ROLLUP: NON-DISRUPTIVE ROLLING UPGRADE

TECHNICAL REPORT 699

VINCENT GRAMOLI, LEN BASS, ALAN FEKETE, DANIEL SUN

MARCH 2015
Abstract—Rolling upgrade consists of upgrading progressively the servers of a distributed system to reduce service downtime. Upgrading a subset of servers requires a well-engineered cluster membership protocol to maintain, in the meantime, the availability of the system state. Existing cluster membership reconfigurations, like Zookeeper and CoreOS et cd, rely on a primary not only for reconfiguration but also for storing information. At any moment, there can be at most one primary, whose replacement necessarily induces disruption.

We propose Rollup, a fully distributed rolling upgrade protocol that uses biquorums to solve consensus. Rollup relies on a candidate leader only for the reconfiguration making service requests scalable. Although Rollup builds upon existing lower-bound results in terms of load and time, its key contribution is to bridge the gap between a long body of theoretical results and recent system achievements through the rolling upgrade application. We evaluate Rollup on an isolated network of 26 physical machines and on an Amazon EC2 cluster of 59 virtual machines. Our comparison against a rolling upgrade using the primary-backup paradigm shows a 6-fold speedup thanks to a reconfiguration building block up to one order of magnitude faster than Zookeeper’s.

Keywords: quorum; strong consistency; Paxos; online upgrade; cluster management; membership changes; active replication

I. INTRODUCTION

Today, major service providers use continuous deployment, for example modifications at Facebook are deployed multiple times a day [15]. To avoid disruption, the service state is replicated in a distributed system and a rolling upgrade progressively upgrades few machines at a time. A rolling upgrade requires subsequent reconfiguration instances, each excluding a small number of $g$ nodes from the set of participants, at a time. These $g$ nodes can be upgraded offline while the state keeps being updated by clients (cf. Figure 1 for $g = 1$). Popular system solutions, like CoreOS used by Facebook, Google and Twitter exploit a key-value store abstraction to replicate the state and a consensus protocol to totally order the state machine configurations. Unfortunately, there is no way to reconfigure this key-value store service without disruption.

The Paxos [28] consensus algorithm that allows candidate leaders to exchange with majorities could be used to reconfigure a key-value store as well. To circumvent the impossibility of implementing consensus with asynchronous communications, Paxos guarantees termination under partial synchrony while always guaranteeing validity and agreement, despite having competing leader candidates proposing configurations. Due to the intricateness of the protocol [39] and the undesirable Paxos anomaly [4], [25], the tendency had been to switch the complexity to a primary-based algorithm that elects a unique leader. Zab, a primary-based atomic broadcast protocol [27] was used in Zookeeper [25], a distributed coordination service. Raft [39] reused the centralization concept of Zookeeper to solve consensus, a problem known to be equivalent to atomic broadcast [22]. The resulting simplification led to the development of various implementations of Raft in many programming languages. The drawback is that these protocols rely heavily on the availability of their primary.

As existing dynamic cluster membership protocols were developed on top of these consensus or atomic broadcast implementations they inherit their primary-based design. For example, Zookeeper’s reconfigurable key-value store [46] forwards both reconfiguration and key-value store requests to the primary, hence benefitting from the pipelining of requests. Raft describes a protocol to change the cluster membership using a transitional configuration. CoreOS et cd [42] integrates the resulting protocol to offer a reconfigurable key-value store. Due to their centralization, they are all badly suited for rolling upgrade where all servers, including the primary, must be upgraded. First, the primary is a bottleneck: the system performance may degrade with the increasing load on the primary [25]. Second and more importantly, excluding this primary from the set of active participants disrupts the service: if the primary is excluded by reconfiguration (even without crashing) the service stops being served before a new primary election terminates at which point pending requests start being served again. Although well suited to reconfigure another service, these protocols are badly suited for rolling upgrade themselves, because a rolling upgrade requires to upgrade “all” servers.
In this paper, we propose Rollup, the first non-disruptive rolling upgrade protocol. It offers a new dynamic cluster membership protocol whose key-value store requests are not routed through a leader but through special quorums. To avoid the leader bottleneck, Rollup uses a variant of the grid quorum system that is known optimal in terms of load [37]. More specifically, it exploits biquorums that consist of two sets of quorums such that each quorum of the first set intersects with any quorum of the second set [30], hence allowing to have pairs of operations writing on disjoint sets of servers (as opposed to majorities or primary/backup quorums) while still guaranteeing strong consistency (linearizability [24]). Its reconfiguration protocol combines Paxos [28] and RAMBO [13], [19], [20], similar to the RDS protocol [10]. While Rollup builds upon known lower-bounds results [37], [14] and formally proven distributed algorithms [10], its general contribution is to bridge the gap between a long body of theoretical research [37], [28], [33], [10] and recent system achievements [42], [26], [39] through the rolling upgrade application.

We evaluated Rollup on an isolated network of 26 physical machines and a virtualized environments of 59 Amazon instances. Rollup consists of more than 3400 lines of Java code and the jar (Java archive) of its bytecode represents 18KB. Our experiments show that Rollup is practical as it upgrades itself on 4 to 50 servers in less than 6 seconds, but this delay can be halved by choosing appropriately a reasonably low granularity $g$. Although Rollup requires numerous messages per participants like any consensus-based protocol, our experiments show that its fully distributed design helps a reasonably large number of configuration participants stay aware of the current configuration.

We compared empirically Zookeeper’s reconfiguration against Rollup’s reconfiguration and show that the latter is up to one order of magnitude faster than the former. In short, the reason is that Zookeeper centralizes the key information at the primary whereas Rollup distributes it on a biquorum system. The efficiency of Zookeeper’s reconfiguration is tied to the steadiness of the primary state while a rolling upgrade has to alter the state of all servers, including the primary, hence affecting substantially the performance of the entire system. Note that this is not an isolated problem as even write requests of CoreOS etcd must all be forwarded to the leader to be consistent.

Although Rollup offers an upgradable service that can be as complex as a distributed storage service, it is not a panacea. In particular, it remains unclear how the same technique can be adapted to upgrade a stronger service including compare-and-swap primitives, like Chubby’s distributed locking service [7].

Finally, we analyzed theoretically the cost of executing a primary-based rolling upgrade, like CoreOS etcd, at large-scale by modeling the rolling upgrade as a discrete time Markov chain. This analysis indicates that using Rollup to upgrade a distributed systems of 25 (resp. 4) servers with a granularity $g = 5$ (resp. $g = 1$) leads to a 6-fold (resp. 7.8-fold) speedup. This relies on the assumption that the primary/leader election ends up selecting one of the available nodes uniformly at random. Hence, the expectation of the number of removed primaries due to reconfigurations can be greater than 1. These results indicate that, in its current form, the primary-based reconfiguration is inherently badly-suited for rolling upgrade.

Section II presents the model. Section III describes Rollup, its reconfiguration and the storage service. Section IV discusses the fault-tolerance, correctness and liveliness of Rollup. Sections V and VI evaluate Rollup experimentally on up to 59 machines and compare it against Zookeeper. Section VII analyzes theoretically the performance benefit of using a quorum-based rather than a primary-based rolling upgrade. Section VIII presents the related work and Section IX concludes.

II. Model

We consider a set of $n \geq 2$ distributed (virtual or physical) servers, with unique identifiers, potentially running services and communicating through message passing. We assume the network communication to be asynchronous but reliable in that every message sent gets eventually delivered with no alteration and no new messages being forged. (We present a TCP-based implementation in Section V.) Each machine may fail by crashing at which point it will stop acting and after a recovery a machine is considered a new machine, but we do not consider arbitrary failures and we assume that there are no more than $\lceil \sqrt{n} - 1 \rceil$ failures during the time it takes to reconfigure (between 45ms and 249ms as we report in Section V-A).

The dynamic set of servers operating the service at a given time are referred to as the participants. We assume that an external oracle provides the client with the address of the participants where the service is available. (Note that this is traditionally achieved by relying on an external or underlying service like DNS servers or an IP anycast layer.) For the sake of simplicity, we consider that each participant runs its own local instance of Rollup to directly treat rollup requests as well.

We assume the existence of at least two ordered versions per service to upgrade so that an operator can issue a rollup request targeting one server to upgrade the service running on all participants from the least recent version to the most recent version. A server exports an interface specific to the current version it operates, hence versions are not necessarily backward compatible. We assume that among the external clients of the service that are not under the control of the administrator, the service responds only to those that request the feature of the current version. It simply ignores incompatible requests by informing the requester about the current version in production and resuming normally.

We consider a replicated protocol, where the same set of information is replicated at multiple servers, also called replicas. The replicas of a particular information can be
organised into quorums that are mutually intersecting sets of replicas. Quorums are appealing for maintaining consistency: updating the value stored at all the members of one quorum guarantees that among all members of any quorum there is at least one replica with the updated value. The set of quorums is called the quorum systems. We give a more precise definition of a particular kind of quorum system that we implemented in Section IV.

III. ROLLUP, AVOIDING DISRUPTION

For the sake of simplicity in the presentation, we start presenting Rollup in the absence of failure and defer the discussion of its fault tolerance to Section IV-A. The Rollup pseudocode is given in Algorithm 1. Rollup relies on two external functions, upg (line 16) and recon (line 15). The key component of Rollup is the reconfiguration recon that we describe in Section III-A and we omit the description of upg that is specific to the way the service is upgraded locally on one machine.

As depicted in Figure 1, Rollup consists in upgrading progressively a service by acting on few servers at a time to ensure that most of the servers keep providing the service. The maximum number of servers transiently removed and upgraded simultaneously, also called granularity, is denoted g. Although this parameter can be tuned to speed up the rolling upgrade, it is preferable to keep it low (typically g = n/10 or g = O(√n)) to maximize the available capacity of the system, as the number of available servers providing the service at any point in time \( t \) is \( m(t) \geq n - g - f \) where \( f \) is the number of failures. Another motivation for keeping g low is also why rolling upgrade is attractive: new versions are error-prone and deploying versions progressively gives time to detect errors and downgrade the system before all clients get affected by the error.

Rollup upgrades all servers present in the system in \( \left\lceil \frac{n}{g} \right\rceil \) phases. Each phase, whose code appears line 13–19, starts when the amount of servers to be upgraded reaches the granularity threshold (\(|\text{toupg}| > g\)) or when there are few servers left non-upgraded (\(|\text{upgd} \cup \text{toupg}| = |S|\)). If this condition is satisfied at line 13, a reconfiguration to upgrade the corresponding batch of servers is triggered. To this end, the members of the new configuration are defined as the members of the former configuration where the servers to be upgraded have been removed (line 14). The reconfiguration service is called to swap from the current configuration to the new one (line 15). Then, the software running on the recently excluded servers get upgraded locally with the new version \( v \) (line 16). Fields get prepared for the upcoming phase: the current configuration and the set of upgraded servers are updated, and the set of servers to upgrade is reset (lines 17–19).

A. Reconfiguration component

We now take a closer look at the recon component of each rollup phase: the distributed reconfiguration. The key to reconfiguration is to totally order configurations using a consensus algorithm and to make sure that the service requests get redirected to the most up-to-date configuration to avoid disruption.

A configuration consists of all service participants and is represented as a biquorum system [50], with two types of quorums (or sets of servers) such that any quorum of one type intersects all quorums of the other type. Such biquorum systems are used to guarantee that active participants agree on a common proposed configuration and to indicate where the data of the service is replicated. Figure 2 gives a high level description of this reconfiguration protocol where horizontal lines indicate servers, overlapping boxes indicate intersecting quorums, and arrows indicate necessary message exchanges. Like Paxos [28], the protocol executes in three phases, each corresponding to a message exchange involving some quorums (the quorums are defined in Section IV-A):

1) The initiator considers himself as a candidate leader, prepares the tentatively greatest ballot number and

---

**Algorithm 1** The Rollup algorithm at server i

1. state of i:
   2. \( S \), the set of service participants
   3. \( cinit \), initial configuration, initially a quorum system over \( S \)
   4. \( ccurr \), the current config, initially \( cinit \)
   5. \( cnext \), the next config, initially \( \emptyset \)
   6. \( upgd \), servers already upgraded, initially \( \emptyset \)
   7. \( toupg \), servers to upgrade, initially \( \emptyset \)

8. rollup(\( v, g \)):
   - \( cinit ← ccurr \)  \( \triangleright \) store initial config
   - \( upgd ← \emptyset \)  \( \triangleright \) reset upgraded set
   - for any \( j \in S \) do
     - \( toupg = toupg \cup \{j\} \)  \( \triangleright \) prepare for upgrade
   - if \( |toupg| = g \lor |\text{upgd} \cup toupg| = |S| \) then
     - \( cnext ← S \backslash toupg \)  \( \triangleright \) make room for upgrade
     - \( \text{recon}(ccurr, cnext) \)  \( \triangleright \) change config.
   - \( \text{upg}(\text{toupg}, v) \)  \( \triangleright \) upgrade unused servers to \( v \)
   - \( ccurr ← cnext \)  \( \triangleright \) move to next config
   - \( \text{upgd} ← \text{upgd} \cup \text{toupg} \)  \( \triangleright \) add upgraded servers
   - \( toupg ← \emptyset \)  \( \triangleright \) no more server to upgrade
   - \( \text{recon}(ccurr, cinit) \)  \( \triangleright \) go back to initial config.
sends it to some read quorums (in our case it sends to all read quorums; while it maximizes fault tolerance, it also incurs more traffic than strictly necessary) of the current configuration and waits for the response of each member of at least one read quorum containing the highest ballot number and the associated configuration each member has already voted for (or ⊥ if none exist).

2) The candidate leader proposes, to some read and write quorums (all of them in our case) of the current configuration, the proposed configuration associated with the largest ballot among the ones he received and the one he prepared, the quorum members either abstain (by sending the largest ballot number they have already voted for) or vote (if they have not yet voted for a larger ballot) by sending their own decision to some read and write quorums of the current configuration.

3) Once a server receives the votes of every member of at least one read and one write quorum of the current configuration, it decides the new configuration and simply propagates this configuration information to a read and write quorum of the current configuration (that is now considered old) and some read and some write quorum of the newly decided configuration. (While the leader may get informed here as well, this is not required.)

The key to ensure that the service is not disrupted is to redirect all service requests from the old to the new configuration: this is achieved as both a read and a write quorum of the old configuration are aware of the new configuration thanks to the propagation of phase (3), as we detail below.

B. Service use-case: Key-value store

We consider a strongly consistent in-memory key-value store in that get(k)/put(k,v)/delete(k)/update(k,v), with the usual semantics, execute as if they were occurring instantaneously between their invocation and response [24]. The reconfiguration transfers the storage state from the old to the new configuration [23]. Avoiding disruption requires that operations can terminate while the state is being transferred. When a participant \( p \) receives an operation request from a client, it first increments a local counter and assigns the resulting counter value (and its own identifier to break tie) to the operation, this value-id pair is referred to as a timestamp. Then this participant \( p \) executes the operation by exchanging messages containing the timestamp in one or two steps, similarly to the way a shared register is emulated in the message-passing model [8], except that the timestamp represents the version of the key-value store not one of its individual pairs:

1) First \( p \) requests the latest updates on the storage state to each member of at least one read quorum of the current configuration. The participants receiving this request answer by sending the new key-value pairs as well as the associated timestamp it currently knows. Once \( p \) receives responses from a read quorum, it deduces the latest updates as the ones associated to the maximal timestamp among the set of timestamps it received.

2) Second, \( p \) sends the new timestamp and the associated current state to each member of at least one write quorum of the current configuration. In case of a write operation (put/delete/update), the new state corresponds to all key-value pairs to write and the new timestamp corresponds to a strictly greater timestamp than the highest one received in step (1). In case of a read operation (get), the new state corresponds to the most up to date one and the timestamp corresponds to the associated one, i.e., the largest timestamp seen during step (1). The phase is complete once \( p \) receives an acknowledgment from all members of a write quorum indicating that they have successfully updated their local copy of the state and timestamp.

During these two steps an operation can learn about an ongoing reconfiguration, in which case the operation may have to contact a read and a write quorum of the new configuration as well without being blocked, hence avoiding service disruption. (Reconfigurations are not fast enough so that an infinite number of them can delay the reads/writes.)

For some reads to be optimal [14], we implemented the confirmed timestamp optimization proposed in RDS and similar to Sfw’s [13] to complete a read request immediately after the first step (i.e., one round). This optimization requires that all write quorum members answer to each other in the last exchange of the write operation. When each member gets a response from all other members of its write quorum it can set the tag associated with the written value to confirmed. Next reads can just return in one phase when they receive a maximum tag that is confirmed.

C. Implementation and optimizations

As opposed to other protocols, our reconfiguration component does not require that a participating server forwards the request to the tentative leader. Instead we assume that any server can issue requests directly to a quorum of servers, acting as a candidate leader. This translates into trading the optimization that consists in getting rid of the prepare phase in case the leader remains unchanged [29] by the optimization of getting rid (in all cases) of the messages between the contacted server and the leader.

Due to practical motivations, we implemented a series of optimizations on top of existing algorithms. RDS does not assume reliable communication channel but we used TCP that offers guarantees (e.g., FIFO delivery, flow control). Our application does not retransmit messages and we did not implement a costly continuous all-to-all gossip-based protocol, as it is the case in RDS, precisely because TCP handles retransmission of packets whose loss is detected at the layer 4. This effort appeared valuable to guarantee Rollup scalability, as we show in Section V-A.

We also noticed that membership was stable enough to avoid closing and reopening TCP connections between
servers. In our case, each server keeps a pool of TCP connections with other servers open. This avoids unnecessary handshakes and maintains ideal window sizes.

IV. ROBUSTNESS AND CORRECTNESS

In this section, we discuss the fault-tolerance, correctness and liveness of Rollup.

A. Grid biquorum system

To tolerate failures Rollup relies on a specific biquorum system: it is a variant of the grid quorum system \( \mathcal{G} \) that has proven optimal load \( \mathcal{c} \) where each row and each column is a separate quorum. This biquorum system is also more scalable than traditional majorities, as the size \( q \) of its quorums goes up to \( \sqrt{n} \) and can survive up to \( (q - 1)^2 \) failures as opposed to majorities \( \mathcal{M} \) that cannot tolerate \( \left\lfloor \frac{n}{4} \right\rfloor \) failures. (Scalability is evaluated in Sections V-A and V-D.) The drawback, though, is that the biquorum system can suffer failures on \( q \) specifically selected servers as described below.

More precisely, the current implementation of Rollup, running on all \( n \) participants, starts by selecting among the \( n \) servers the largest lower square value \( m = q^2 \leq n \) to build a biquorum system represented as a grid. (If \( n < 4 \) the configuration falls back to majorities.) Figure 3 represents a system of \( n = 11 \) servers with a quorum system of \( m = 9 \) participants including the first three columns of servers. The membership of the biquorum actually gets replicated at all servers to persist; a new server entering the system or recovering needs this information to bootstrap, even before it gets included in the list of participants through reconfiguration. Note that the oracle helps the node retrieve this information.

For the sake of fault tolerance, Rollup replicates data related to the service (e.g., key-value pairs stored via write requests) on a column of this grid, also called a write quorum, of size \( q = \sqrt{m} \leq \sqrt{n} \) servers and contacts a row, also called a read quorum, to fetch the most up-to-date pairs as we described in Section III-B.

Worth noting that the reconfiguration needs the responses of all members of at least one read and at least one write quorum to terminate, hence so does Rollup. Since the example above has a biquorum system defined as a grid of 9 servers, the failures of 3 servers in this case can prevent Rollup from terminating (e.g., if a whole diagonal of the grid fails, no pair of read and write quorum would respond). Reconfiguration actually solves this problem by allowing to reconfigure the current quorum system before the 3rd server fails, hence coping potentially with an unbounded number of failures.

B. Correctness and liveness

The proof of atomicity of the key-value store service relies on the order defined by the timestamps used during reads and writes. Recall that such a timestamp consists of a counter and the tie-breaker identifier of the server that received the client request. We refer to the timestamp of a read operation \( t_s \) as the largest timestamp observed at the end of the first phase of the read operation, we refer to the timestamp of a write operation \( t_w \) as the incremented timestamp propagated during the second phase of the write.

These timestamps define a partial order on the operations executed so that an operation \( \pi \prec \pi' \) if either \( \pi \) is a write, \( \pi' \) is a read and \( t_s \pi = t_s \pi' \), or \( t_s \pi < t_s \pi' \). This allows us to prove the three last properties of atomicity \( \mathcal{A} \) (given that the first property is induced by the others). First, given the intersection between quorums, if the response of \( \pi \) precedes the invocation of \( \pi' \) then we cannot have \( t_s \pi' \not< t_s \pi \). Second, given that writes always increment the timestamp, if \( \pi \) is a write operation and \( \pi' \) is any operation, then either \( \pi' \prec \pi \) or \( \pi \prec \pi' \). Third, given the specification of a storage, a read \( \pi \) always returns the latest value written or the initial value (if no such write exists). The extension of the proof to the case where two configurations/versions are active at the same time relies on the assumptions that a quorum of the old configuration is always notified of the new configuration/version, and that an operation learning the existence of a new configuration/version restarts and uses the current version and quorums of the new configuration. Like configurations \( \mathcal{C} \), versions are totally ordered given that Paxos solves validity and agreement.

The proof of liveness requires additional assumptions that are not needed for correctness. First, Rollup can only terminate if the servers to be upgraded are active, but if Rollup does not get a response after a timeout it can safely restart by excluding this node. Eventually if a read and a write quorum are responsive, the system will be upgraded to a final configuration comprising only upgraded servers. Second, we mentioned that the maximal number of failures that can occur during a reconfiguration is bounded by \( \sqrt{n - T} \) – 1. Given that Rollup has to upgrade at least one server at a time, the number of participants is \( n - 1 \). Rollup relies on the grid biquorum system with at least one read and one write quorum that do not fail, meaning that Rollup can only fail with \( \frac{\sqrt{n - T}}{2} \) failures affecting one server of each row and one server of each column of the grid. Third note that if these failures are not frequent enough to occur within the same reconfiguration lifespan then a reconfiguration can successfully update the group of participants to reset the number of failures to 0, provided that enough nodes are available.
Table 1. Performance result of the service in terms of reads (left) and writes (right) during Rollup (naive) and size-aware Rollup (right) with the cloud infrastructure comprising 60 instances. The service operations terminate provided that the read/write operations do not get redirected to an infinite sequence of new configurations. The server receiving the request cannot fail too frequently as some operations restart to eventually complete. As the communication is reliable, messages between non-crashed servers get eventually delivered, which ensures termination.

<table>
<thead>
<tr>
<th>Number of upgraded servers</th>
<th>Rollup (naive)</th>
<th>Rollup (size-aware)</th>
<th>Recon</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The figure shows that recon latency is slower than key-value store operations (8-24ms), so it is impossible for Rollup to force reads/writes to restart infinitely often. As the communication is reliable, messages between non-crashed servers get eventually delivered, which ensures termination.

V. EXPERIMENTAL ANALYSIS

To evaluate Rollup, we deployed it on a physical cluster and on Amazon Elastic Compute Cloud. The cluster consists of 35 physical machines of 1 GHz VIA C3 processor with 512 MB of RAM, a 20 GB local hard disk and 2 Ethernet ports running Linux Debian 3.2.51-1 and Java 1.7.0.51 with HotSpot build 24.51-b03. The cloud infrastructure comprises 60 t2.micro and t2.small virtual machines running Ubuntu Server 14.04 LTS with 1 virtual CPU and 1 (micro) to 2 (small) GB of memory.

A. Avoiding traffic congestion

To measure the performance of Rollup, we run it on different sizes of the distributed systems. As our quorum systems are automatically set to \( \lfloor \sqrt{n} \rfloor \) we chose square numbers for \( n \) between 4 and 25. (We omitted \( n = 1 \) as a quorum system of size 1 is a singleton prone to a single point of failure.) We measured the performance of two versions of Rollup in terms of the latency of the rolling upgrade of the whole system. “Rollup (naive)” is the time taken by Rollup to upgrade the system with a granularity set to \( g = 1 \) regardless of the system size \( n \). “Rollup (size-aware)” is the time taken by Rollup to reconfigure the system with a granularity \( g = \sqrt{n} \) so that the latency results are given for \( \langle n,g \rangle \) pairs of \( \langle 4,2 \rangle, \langle 9,3 \rangle, \langle 16,4 \rangle, \langle 25,5 \rangle \).

Figure 4(a) depicts the latency of the two Rollup versions in addition to the latency of a reconfiguration for the baseline. These two latencies are measured by the client machine as the time between its invocation of a recon or rollup command to one server and the time the response is received from the server at the client. The reconfiguration latency is reasonably low, from 45ms for 4 servers to 249ms for 25 servers. Interestingly, the naive Rollup does not scale up to 25 servers as its latency is superlinear in the amount of servers, however, the latency of the size-aware Rollup increases only linearly, actually upgrading 25 servers twice faster.

This difference is explained by traffic congestion: for the sake of fault tolerance, Rollup sends messages to replicated quorums even though responses from a single quorum are necessary. At \( n = 25 \) servers, this corresponds to having two read quorums and two write quorums (four quorums) exchanging with two read quorums and two write quorums (four quorums) during the second phase of a reconfiguration and four quorums exchanging with four quorums of the current configuration and four quorums of the new configuration in the third phase. As each quorum is of size 5 and 4 in the configurations of this upgrade, and each read quorum intersects each write quorum at one node, two read and write quorums contain 18 and 14 distinct
nodes in the corresponding configurations. Hence, the first reconfiguration from \( n \) to \( n - g \) servers produces 10 messages in the prepare phase, \( 18 + 18^2 \) in the propose phase and \( 18 \times (18 + 14) \) in the propagate phase for a total of 928 messages. Similarly, the \( \left\lceil \frac{2n}{g} \right\rceil \) following reconfigurations from \( n - g \) servers to \( n - g \) other servers incur 610 messages, and the last reconfiguration from \( n - g \) to \( n \) servers incur 666 messages. We thus have 16234 messages if \( g = 1 \) but only 4034 if \( g = 5 \).

B. Local upgrade and granularity effects

To better evaluate the impact of the granularity on the time it takes to complete Rollup, we experimented Rollup on 16 servers with a granularity varying among \{1,2,4,8,12\}. In addition, we measured the cumulative time taken to download the new version of Rollup as a jar of 18KiB located on a remote server.

Figure 4(b) depicts the total time for Rollup to complete and the cumulative time of the local upgrades on 25 servers when the granularity increases from 1 to 12. With granularity 1, all local upgrades must execute one after the other hence leading to a relatively high cumulative upgrade time that consists of the sum of each local upgrade time. With a larger granularity, like 12, there are 12 upgrades that execute in parallel hence limiting the cumulative upgrade time of all servers. On the one hand, we can clearly see that the local upgrade time plays a key role in the Rollup latency, which explains why a too low granularity makes the process so long. On the other hand, it is clear that the gain of parallelising local upgrade tends towards some limit induced by Amdahl’s law (note the non-linear x-axis) so that it does not pay off having a too large granularity either. A right parameter in this example seems to be \( g = 4 \) as it reduces Rollup latency substantially, yet it lets most of the servers (12 of them) share the service load.

C. How Rollup impacts the service usage

To evaluate the impact of Rollup on the performance of the underlying service, we compared the service latency with and without Rollup running. (Section V-E shows non-disruption.) The service latency is measured as the interval of time between the invocation of a read/write request issued by a client machine and the time of reception by the client of the response from the server.

Figure 4(c) reports the average latency of reads and average latency of writes when Rollup executes and when Rollup does not execute. The time taken for reading without Rollup is impressively low and remains low despite system growth. The difference with write is explained by the confirmed timestamp optimization (Section III-B) that allows read operations to complete after a single message exchange with a read quorum. As expected the time taken by a read or a write to complete is larger when a Rollup instance is concurrently running. This is explained by the fact that the network resource is limited, and Rollup and the storage service use different messages while sharing the same communication channels between quorums of participants. We also observe that the difference increases with the size of the system: at \( n = 4 \) servers, the average latency is 8ms without Rollup and 11ms with Rollup (137% increase) whereas at \( n = 25 \) the average latency is 13ms without Rollup and 24ms during Rollup (180% increase). This observation tends to confirm our previous thoughts on the traffic congestion. While this slowdown is noticeable, it remains negligible compared to the existing rolling upgrade alternatives that stop treating requests for 1 to 2 minutes [40].

D. Scalability

To evaluate our solution at larger scale in the cloud, we conducted performance experiments in Amazon EC2. To see whether our solution was greedy in resources we deployed Rollup on 50 t2.micro instances and ran it on a biqorum system of \( n = 49 \) servers and 1 separate client. (Note that the client load is quite lightweight compared to the message exchanges involved on the server side by the quorum members as quantified in Section V-A.) Hence the quorum size as well as the granularity are \( \sqrt{n} = g = 7 \). Despite the minimal resources of micro instances, we observe similar performance results as the ones observed on our cluster. We report the CDF of the latencies for reconfiguration and rolling upgrade in Figure 5.

The slowest latency we have observed over 100 reconfigurations was 1015 milliseconds. This benchmark would trigger one reconfiguration after another to the same server and the maximum latency was observed during the first reconfiguration of one of these sequences due to the TCP slow-start strategy. The mean latency over all these runs was 82.41ms, while the minimum latency was 38ms and the median was 56ms. We also ran 10 rolling upgrades and observed an average latency of 2003.8ms between a minimum of 1652ms and a maximum of 3161ms, confirming that Rollup scales well to 49 servers.

While consensus-based storages often involve too many messages targeting a primary bottleneck to scale to large number of participants [25], we achieve scalability by avoiding consensus for most operations (read/write) and executing it exclusively during (more sporadic) rolling upgrades in a fully distributed way. Note, however, that other solutions exist. For example, the simulation of SQUARE showed scalable performance as it exploits adaptive quorum
system and more frequent but local reconfigurations, however, the model is different as any failure detection triggers a new reconfiguration [21]. Another scalable protocol, called GIARFFE, is especially designed to cope with the lack of scalability of Zookeeper by distributing its tree-based structure into trees [44].

E. Non-disruption

We also evaluated another configuration of 59 virtual machines, with 9 servers running as t2.small instances and 50 clients running as t2.micro instances. Figure 6 depicts the throughput for few minutes as each client would keep sending 16KiB requests with 10% writes and 90% reads to four randomly chosen servers. Each client computes the number of requests served in every slot of 300 milliseconds and the results for all clients are aggregated within periods of 3 seconds (as described in Zookeeper [46]). We can see that the throughput is not as efficient as Zookeeper's obtained on their own cluster also because Zookeeper is weakly consistent and optimized for pipelining while our key-value store is strongly consistent. Rollup was executed twice by reconfiguring but without upgrading any software: at times 34 seconds and 1mn30 with parameter $g = b = 3$. Even though it is clear that Rollup impacts the key-value store throughput due to the large number of messages it produces (as discussed in Section V-A) it does not disrupt the key-value store service. Actually, reads and writes may simply have to access the quorums of two configurations during a reconfiguration instead of one. Note that this is in contrast with Zookeeper that experiences a throughput of 0 in case of primary reconfiguration [46].

VI. THE PRIMARY-BASED RECONFIGURATION

In this section, we illustrate empirically why rolling upgrade suffers more from the primary-based design than our biquorum-based design. We used Zookeeper’s reconfiguration [46] as the baseline as this protocol is at the heart of recent proposals like Raft [39] and the CoreOS etcd’s membership management [42] used by academics and industrials.

A. Zookeeper’s primary-based reconfiguration

Zookeeper offers a reconfiguration service that allows an operator to adapt the number of active servers participating in the service [46]. In order to totally order the subsequently installed configurations, Zookeeper uses an atomic broadcast protocol called Zab [27] whereas Rollup uses the Paxos consensus protocol [28]. We evaluated the performance of the Zookeeper reconfiguration [46] that is part of the public release Zookeeper-107 [47] and found that its latency can be of different orders of magnitude depending on the server getting reconfigured. More precisely, a reconfiguration adding or removing a follower takes about a hundred of milliseconds to complete whereas a reconfiguration removing a primary takes about a couple of seconds to complete. This significant difference makes it hard for a user of the reconfiguration service who ignores the identity of the primary to predict the performance of a reconfiguration.

Figure 7 uses log-scale on the y-axis to depict this variation of times taken to reconfigure an initially empty storage service. To this end, it reports the time between the invocation of the reconfiguration request and the response to a write (create) request immediately following the reconfiguration response. We choose a write operation as Zookeeper’s reads are weakly consistent by default. Note that the write latency we generally observed on Zookeeper (but omitted here) is around 10 milliseconds which is negligible compared to the 7 seconds taken sometimes to reconfigure and write. Interestingly, we observed that a write request would generally take longer when executed after a reconfiguration. The reason is that the primary acknowledges the reconfiguration before its completion while the write has to execute on an up-to-date configuration of
the system.

The reason for the primary (or leader) removal to be substantially longer than the follower removal is that the primary plays multiple important roles in Zookeeper. Zookeeper follows a primary/backup strategy by centralizing the data and configuration information at the primary. As a result, removing the Zookeeper primary requires to elect a new primary that can take over this information. In particular, it turns out that the connection information is also maintained at the primary, this causes the clients to disconnect when the primary gets replaced. Interestingly, the clients do not have to be connected to the primary to be affected by the primary reconfiguration. Recent information even indicates that the write service has to be disrupted when the primary is removed. Because no two primaries can coexist and electing a primary takes time, our experiments confirm that removing a primary through reconfiguration incurs a similar performance drop to the one due to primary failure.

B. Quorum vs. primary

An interesting advantage of Rollup is that its primary only proposes new configurations to be voted upon, it does not own the responsibility of making the data persistent and keeping track of the current configurations and its role stops once the reconfiguration is complete. As the data and configuration information is distributed in Rollup, the responsibility is not owned by a single server but shared by multiple ones. This leads to a reconfiguration whose performance is comparable to Zookeeper’s reconfiguration of a follower (cf. Figure 8).

 VII. TIME COMPLEXITY

This section complements the experimental evaluation of the duration of primary-based reconfiguration by computing the expected duration of the primary-based rolling upgrade.

The rolling upgrade process consists of reconfiguring batches at a time until all servers are upgraded. For the sake of simplicity in the analysis, let \( n = b \) be the number of servers. A primary-based approach can only have at most one primary and thus elect a new primary (or a leader) right after the reconfiguration process excludes the primary from the set of active participants. Assume that initially and upon new primary elections, the primary is always taken uniformly at random among the participants.

Let \( X \) be the random variable representing the number of reconfigured batches containing a primary during the rolling upgrade process. On \( b \) batches, we are interested in the duration \( D_{\text{Primary}} \) of the rolling upgrade. Let \( d_P \) denote the time needed to upgrade a batch with a primary (or leader) and \( d_B \) the time needed to upgrade a batch with only backups (or followers).

\[
D_{\text{Primary}}(b) = d_PX + d_B(b - X).
\]

\[
E[D_{\text{Primary}}(b)] = E[d_PX + d_B(b - X)]
\]

\[
= d_PE[X] + d_B(b - E[X])
\]

The process can be described as a discrete time Markov chain whose states are the potential batches with a primary, 1 to \( b \), plus an absorbing state \( s \) as depicted in Figure 9.

A transition is labeled with the probability of one node from the set of active participants. Assume that initially and upon new primary elections, the primary is always taken uniformly at random among the participants.

Let \( X \) be the random variable representing the number of reconfigured batches containing a primary during the rolling upgrade process. On \( b \) batches, we are interested in the duration \( D_{\text{Primary}} \) of the rolling upgrade. Let \( d_P \) denote the time needed to upgrade a batch with a primary (or leader) and \( d_B \) the time needed to upgrade a batch with only backups (or followers).

The expected duration of the rolling upgrade using a primary-based approach, like etcd, we assume that the necessary primary/leader election selects a node among the active participants uniformly at random. CoreOS etcd uses the Raft consensus protocol similar to Paxos except that it requires a unique primary through which all requests must be forwarded.

A. Markov model for expected durations

The rolling upgrade process consists of reconfiguring batches at a time until all \( n \) servers are upgraded. For the sake of simplicity in the analysis, let \( n = b \) be a multiple of \( g \). A primary-based approach can only have at most one primary and thus elect a new primary (or a leader) right after the reconfiguration process excludes the primary from the set of active participants. Assume that initially and upon new primary elections, the primary is always taken uniformly at random among the participants.

Let \( X \) be the random variable representing the number of reconfigured batches containing a primary during the rolling upgrade process. On \( b \) batches, we are interested in the duration \( D_{\text{Primary}} \) of the rolling upgrade. Let \( d_P \) denote the time needed to upgrade a batch with a primary (or leader) and \( d_B \) the time needed to upgrade a batch with only backups (or followers).

\[
D_{\text{Primary}}(b) = d_PX + d_B(b - X).
\]

\[
E[D_{\text{Primary}}(b)] = E[d_PX + d_B(b - X)]
\]

\[
= d_PE[X] + d_B(b - E[X])
\]

The process can be described as a discrete time Markov chain whose states are the potential batches with a primary, 1 to \( b \), plus an absorbing state \( s \) as depicted in Figure 9.

A transition is labeled with the probability of one node from the set of active participants. Assume that initially and upon new primary elections, the primary is always taken uniformly at random among the participants.

Let \( X \) be the random variable representing the number of reconfigured batches containing a primary during the rolling upgrade process. On \( b \) batches, we are interested in the duration \( D_{\text{Primary}} \) of the rolling upgrade. Let \( d_P \) denote the time needed to upgrade a batch with a primary (or leader) and \( d_B \) the time needed to upgrade a batch with only backups (or followers).

\[
D_{\text{Primary}}(b) = d_PX + d_B(b - X).
\]

\[
E[D_{\text{Primary}}(b)] = E[d_PX + d_B(b - X)]
\]

\[
= d_PE[X] + d_B(b - E[X])
\]
batches with probability \( \frac{1}{n-1} \) or to anyone of the previously upgraded batches (represented by a single transition to the sink state). \( X \) actually represents the number of times a primary was upgraded and its value corresponds to the number of states visited before absorption in the Markov process.

The resulting transition matrix is thus strictly upper triangular. As an example, if we consider the case where \( n = 25 \) and \( b = g = 5 \) we have \( E[X] = 1.6641406 \).

As we reported in the previous section (cf. Figure 8), the time \( d_p \) needed to upgrade a batch with the primary and the time \( d_B \) needed to upgrade a batch of backups are, respectively, \( d_p = 2008 \) and \( d_B = 124.5 \) milliseconds on average. By contrast, Rollup achieves any reconfiguration in \( d = 117 \) milliseconds on average.

Finally, we can deduce from Equation II that the duration in this case is

\[
E[D_{Primary}(5)] = 2008 \times 1.6641406 + 124.5 \times 3.358594 = 3714 \text{ milliseconds.}
\]

In comparison, the duration to execute Rollup is

\[
E[D_{Rollup}(5)] = 117 \times 5 = 585 \text{ milliseconds, which is more than } 6 \times \text{ faster than a primary-based rolling upgrade.}
\]

**B. Rolling upgrade at various scales**

For a given granularity and system size, we can derive the time of a rolling upgrade in expectation. Using the time of primary reconfiguration, backup reconfiguration and Rollup reconfiguration as given in Figure 8 we report the expected duration of rolling upgrade whether it is based on primary/backup or on Paxos. We vary the granularity as a portion \( p \) of all \( n \) nodes and convert it into \( max(1, \lfloor np \rfloor) \) nodes so that a rolling upgrade is done in one reconfiguration only when \( p = 100\% \).

Figure 10 simulates the expected durations of a rolling upgrade based on primary/backup reconfiguration and a rolling upgrade based on our reconfiguration (Paxos-based). The expected duration depends on the chosen granularity expressed as a percentage of the system size. In both cases, the expected duration decreases as the granularity increases. As expected, the performance gain of Rollup over the primary-backup rolling upgrade is more visible when the granularity is large. This speedup factor varies over all the configurations we have tested, from 3 and 11.

**Fig. 10.** Comparison of the expected duration of a Primary-based rolling upgrade against our Paxos-based rolling upgrade (without counting the local upgrade time in any scenario)

For example, the speedup factor for the case \( n = 4 \) and \( g = 1 \) is 7.8, indicating that Rollup has a significant performance advantage over a primary-based rolling upgrade.

**VIII. RELATED WORK**

Rolling upgrade is not a novel research challenge as it had impact in distributed computing fourteen years ago [6]. There are few automated solutions for rolling upgrade as most production tools [41], [47], [43], [34] still require manual updates to check for bugs in order to prevent various software failures [12] mostly induced by mixed version [48].

Raft [39] describes a protocol to change the cluster membership using a special transitional configuration, called joint consensus, that consists of the old and the new configurations. If the leader does not belong to the new configuration, then it Maintains this joint consensus until it commits the new configuration. At this point a new leader election is needed before other requests can be served. The same algorithm is used in CoreOS etcd [42]. Note that write requests are forcefully forwarded to the leader both in CoreOS etcd and in Zookeeper’s reconfiguration [46]. Our service requests do not need any leader, hence the service is never disrupted despite \( \lfloor \sqrt{n} \rfloor - 1 \) hardware or software crash failures and can tolerate up to \( (\lfloor \sqrt{n} \rfloor - 1)^2 \) failures between upgrades.

Data Guard [40] uses a logical standby database that gets upgraded from version \( v_0 \) to \( v_1 \) while the primary provides the database service version \( v_0 \). When the standby upgrade is deemed successful, Data Guard resynchronizes the standby with the primary and executes a switchover.

\[^{4}https://coreos.com/docs/cluster-management/scaling/etcdd-optimal-cluster-size/\]
for the standby to start providing the version $v_1$ of the service. The database on the original primary gets upgraded while the standby server provides the service. When the upgrade of the original primary is complete, Data Guard resynchronizes the two databases both operating version $v_1$. A final optional switchover completes the upgrade. The downtime is thus reduced from 3 hours to the time it takes Data Guard to execute the switchover: 1 minute (or 2 minutes if the second switchover is performed). While our solution also synchronizes an in-memory storage service, it does not experience downtime.

Imago \cite{11} proposes an atomic upgrade for a multi-tier architecture as an alternative to rolling upgrade. This upgrade protocol was shown effective in upgrading a complex service like Wikipedia, that involves MediaWiki that requires MySQL and PHP that requires, in turn, Apache, however, the upgrade cannot terminate if the update request rate is higher than the data transfer as the transfer will not catch up with the latest data store state \cite{13}. To ensure termination, Imago disrupts write requests during transfer and switchover either by marking the data tier as read-only or by simply blocking all incoming write requests. This disruption is necessary for Imago to transfer the data from one copy to another. The switchover is only executed after the two copies are perfectly synchronized. When the switchover completes, writes start being served again.

Reconfiguring a distributed system providing a replicated service, like a storage, is an interesting problem \cite{11} addressed by two types of solutions, whether consensus is used. In theory it is clear that consensus cannot be solved in an asynchronous environment even in the presence of software faults \cite{16}, however, Paxos is generally used to ensure validity and agreement anytime while guaranteeing termination if at least a majority of servers do not fail and once the network becomes reliable and delivers messages \cite{28}. The consensus-less solutions require similar assumptions to guarantee liveness but usually require service requests to traverse the directed acyclic graph of partially ordered configurations before completing \cite{2}.

Thanks to its ability to totally order configurations \cite{45}, consensus-based reconfiguration has attracted lots of attention \cite{33, 36, 35, 20, 9, 10, 46}. however, it usually requires several message delays to achieve basic operations or garbage collect old configurations \cite{19}. Other solutions target reconfiguration in the more general context of Byzantine failures \cite{35}. The Reconfigurable Distributed Storage (RDS) was formally proved correct \cite{10} using the IOA formalism \cite{32, 17} but was never experimented on load-optimal quorum systems. Variants of this algorithm, including DSR \cite{5}, uses a majority for simplicity reasons but maintains the virtual synchrony property. As far as we know, Rollup is the only way to upgrade without disruption.

A key motivation for designing cluster membership protocols in a centralized way was the Paxos anomaly \cite{4, 25}: Paxos decides on individual proposed values, potentially violating dependencies between values proposed by the same requester. In Zookeeper \cite{25}, update operations can partially update a tree structure of znodes hence clients can request to create a znode and request, immediately after, to create a child node of this parent znode. If these two requests are concurrent, Paxos can decide upon one of the two operations, say the creation of the child node, and discards the other, say the creation of the parent znode. This would lead to an incorrect state where the child node cannot be created because its parent znode does not exist. Rollup does not suffer from the Paxos anomaly: it does not use consensus to order write requests as opposed to Paxos-based replicated state machine approaches, like SMART \cite{31}. Hence, if two updates occur on the same key-value pair concurrently, then they will both succeed in an order that depends on their timestamp, but none of them will be discarded.

IX. CONCLUSION

The main appeal of rolling upgrade is to provide availability during upgrades. Existing consensus-based techniques use a separate key-value store to manage cluster membership but cannot be reconfigured without disrupting their service. Our contribution is Rollup, a non-disruptive consensus-based rolling upgrade protocol. Rollup builds upon a large body of theoretical results but is, as far as we know, the first system to provide a reconfiguration building block that reaches proven optimality in terms of server load and service latency. We evaluated Rollup on two different distributed environments. Rollup proved efficient as its reconfiguration is non-disruptive and its service scalable. In particular its reconfiguration protocol is up to one order of magnitude faster than Zookeeper’s. Finally, our theoretical analysis shows that a quorum-based rolling upgrade can speedup a primary-based one by $6 \times$.

Acknowledgements

We wish to thank Michael Troup for bringing to our attention the delay between the Zookeeper reconfiguration acknowledgment and the time subsequent writes can complete and Guillaume Smith for suggesting the primary position as the Markov chain state. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References


