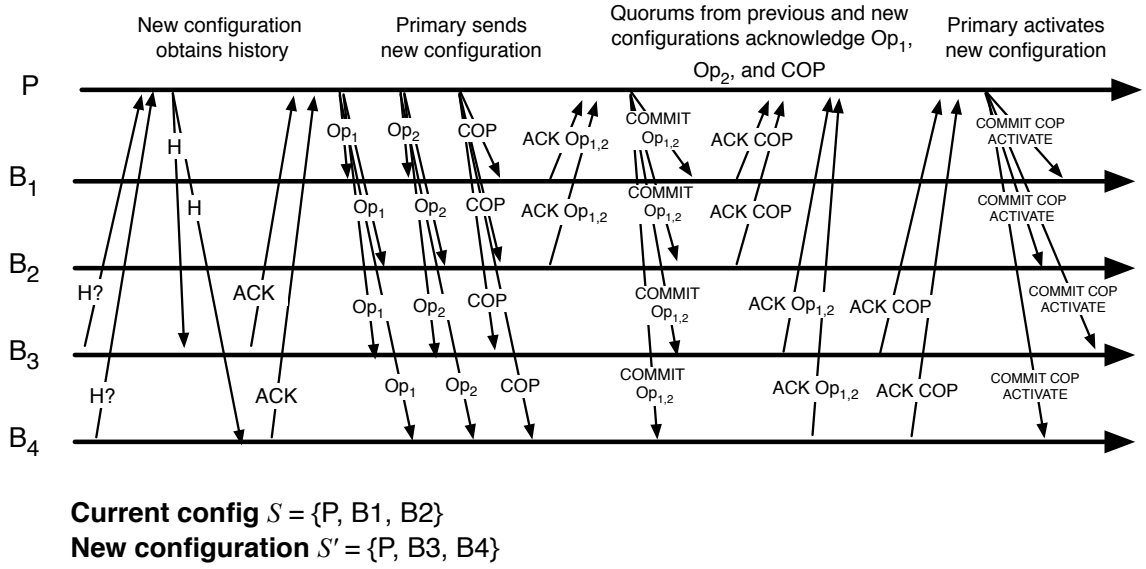

Figure 3: Reconfiguration with a stable primary P.

As mentioned earlier, in order to be compatible with Zookeeper's existing mechanism for recovery from leader failure, we guarantee an additional property:

Activation Property. before ACTIVATE is received by a quorum of S', all updates that may have been committed are persisted to stable storage by a quorum of S.

To guarantee it, we make a change in step 2:

step 2': an update scheduled after *cop* and before the activation message for S' is sent can be committed by a primary in S' only once a quorum of both S and S' acknowledge the update (of course, we also require all preceding updates to be committed). Updates scheduled after the ACTIVATE message for S' is sent, need only be persisted to stable storage by a quorum of S' in order to be committed.

Since the current primary is stable, it becomes the primary of S', and it may skip the startup-phase of a new primary (described in Section 2), since in this case it knows that no updates were committed in S'.

Cascading reconfigurations. Even before ACTIVATE is sent for a configuration S', another reconfiguration operation cop' proposing a configuration S'' may be scheduled by the primary (see Figure 4 below). For example, if we reconfigure to remove a faulty member, and meanwhile detect another failure, we can evict the additional member without ever going through the intermediate step. We streamline cascading reconfigurations by *skipping* the activation of S'.

In the following example, updates u_1 through u_4 are sent by the primary speculatively, before any of them commits, while u_5 is scheduled after all previous updates are committed and the activation message for the last proposed configuration (S'') is sent out.

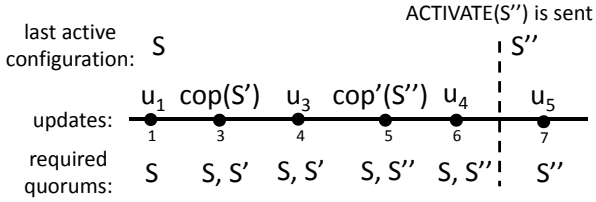

Figure 4: Cascading reconfigurations

Notice that for a given update, only the last active and the last proposed configuration (at the time this update is scheduled) are involved in the protocol steps for that update. Once there is a sufficient window of time between reconfigurations that allows state-transfer to the last proposed configuration to complete, the primary activates that configuration. We note that currently the described extension of the protocol to support multiple concurrent reconfigurations is not being integrated into Zookeeper; for simplicity, a reconfiguration request is rejected if another reconfiguration is currently in progress. (The issuing client may resubmit the reconfiguration request after the current reconfiguration operation completes.)

3.2 Primary failure or replacement

Until now, we assumed that the primary does not fail during the transition to S' and continues as the primary of S'. It remains to ensure that when it is removed or fails, safety is still guaranteed. First, consider the case that

the current primary in S needs to be replaced. There are many reasons why we may want to replace a primary, e.g., the current primary may not be in S', its new role in S' might not allow it to continue leading, or even if the IP address or port it uses for communication with the backups needs to change as part of the reconfiguration.

Our framework easily accommodates this variation: The old primary can still execute operations scheduled after *cop* and send them out to connected backups but it does not commit these operations, as these logically belong in S'. It is the responsibility of a new primary elected in S' to commit these operations. As an optimization, we explicitly include in an ACTIVATE message the identity of a designated, initial primary for S' (this is one of the backups in S', which has acknowledged the longest prefix of operations, including at least *cop*). As before, this primary does not need to execute the startup-phase in S' since we know that no primary previously existed in S'. Obviously, if that default primary fails to form a quorum, we fall-back to the normal primary election in S'.

Likewise, the case of a primary failure after S' has been activated is handled as a normal Zab leader re-election.

An attempted reconfiguration might not even reach a quorum of backups in S, in which case it may disappear from the system like any other failed command.

We are left with the interesting case when a primary-candidate b in S discovers a pending attempt for a consensus on $cop(S')$ by the previous primary. This can mean either that *cop* was already decided, or simply that some backup in the quorum of b heard *cop* from p. As for any other command in the prefix b learns, it must first commit *cop* in S (achieving the consensus decision required in step 1). However, executing *cop* requires additional work, and b must follow the reconfiguration steps to implement it.

The only deviation from the original primary's protocol is that b must follow the startup-phase of a new primary (Section 2) in both S and S'. In order to do so, b connects to the servers in S'. When connecting to a server b' in S', b finds out whether b' knows of the activation of S' (or a later configuration). If S' has been activated, servers in S' may know of newer updates unknown to b, hence b should not attempt to perform state transfer (otherwise it may cause newer updates to be truncated). Instead, b restarts primary re-election in S' (and in particular connects to an already elected primary in S' if such primary exists). Otherwise, b implicitly initiates state-transfer to b' (much like its predecessor did). This includes at least all updates up to *cop* but may also include updates scheduled by the previous primary after *cop*.

This leads us to a subtle issue resulting from our desire to introduce as few changes as possible to the existing implementation of leader recovery in Zookeeper. Recall that the stream of updates by the previous primary may continue past *cop*, and so backups in S' may have a longer history of commands than b. In Zookeeper, connecting to b would cause them to truncate their history. This is exactly why we chose to preserve the *Activation Property*. If b succeeds to connect to a quorum of S' without learning of the activation of S', we know that all updates that may have been committed are stored at a quorum of S. Thus, b will find all such updates once completing the startup-phase in S; in fact, in Zookeeper the candidate b is chosen (by preliminary selection) as the most up-to-date backup in S (that can communicate with a quorum of S), so it will already have the full prefix and no actual transfer of updates is needed during the startup-phase.

Finally, note that b might discover more than a single future reconfiguration while performing its startup-phase in S. For example, it may see that both S' and S'' were proposed. b may in this case skip S' and run the startup-phase in S and S'', after which it activates S''.

3.3 Progress guarantees

As in [2], the fault model represents a dynamic interplay between the execution of reconfiguration operations and the "adversary": The triggering of a reconfiguration event from S to S' marks a first transition. Until this event, a quorum of S is required to remain alive in order for progress to be guaranteed. After it, both a quorum of S and of S' are required to remain alive. The completion of a reconfiguration is generally not known to the participants in the system. In our protocol, it occurs when the following conditions are met: (a) a quorum of S' receives and processes the ACTIVATE message for S', and (b) all operations scheduled before S' is activated by a primary are committed. The former condition indicates that S' can independently process new operations, while the latter indicates that all previous operations, including those scheduled while the reconfiguration was in progress, are committed (it is required due to the *Activation Property* and step 2'). Neither conditions are externally visible to a client or operator submitting the reconfiguration command. However, there is an easy way to make sure that both condition are met: after the reconfiguration completes at the client, it can submit a no-op update operation; once it commits, we know that both conditions (a) and (b) are satisfied (the no-op update can be automatically submitted by the client-side library). An alternative way to achieve this is to introduce another round to the reconfiguration protocol (which, for simplicity and compatibility with Zab, we decided to avoid). Either way, once (a) and (b) are satisfied, the fault model transitions for the second time: only a quorum of S' is required to survive from now on.

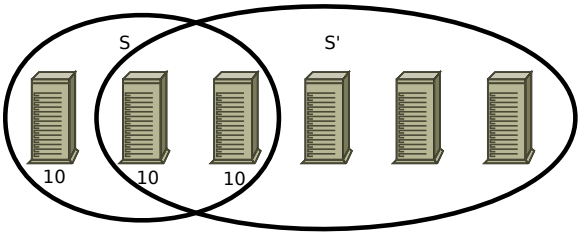

Figure 5: A balanced service (10 clients are connected to each server) about to move to a new configuration S'.

4 Reconfiguring the Clients

Once servers are able to reconfigure themselves we are left with two problems at the client. First, clients need to learn about new servers to be able to connect to them. This is especially important if servers that a client was using have been removed from the configuration or failed. Second, we need to rebalance the load on the servers. ZooKeeper clients use long-lived connections and only change the server they are connected to if it has failed. This means that new servers added to a configuration will not take on new load until new clients start or other servers fail. We can solve the first problem using DNS and by having clients subscribe to configuration changes (see Section 5) in Zookeeper. For lack of space here we concentrate on the second problem.

Figure 5 shows a balanced service with configuration S that is about to move to S'. There are 30 clients in the system and each of the three servers in S serves 10 of the clients. When we change to S' we would like to make sure the new system is also load-balanced. In this example this means that each server should service 6 clients. We would also like to move as few clients as possible since session reestablishment puts load on both the clients and the servers and increases latency for client requests issued while the reestablishment is in process. A final goal is to accomplish the load balance using only logic at the clients so as not to burden the servers.

We denote by M the set of servers that are in both configurations, $S \cap S'$. Machines that are in the old configuration S but not in the new configuration we will label O, that is, $O = S \setminus M$. Machines that are in the new configuration S' but not in the old configuration are labeled N, that is, $N = S' \setminus M$. Denote the total number of clients by C. The number of clients connected to server i in S is denoted by *l(i, S)*.

In general, for a server $i \in S'$, the expected number of clients that connect to i in S', *E(l(i, S'))* is the number of clients connected to it in S plus the number of clients migrating from other servers in S to i (we denote a move from server j to server i by $j \to i$ and a move to any of the servers in a set G by $j \to G$) minus the number of clients migrating from i to other servers in S':

$$E(l(i,S')) = l(i,S) + \sum_{j \in S \wedge j \neq i} l(j,S) * Pr(j \to i) - l(i,S) \sum_{j \in S' \wedge j \neq i} Pr(i \to j)$$

We solve for the probabilities assuming that the load was uniform across all servers in S and requiring that the expected load remains uniform in S' (in the example of Figure 5, we require that $E(l(i,S')) = 6$). Intuitively, the probability of a client switching to a different server depends on whether the cluster size increases or shrinks, and by how much. We have two cases to consider:

Case 1: $|S| < |S'|$ Since the number of servers is increasing, load must move off from all servers. For a server $i \in M$ we get: $E(l(i,S')) = l(i,S) - l(i,S) * Pr(i \to N)$. We can substitute $l(i,S) = C/|S|$ since load was balanced in S, and $E(l(i,S')) = C/|S'|$ since this is what we would like to achieve. This gives:

Rule 1. *If $|S| < |S'|$ and a client is connected to M, then with probability $1 - |S|/|S'|$ the client disconnects from its server and then connects to a random server in N. That is, the choice among the servers in N is made uniformly at random.*

Notice that clients connected to servers in O should move only to N as servers in M have too many clients to begin with.

Rule 2. *If $|S| < |S'|$ and a client is connected to O, then the client moves to a random server in N.*

Case 2: $|S| \geq |S'|$ Since the number of servers decreases or stays the same, the load on each server in S' will be greater or equal to the load on each server in S. Thus, a server in M will not need to decrease load:

Rule 3. *If $|S| \geq |S'|$ and a client is connected to a server in M, it should remain connected.*

The total collective load in S' on all servers in M is the load on M in S plus the expected number of clients that move to M from O:

$$\frac{|M|C}{|S'|} = \frac{|M|C}{|S|} + \frac{|O|C}{|S|} * Pr(i \to M | i \in O)$$

We thus get our last rule:

Rule 4. *If $|S| \geq |S'|$ and a client is connected to a server in O, it moves to a random server in M with probability $\frac{|M|(|S|-|S'|)}{|S'||O|}$; otherwise, moves to a random server in N.*

By having each client independently apply these rules, we achieve uniform load in a distributed fashion.

5 Implementation and Evaluation

We implemented our server and client-side protocols in Apache Zookeeper. To this end we updated the server-side library of ZooKeeper (written in Java) as well as the two client libraries (written in Java and in C). We

added a *reconfig* command to the API that changes the configuration, a *config* command that retrieves the current configuration and additionally allows users to subscribe for configuration changes and finally the *update-server-list* command that triggers the client migration algorithm described in Section 4. We support two reconfiguration modes. The first is incremental – it allows adding and removing servers to the current configuration. The second type of reconfiguration is non-incremental, which means that the user specifies the new configuration. This method allows changing the quorum system dynamically. We allow adding and removing servers as well as changing server roles. We also support dynamically changing the different network addresses and ports used by the system.

In the remainder of this section we evaluate the impact of reconfigurations on Zookeeper clients. We focus on the effect on throughput and latency of normal operations as well as on load balancing.

We performed our evaluation on a cluster of 50 servers. Each server has one Xeon dual-core 2.13GHz processor, 4GB of RAM, gigabit ethernet, and two SATA hard drives. The servers run RHEL 5.3 using the ext3 file system. We use the 1.6 version of Sun's JVM.

We used the Java server configured to log to one dedicated disk and take snapshots on another. Our benchmark client uses the asynchronous Java client API, and each client is allowed up to 100 outstanding requests. Each request consists of a read or write of 1K of data (typical operation size). We focus on read and write operations as the performance of all the operations that modify the state is approximately the same, and the performance of non state modifying operations is approximately the same. When measuring throughput, clients send counts of the number of completed operations every 300ms and we sample every 3s. Finally, note that state-transfer is always performed ahead of time and a reconfig operation simply completes it, thus our measurements do not depend on the size of the Zookeeper database.

Throughput. We first measure the effect of dynamic reconfigurations on throughput of normal operations. To this end, we used 250 simultaneous clients executing on 35 machines, up to 11 of which are dedicated to run Zookeeper servers (typical installations have 3-7 servers, so 11 is larger than a typical setting). Figure 6 shows the throughput in a saturated state as it changes over time. We show measurements for workloads with 100%, 50%, 30% and 15% write operations. The ensemble is initially composed of 7 servers. The following reconfiguration events are marked on the figure: (1) a randomly chosen follower is removed; (2) the follower is added back to the ensemble; (3) the leader is removed; (4) former leader is added back to the ensemble as a follower; (5) a randomly chosen follower is removed, and (6) the follower is added back to the ensemble.

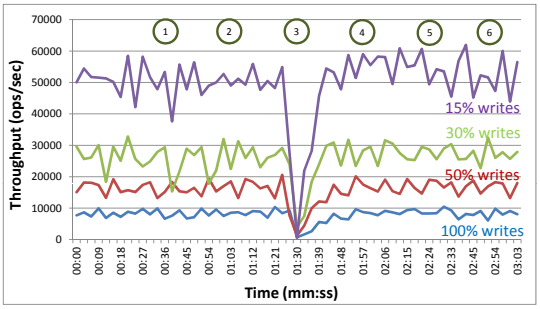

Figure 6: Throughput during configuration changes.

Unsurprisingly, removing the leader has the most significant effect on throughput. In Zookeeper, any leader change (e.g., due to the failure of the previous leader) always renders the system temporarily unavailable, and a reconfiguration removing the leader is no different in that respect. Note that in Zookeeper, each follower is connected only to one leader. Thus, when the leader changes, followers disconnect from the old leader and only after a new leader is established can submit further operations. While this explains why write operations cannot be executed in the transition period (and the throughput drop for a 100% write workload), the reasons for disabling any read activity during leader election (which causes the throughput drop for read intensive workloads) are more subtle. One of the reasons is that Zookeeper guarantees that all operations complete in the order they were invoked. Thus, even asynchronous invocations by the same thread have a well defined order known in advance to the programmer. Keeping this in mind, consider a read operation that follows a write by the same client (not necessarily to the same data item). The read will only be able to complete after the write, whereas writes await the establishment of a new leader[5].

The throughput quickly returns to normal after a leader crash or removal. Notice that read intensive workloads are more sensitive to removal and addition of followers. This is due to the effect of client migration to other followers for load balancing (we explore load-balancing further in Section 5.1). Still, the change in throughput with such reconfigurations is insignificant compared to normal fluctuations of system throughput. The reason is the in-order completion property of Zookeeper mentioned above; writes, which are broadcasted by the leader to followers, determine the throughput of the system. More precisely, the network interface of the leader is the bottleneck. Zookeeper uses a single IP address for leader-follower communication. The throughput of

[5]In Zookeeper 3.4, each operation is blocked until every operation (not necessarily by the same client) previously submitted to the same follower completes; this is not necessary to guarantee the in-order completion semantics and may therefore change in the future.

the system therefore depends on the number of servers connected to the leader, not the number of followers in the ensemble. Note, however, that removing or adding a server from the cluster using the reconfig command does not necessarily change the number of connections. Although a removal excludes a server from participating in Zab voting it does not necessarily disconnect the follower from the leader; an administrator might want to first allow clients to gracefully migrate to other followers and only then disconnect a removed follower or shut it down. In addition, removing a follower is sometimes necessary as an intermediate step when changing its role in the protocol (for example, in some situations when converting an observer to a follower). Figure 7 illustrates this point. It shows two executions, with 30% writes, 250 clients and 11 servers initially in the cluster. There are two reconfiguration events, each removes multiple servers from the cluster. In one execution, the removed servers are turned off while in the other (similarly to Figure 6) removed followers maintain their connections to the leader. The graph shows that disconnecting the servers indeed increases system throughput. This shows, that over-provisioning a cluster by adding more replicas (even if those replicas are observers) can be detrimental to Zookeeper throughput. A better strategy is to reconfigure the system dynamically with changing load.

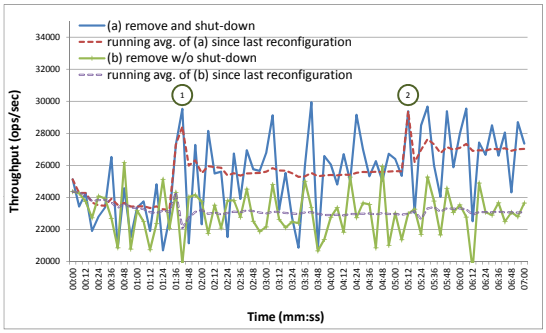

Figure 7: Throughput during configuration changes. Initially there are 11 servers in the cluster. The workload includes 30% writes. Configuration changes: (1) four followers are removed, (2) two additional followers are removed.

Latency. Next, we focus on the effect of reconfiguration on the latency of other requests. We measured the average latency of write operations performed by a single client connected to Zookeeper; the writes are submitted in batches of 100 operations, after all previously submitted writes complete. Initially, the cluster contains seven replicas and writes have an average latency of 10.8ms[6].

We then measured the impact of removing replicas on latency. A client submits a reconfiguration request to remove four randomly chosen followers which is immediately followed by a second write batch. If we use the reconfiguration procedure described in Section 3, we get an average latency again of 10.8ms. However, if we stall the request pipeline during the reconfiguration, the average latency increases to 15.2ms.

With three replicas, our average write latency is 10.5ms. The client then requests to add back four replicas, followed by another write batch. Using our approach write latency is at 11.4ms and jumps to 18.1ms if we stall the pipeline.

Leader removal. Finally, we investigate the effect of reconfigurations removing the leader. Note that a server can never be added to a cluster as leader as we always prioritize the current leader. Figure 8 shows the advantage of designating a new leader when removing the current one, and thus avoiding leader election. It depicts the average time to recover from a leader crash versus the average time to regain system availability following the removal of the leader. The average is taken on 10 executions. We can see that designating a default leader saves up to 1sec, depending on the cluster size. As cluster size increases, leader election takes longer while using a default leader takes constant time regardless of the cluster size. Nevertheless, as the figure shows, cluster size always affects total leader recovery time, as it includes synchronizing state with a quorum of followers.

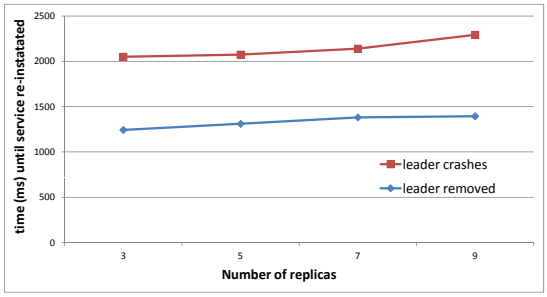

Figure 8: Unavailability following leader removal or crash.

5.1 Load Balancing

In this section, we would like to evaluate our approach for load balancing clients as part of configuration changes. To this end, we experiment with a cluster of nine servers and 1000 clients. Clients subscribe to configuration changes using the *config* command and update their list of servers using the *update-server-list* command when notified of a change. In order to avoid mass migration of clients at the same time, each client waits for a random period of time between 0 and 5sec. The graphs presented below include four reconfiguration events: (1) remove one random server; (2) remove two random servers; (3) remove one random server and add the three previously removed servers, and (4) add the server removed in step 3.

[6]the average latencies presented here are taken over 150 executions or the described experiment and lie within 0.3ms of the real average with 95% confidence

We evaluate load balancing by measuring the minimum and maximum number of clients connected to any of the servers and compare it to the average (number of clients divided by the current number of servers). When the client connections are balanced across the servers, the minimum and maximum are close to the average, i.e., there are no overloaded or under-utilized servers.

Baseline. Our first baseline is the current implementation of load balancing in Zookeeper. The only measure of load is currently the number of clients connected to each server, and Zookeeper is trying to keep the number of connections the same for all servers. To this end, each client creates a random permutation of the list of servers and connects to the first server on its list. If that server fails, it moves on to the next server on the list and so on (in round robin). This approach works reasonably well when system membership is fixed, and can easily accommodate server removals. It does not, however, provide means for incorporating a new server added to the cluster. In order to account for additions in this scheme, we replace the client's list with a new list of servers. The client maintains its connection unless its current server is not in the new list. Figure 9 shows that load is balanced well as long as we perform removals (steps 1 and 2), however when servers are added in steps 3 and 4 the newly added servers are under-utilized. In the beginning of step 3 there are six servers in the system, thus approximately 166 clients are connected to every server. When we remove a server and add three new ones in step 3, the clients connected to the removed server migrate to a random server in the new configuration. Thus, every server out of the eight servers in the new configuration gets an expected 21 additional clients (the newly added servers will only have these clients, as no other clients disconnect from their servers). In step 4 we add back the last server, however no clients migrate to this server. Although all clients find out about the change and update their lists, no client disconnects from its server as it is still part of the system.

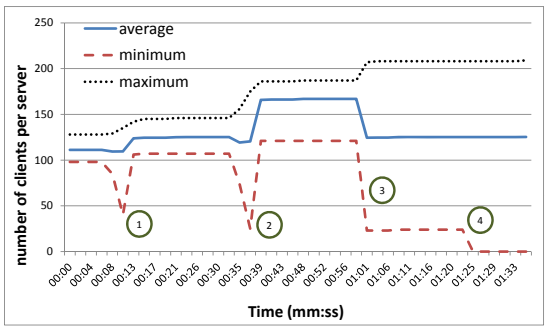

Figure 9: Baseline load balancing.

To mitigate the problem illustrated in Figure 9 we could of course disconnect all clients and re-connect them to randomly chosen servers in the new configuration. This, however, creates excessive migration and unnecessary loss of throughput. Ideally, we would like the number of migrating clients to be proportional to the change in membership. If only a single server is removed (or added), only clients that were (or should be) connected to that server should need to migrate.

Consistent Hashing. A natural way to achieve such limited migration, which we use as a second baseline, is to associate each client with a server using consistent hashing [14]. Client and server identifiers are randomly mapped to points in an m-bit space, which can be seen as circular (i.e., 0 follows $2^m - 1$). Each client is then associated with the server that immediately follows it in the circle. If a server is removed, only the clients that are associated with it will need to migrate by connecting to the next server on the circle. Similarly, if a new server is added a client migrates to it only if the new server was inserted between the client and the server to which it is currently connected. In order to improve load balancing, each server is sometimes hashed k times (usually k is chosen to be in the order of $log(N)$, where N is the number of servers). To evaluated the approach, we implemented it in Zookeeper. Figure 10 shows measurements for $k = 1$, $k = 5$ and $k = 20$. We used MD5 hashing to create random identifiers for clients and servers ($m = 128$). We can see that higher values of k achieve better load balancing. Note, however, that load-balancing in consistent hashing is uniform only with "high probability", which depends on N and k. In the case of Zookeeper, where 3-7 servers (N) are usually used, the values of N and k are not high enough to achieve reasonable load balancing.

Probabilistic Load Balancing. Finally, Figure 11 shows measurements of load-balancing with the approach we have implemented in Zookeeper as outlined in Section 4. Unlike consistent hashing, in this approach *every* client makes a probabilistic decision whether and where to migrate, such that the expected number of clients per server is the same for every server. As we can see from the figure the difference in number of clients between the server with the most clients and the least clients is very small. Using our simple case-based probabilistic load balancing we are able to achieve very close to optimal load-balance using logic entirely at the client.

6 Related Work

Primary order is commonly guaranteed by Primary/Backup replication systems, e.g., Chubby [5], GFS [8], Boxwood [19], PacificA [21], chain replication [20], Harp [17] and Echo [11]. Although Paxos does not guarantee primary order [13], some systems

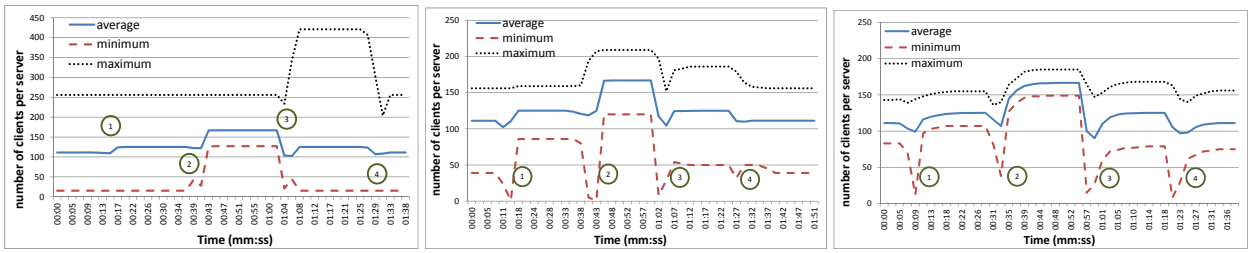

Figure 10: Load balancing using consistent hashing, with $k = 1$ (left), $k = 5$ (middle), and $k = 20$ (right).

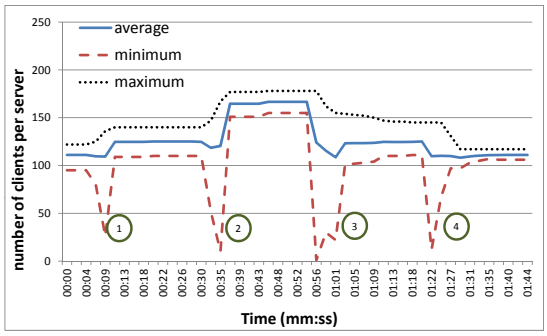

Figure 11: Load balancing using our method (Section 4).

implementing Paxos (such as Chubby and Boxwood) have one outstanding decree at-a-time, which in fact achieves primary-order. This is done primarily to simplify implementation and recovery [19]. Unlike such approaches, we do not limit the the concurrent processing of operations.

Unlike systems such as RAMBO [9], Boxwood [19], GFS [8], Chubby [5], chain replication [20] and PacificA [21] that use an external reconfiguration service, we use the system itself as the reconfiguration engine, exploiting the primary order property to streamline reconfigurations with other operations. Zookeeper is often used by other systems for the exact same purpose, and thus relying on another system for reconfiguring Zookeeper would simply push the problem further as well as introduce additional management overhead. An additional difference from RAMBO is that in our design, every backup has a single "active" configuration in which it operates, unlike in RAMBO where servers maintain a set of possible configurations, and operate in all of them simultaneously. Finally, RAMBO and several other reconfigurable systems (see [1] for a survey), are designed for reconfiguring read/write storage, whereas Zookeeper provides developers with arbitrary functionality, i.e., a universal object via consensus [10]; the read/write reconfiguration problem is conceptually different [2] than the one we address in this paper.

SMART [18] is perhaps the most practical implementation of Paxos [15] SMR published in detail. SMART uses Lamport's α parameter to bound the number of operations that may be executed concurrently (see Section 2). In addition, SMART uses configuration-specific replicas: if the cluster consists of replicas A, B, and C and we are replacing C with D, SMART runs two replicas of A and two of B, one in the new configuration and one in the old, each running its own instance of the replication protocol. An important design consideration in our work has been to introduce minimal changes to Zookeeper, as it is used in production by many commercial companies. Dynamically creating additional Zookeeper replicas just for the purpose of reconfiguration adds an implementation and management overhead that would not be acceptable to Zookeeper users. Unlike SMART, we do not limit concurrency or require any additional resources to reconfigure.

FRAPPE [4] proposes a different solution. Each server in FRAPPE works with a set of possible configurations, similarly to RAMBO. If a reconfiguration is proposed for history slot n, any number of operations can be proposed after n, however their completion is speculative – users are aware that even though these operations commit they may later be rolled back if a different operation is chosen for slot n. This requires servers to maintain a speculative execution tree, each branch corresponding to an assumption on the decision on some reconfiguration for a particular history slot. In case the reconfiguration is chosen for slot n and once state transfer is complete, the speculative operations become permanently committed and the corresponding tree-branch is merged into the "trunk". Otherwise, the branch is simply abandoned. Similarly to SMART and FRAPPE, we do not require any intersection between the memberships of consecutive configurations. The algorithm presented in this paper processes updates speculatively, similar to FRAPPE. However, our algorithm does not require servers to work with or explicitly manage multiple configurations and it does not expose speculative operation completions to the clients.

Group communication systems that provide virtual synchrony [7, 3] are perhaps closer to Zookeeper than Paxos-style replicated state machines. In such systems, a group of processes may exchange messages with others in the group, and the membership of the group (called a view) may change. Virtual synchrony guarantees that all processes transferring from one view to the next agree on the set of messages received in the previous view. Note

that they do not necessarily agree on the order of messages, and processes that did not participate in the previous view do not have to deliver these messages. Still, virtual synchrony is similar to primary order in the sense that it does not allow messages sent in different configurations to interleave just as primary order does not allow messages sent by different leaders to interleave. Unlike state-machine replication systems, which remain available as long as a quorum of the processes are alive, group communication systems must react to every failure by removing the faulty process from the view. While this reconfiguration is in progress, client operations are not processed. Other systems, such as Harp [17] and Echo [11] follow similar methodology, stopping all client operations during reconfigurations. Conversely, our design (similarly to state-machine replication systems) tolerates failures as long as a quorum of the replicas remains available, and allows executing client operations while reconfiguration and state-transfer are in progress.

7 Conclusions

Reconfiguration is hard in general. It becomes especially hard when reconfiguring the configuration service. While intuitively it seems simple, care must be taken to address all failure cases and execution orderings.

Our reconfiguration protocol builds on properties of Primary/Backup systems to achieve high performance reconfigurations without imposing a bound on concurrent processing of operations or stalling them, and without the high management price of previous proposals.

The load balancing algorithm for distributing clients across servers in a new configuration involves decisions made locally at the client in a completely distributed fashion. We guarantee uniform expected load while moving a minimum number of clients between servers.

We implemented our protocols in an existing open-source primary/backup system, and are currently working on integrating it into production. This involved simple changes, mostly to the commit and recovery operations of Zookeeper. Our evaluation shows that there are minimal disruptions in both throughput and latency using our approach.

While the methods described in this paper were implemented in the context of ZooKeeper, the primary order property we have taken advantage of is commonly provided by Primary/Backup systems.

Acknowledgments

We would like to thank Marshall McMullen for his valuable contributions to this project. We thank the Zookeeper open source community and in particular to Vishal Kher, Mahadev Konar, Rakesh Radhakrishnan and Raghu Shastry for their support, helpful discussions, comments and thorough reviews of this work. Finally, we would like to thank the anonymous reviewers and our shepherd, Christopher Small, for their comments.

References

[1] AGUILERA, M. K., KEIDAR, I., MALKHI, D., MARTIN, J.-P., AND SHRAER, A. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS 102* (2010), 84–108.

[2] AGUILERA, M. K., KEIDAR, I., MALKHI, D., AND SHRAER, A. Dynamic atomic storage without consensus. *J. ACM 58*, 2 (2011), 7.

[3] BIRMAN, K., MALKHI, D., AND VAN RENESSE, R. Virtually synchronous methodology for dynamic service replication. Tech. Rep. 151, MSR, Nov. 2010.

[4] BORTNIKOV, V., CHOCKLER, G., PERELMAN, D., ROYTMAN, A., SHACHOR, S., AND SHNAYDERMAN, I. Frappé : Fast replication platform for elastic services. In *ACM LADIS* (2011).

[5] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *OSDI* (2006), pp. 335–350.

[6] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *PODC* (2007), pp. 398–407.

[7] CHOCKLER, G., KEIDAR, I., AND VITENBERG, R. Group communication specifications: a comprehensive study. *ACM Comput. Surv. 33*, 4 (2001), 427–469.

[8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[9] GILBERT, S., LYNCH, N. A., AND SHVARTSMAN, A. A. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing 23*, 4 (2010), 225–272.

[10] HERLIHY, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[11] HISGEN, A., BIRRELL, A., JERIAN, C., MANN, T., SCHROEDER, M., AND SWART, G. Granularity and semantic level of replication in the echo distributed file system. In *Proceedings of the IEEE Workshop on the Management of Replicated Data* (November 1990).

[12] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technology Conference (2010)* (2010).

[13] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *DSN* (2011), pp. 245–256.

[14] KARGER, D. R., LEHMAN, E., LEIGHTON, F. T., PANIGRAHY, R., LEVINE, M. S., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC* (1997), pp. 654–663.

[15] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[16] LAMPORT, L., MALKHI, D., AND ZHOU, L. Reconfiguring a state machine. *SIGACT News 41*, 1 (2010), 63–73.

[17] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the harp file system. In *SOSP* (1991), pp. 226–238.

[18] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The smart way to migrate replicated stateful services. In *EuroSys* (2006), pp. 103–115.

[19] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI* (2004), pp. 105–120.

[20] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *OSDI* (2004), pp. 91–104.

[21] WEI LIN, MAO YANG, L. Z., AND ZHOU, L. Pacifica: Replication in log-based distributed storage systems. Tech. Rep. MSR-TR-2008-25, MSR, Feb. 2008.

[22] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *SOSP* (2011), pp. 159–172.