Rollup: Non-Disruptive Rolling Upgrade with Fast Consensus-Based Dynamic Reconfigurations

Vincent Gramoli, Len Bass, Alan Fekete, Daniel Sun

***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 CoreOS etcd, 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 induces disruption.**

We propose Rollup, a non-disruptive rolling upgrade protocol with a fast consensus-based reconfiguration. Rollup relies on a candidate leader only for the reconfiguration and scalable biquorums for service requests. While Rollup implements a non-disruptive cluster membership protocol, it does not offer a full-fledged coordination service. We analyzed Rollup theoretically and experimentally on an isolated network of 26 physical machines and an Amazon EC2 cluster of 59 virtual machines. Our results show an 8-fold speedup compared to a rolling upgrade based on a primary for reconfiguration.

Keywords: quorum; Paxos; online upgrade; cluster management; membership change; active replication

I. Introduction

Today, major service providers use continuous deployment, for example modifications at Facebook are deployed multiple times a day [17]. To avoid disruption, the service state is replicated in a distributed system and a *rolling upgrade* progressively upgrades one batch of machines at a time. A rolling upgrade requires subsequent reconfiguration instances, each excluding a small batch 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[1], 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 [32] consensus algorithm 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 [46] the tendency had been to switch to an alternative algorithm where requests are centralized at a primary. Zab, a primary-based atomic broadcast protocol [31] was used in Zookeeper [29], a distributed coordination service. Raft [46] reused the centralization concept of Zookeeper to solve consensus, a problem known to be equivalent to atomic broadcast [26]. The resulting simplification led to various implementations of Raft. The drawback is that these protocols rely heavily on the availability of their primary.

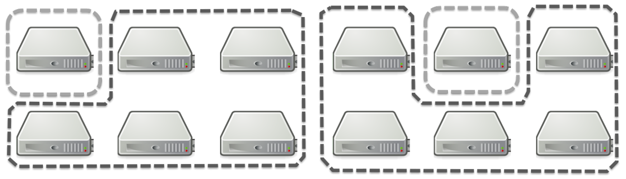

(a) The top left server gets upgraded (b) The second server gets upgraded

Fig. 1. System of $n = 6$ servers after the 1st reconfiguration (a) and 2nd reconfiguration (b) of a rolling upgrade with granularity $g = 1$ (i.e., $n - g = 5$ participants)

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 [54] forwards both reconfiguration and store requests to the primary, hence benefiting from pipelining. CoreOS etcd [50] offers a similar reconfigurable key-value store. Due to their centralization, these services are badly suited for rolling upgrade where all servers, including the primary, must be upgraded. Excluding this primary from the set of 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. These protocols are badly suited for rolling upgrade, because a rolling upgrade requires to upgrade "all" servers.

In this paper, we propose *Rollup*, the first non-disruptive rolling upgrade protocol of a stateful service, in that clients are served regardless of the speed of a concurrent upgrade. It offers a new membership change 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 [44]. 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 [36], hence allowing to have

Vincent Gramoli is with NICTA and University of Sydney, Australia.
Len Bass is with NICTA, Australia.
Alan Fekete is with NICTA and University of Sydney, Australia.
Daniel Sun is with NICTA, Australia.

[1]http://www.computerweekly.com/blogs/open-source-insider/2015/02/coreos-linux-its-how-google-facebook-and-twitter-run-at-scale.html

pairs of operations writing on disjoint sets of servers (as opposed to majorities or primary/backup quorums) while still guaranteeing strong consistency (linearizability [28]). While Rollup builds upon known lower-bounds [44], [16] and formally proven distributed algorithms, like RDS [11] and Paxos [32], its general contribution is to bridge the gap between a long body of theoretical results [44], [32], [40], [11] and recent system achievements [50], [30], [46] 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 18 KiB. 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 of the primary 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. Zookeeper is as far as we know the only system whose cluster membership change was published, available online and documented [54]. The efficiency of Zookeeper's reconfiguration is however tied to the steadiness of the primary that must be altered during the course of a rolling upgrade. 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.[2]

It is important to understand the limitations of Rollup. Even though Rollup implements a non-disruptive upgrade of a key-value store service, it remains unclear whether one can implement a non-disruptive upgrade of a full-fledged coordination service. In particular, Zookeeper offers ephemeral nodes for coordination through distributed locking while Rollup does not offer distributed locking, multi-object transaction or compare-and-swap (CAS) primitives. One could naively implement these extra services using the consensus protocol at the heart of Rollup, however, it is clear that the upgrade would disrupt these. In particular, a CAS object may even need all its reads, writes and CASes to be totally ordered with respect to reconfigurations.

Finally, we analyzed theoretically the cost of executing a primary-based rolling upgrade based on CoreOS etcd at large-scale by modelling the rolling upgrade as a discrete time Markov chain. This analysis indicates that using Rollup to upgrade a distributed system of 4 to 100 nodes with a granularity from 10% to 100% of the participants leads to an average speedup of 10×. This result is when the primary/leader election ends up selecting one of the available nodes uniformly at random, making the expected number of upgraded primaries greater than 1. We also show that if one can ensure that the number of primaries upgraded is exactly 1, then Rollup is still 8× faster than a primary-based rolling upgrade. 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. Section IV discusses its fault-tolerance, proves its correctness and analyzes its complexity. 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 servers, with unique identifiers, potentially running services and communicating through message passing. Each machine may fail by crashing at which point it will stop acting and after a recovery a machine is considered a new machine. Rollup uses *configurations* and replaces a configuration by another by installing a new configuration before discarding the old one so that there can be two configurations used, so called *active*, at the same time. For liveness, we assume that in each active configuration there are no more than $\lfloor \sqrt{n} \rfloor - 1$ failures if $n \geq 4$ and no more than $\lceil \frac{n}{2} \rceil - 1$ failures otherwise.

The dynamic set of servers P_t operating the service at a given time t are referred to as the *participants*. Although S nodes are involved in the rolling upgrade, only $P_t \leq S$ of them serve the service, the remaining $S \setminus P_t$ do not participate as they get upgraded. We assume the network communication to be asynchronous and messages can be lost but no messages are forged or modified. We also assume that an external oracle provides the client with the address of the participants where the service is available, as achieved in practice via IP anycast or DNS.

Rollup upgrades the distributed service from an old version to a new version and we assume that all servers know which one is the most recent version when it obtains it. Depending on its version, the service exports a specific interface to its clients. We do not require backward compatibility of versions: newer version does not need to offer a superset of the features offered by the older version. Instead, we assume that a client issuing a *wrong request* by requesting a feature no longer or not yet supported by the current version, receives from the server a response indicating that the client version should be changed. While these wrong requests are not served, this exchange is necessary for the client to catch up with the current version as we do not require the clients to be active for receiving version upgrade notifications. To upgrade a multi-tier architecture where servers of one tier are the clients of another tier, all servers should participate in the same Rollup instance to avoid disruption.

[2]https://coreos.com/docs/cluster-management/scaling/etcd-optimal-cluster-size/.

Algorithm 1 The Rollup algorithm at server i

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

8: rollup(v, g)_i:                      ▷ rolling upgrade to v with granularity g
9:   cinit ← ccurr                                         ▷ store initial config
10:  upgd ← ∅                                            ▷ reset upgraded set
11:  for all j ∈ S do                                         ▷ for each server
12:    toupg = toupg ∪ {j}                                 ▷ prepare for upgrade
13:    if |toupg| ≥ g ∨ |upgd ∪ toupg| = |S| then          ▷ run batch upgrade
14:      cnext ← S \ toupg                                 ▷ make room for upgrade
15:      recon(ccurr, cnext)                                   ▷ change config.
16:      upg(toupg, v)                 ▷ upgrade unused servers to v in parallel
17:      ccurr ← cnext                                     ▷ move to next config
18:      upgd ← upgd ∪ toupg                              ▷ add upgraded servers
19:      toupg ← ∅                                     ▷ no more server to upgrade
20:  recon(ccurr, cinit)                               ▷ go back to initial config.
```

We consider a replicated protocol, where the same set of information is replicated at multiple servers, also called *replicas* as defined by the configuration. The replicas of a configuration are organised into *quorums* that are mutually intersecting sets of replicas. The set of quorums is called the *quorum system*. We give a more precise definition of the particular kind of quorum system that we implemented in Section IV. In Section IV-C, we present additional assumptions about partial synchrony that are necessary to analyze the time complexity of Rollup.

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 (lines 15 and 20): 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 of upgrading progressively a service by acting on one *batch* of 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. There is a tradeoff between high values of g that speedup the rolling upgrade and low values of g that maximize system capacity, and it is reasonable to keep $g = n/10$ or $g = O(\sqrt{n})$.

Rollup upgrades all servers present in the system in $\lceil \frac{n}{g} \rceil$ *runs*. Each run, whose code appears lines 13–19, starts when the amount of servers to be upgraded reaches the granularity threshold ($|toupg| \geq g$) or when there are few servers left non-upgraded ($|upgd \cup toupg| = |S|$). 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 set of participants 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 next run: the current configuration and the batch of upgraded servers are updated, and the batch of servers to upgrade is reset (lines 17–19).

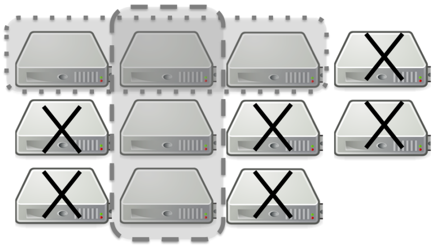


Fig. 2. A system of $n = 11$ servers tolerating 6 failures: a biquorum system of the 9 nodes on the left has a read quorum (dotted line) and a write quorum (dashed line) available

A. Reconfiguration component

We now take a closer look at the recon component of each Rollup run. This is a distributed reconfiguration similar to the RDS algorithm [11]. 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 a set of participants and is represented as a *biquorum system* [36], with two types of quorums (or sets of servers) such that any quorum of one type intersects all quorums of the other type as depicted in Figure 2. 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 3 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 [32], the protocol executes in three phases, each corresponding to a message exchange involving some quorums (defined in Section IV-A):

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

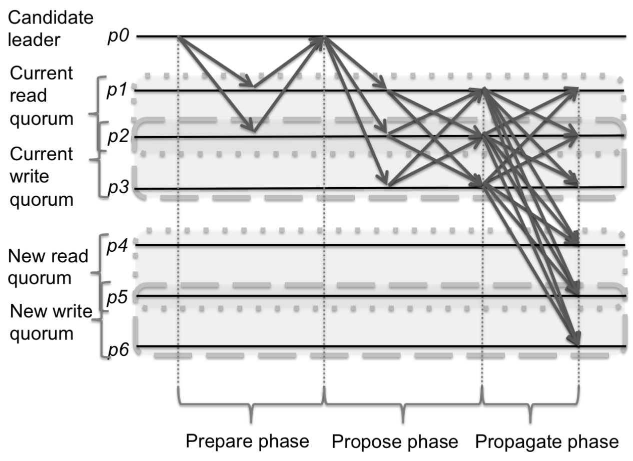


Fig. 3. The 3-phase reconfiguration protocol at the heart of each Rollup run (recon request and acknowledgement as well as duplicated messages sent to additional quorums for the sake of fault tolerance are omitted for clarity): (1) upon request reception, the leader prepares by exchanging with a read quorum, (2) it proposes a new configuration to a read and a write quorum and, if voted, (3) these quorums propagate to a read and a write quorums of both the old configuration and the freshly decided configuration

with the largest ballot among the ones he received and the one he prepared, each quorum member either abstains (by sending the largest ballot number it has already voted for) or votes, which prevents it from voting for a smaller ballot later. The vote/abstain decision is sent 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 a 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.)

These 3 phases follow the specification of RDS [11], differences with RDS are listed in Section III-C. 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 with the usual CRUD interface that allows to create (put(k,v)), read (get(k)), update(k,v) and delete(k), with the usual semantics. These operations execute as if they were occurring instantaneously between their invocation and response [28]. The reconfiguration transfers the storage state from the old to the new configuration [27], but the data stored cannot survive a reconfiguration of their format. 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 the timestamp in one or two steps, like the emulation of registers in message-passing [5], [39], except that the timestamp represents the version of the whole key-value store, not one key:

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 either sending the new key-value pairs as well as the associated timestamp it currently knows or informing the client of a newer configuration (if there is one). Once *p* receives the key-value pairs and timestamps 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). Again participants may inform the client about a new configuration if there is one, otherwise they simply acknowledge. 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 a new configuration due to an ongoing reconfiguration, in which case the operation has to restart the step and contact a read and a write quorum of the new configuration. Note that these operations are never blocked and complete regardless of the time it takes to reconfigure, hence avoiding service disruption. Moreover, reconfigurations are not fast enough so that an infinite number of them can delay the operations (Section V-A indicates that reconfigurations take 45-249 ms while Section V-C indicates that operations take 8-24 ms).

For some reads to be optimal [16], we implemented the *confirmed timestamp* optimization proposed in RDS and similar to SFW's [20] 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. Subsequent reads can just return in one phase when they receive a maximum timestamp that is confirmed.

C. Implementation and optimizations

We implemented a distributed reconfiguration similar to RDS but with differences: the models are different, the former reconfigures a key-value store while the latter reconfigures a register, the former sends point-to-point messages when needed while the latter uses all-to-all gossip, the former completes in three phases while the latter must forward the request to the leader to complete in two phases.

As opposed to other protocols like RDS our reconfiguration 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 of getting rid of the prepare phase in case the leader remains unchanged [35] 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.

IV. Robustness and Correctness

In this section, we discuss fault-tolerance, show Rollup correct and analyze its time complexity.

A. Grid biquorum system

To tolerate failures Rollup relies on a specific biquorum system: it is a variant of the grid quorum system [9] that has proven optimal load [44] 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 $\sqrt{n}$ as opposed to majorities [32] that grow linearly with n. (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, which makes its failure probability non Condorcet [12], 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 2 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.

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 3^{rd} server fails, hence coping with an unbounded number of failures.

Unlike the failure probability of the grid quorum system, the failure probability of the majority quorum system is known to be Condorcet [49]. In other words, the probability $f(majority)$ that no pairs of majorities intersect, if any node fail with an independent probability $p < \frac{1}{2}$, tends to 0 as the number n of quorum system nodes grows: $\lim_{n\to\infty} f(majority) = 0$. This property is interesting as it measures fault tolerance as the quorum system grows.

Despite these drawbacks, we opted to use the grid quorum system when $n \geq 4$ as it can easily be used as a biquorum system, a concept that does not benefit from majorities. Biquorum systems are an important building block of our work: given that Paxos does not require that quorums of proposers intersect and similarly RDS does not require that read quorums intersect, Rollup does not need its read quorums to intersect, hence allowing some reads to return after only $2\lfloor\sqrt{n}\rfloor$ messages—for comparison, a majority quorum system would require at least $2\lceil\frac{n+1}{2}\rceil$ messages. If $n = 4$, reading from a majority quorum system would require at least 6 messages whereas reading from a grid quorum system may simply need 4 messages.

B. Correctness

The proof of atomicity of the key-value store service relies on the proof that our reconfiguration solves the validity and agreement properties of consensus. Recall first that the timestamp used by each operation consists of a counter and the tie-breaker identifier of the server that received the client request. We refer to the timestamp ts_π of an operation π as the largest timestamp observed at the end of the first phase of the operation π. These timestamps will help us define a partial order on the operations. Below we order operations depending on their timestamps in the context of a key-value store.

Theorem 1: Let K be the key-value store of Rollup that exports one read operation, namely get, and three write operations, namely put, update, and delete. If operation π completes before operation π' starts, then

- $ts_\pi \leq ts_{\pi'}$ in any case and
- $ts_\pi < ts_{\pi'}$ if π is a write operation.

Proof: The timestamp is used here to indicate the version of the entire key-value store but the proof follows from the register proof that assumes a timestamp per register [11, Theorem 5.7]: *(i)* If π and π' use the same configuration then the write quorum accessed by π intersects with the read quorum accessed by π'. *(ii)* It is not possible that π uses a configuration c ordered after configuration c' of π'. This would imply that there is a reconfiguration to a third configuration c'' after c that ends before π starts. This reconfiguration would notify a quorum of c about c'' and π' would discover it. *(iii)* If π uses a configuration ordered before π' configuration then consecutive reconfigurations will propagate the timestamp from the configuration of π to the configuration of π'. ■

Let us show that the key-value store ensures the three last properties of atomicity [38, Theorem 13.16]. As the first property of atomicity is induced by the others this is sufficient to prove that the key-value store is atomic.

Theorem 2: Let K be the key-value store of Rollup, let $get(k)$ be a read operation on key k and let $put(k,*)$, $update(k,*)$ and $delete(k)$, where $*$ indicates any possible value, be write operations on k. K is atomic if there exists a partial order so that in every execution α of K where every operation completes, the following properties hold:

1) all write operations are totally ordered, and every read operation is ordered with respect to all these writes;
2) there do not exist read or write operations π and π' such that π completes before π' starts, yet $\pi' \prec \pi$;
3) every read operation on k that is ordered after any write on k, returns the value of the last write on k preceding it in the partial order, and any read operation on k ordered before all writes returns the initial value associated with k.

Proof: Let $\langle \Pi, \prec \rangle$ be the partial order defined by the operation timestamps over all the operations Π so that $\pi \prec \pi'$ if $ts_\pi < ts_{\pi'}$ for all $\pi, \pi' \in \Pi$. Property (1) follows from the definition of timestamps. Property (3) follows from the sequential specification of a key-value pair, where a read applying to key k returns the last value v associated with key k. To show that the remaining property (2) holds, let us restate Therorem 1 in terms of the partial order. By Theorem 1, we know that if operation π completes before operation π' then $ts_\pi \leq ts_{\pi'}$ and the inequality is strict if π is a write. Hence, we cannot have $ts_{\pi'} < ts_\pi$. By definition of the partial order, we cannot have $\pi' \prec \pi$, which leads to the result. ■

Note that we did not require that messages get delivered as even an incomplete write (put, update, and delete) propagating to only a partial quorum does not violate atomicity: its linearization point will be after the operations that did not observe its timestamp and will be before any operation that encountered and propagated its timestamp.

C. Time complexity

Here we give an upper-bound on the time it takes for Rollup to complete depending on its granularity g and the time it takes to upgrade each individual machine u. To this end, we make additional assumptions that are not needed for correctness. In particular, we assume that there exists a point in time ℓ after which the system stabilizes and messages are exchanged among all participants in a bounded delay d. More precisely, the following assumptions hold after time ℓ:

1) Messages are received in at most d time, where d is a constant;
2) A machine gets upgraded locally from the current version to the new version in at most u time;
3) Nodes respond to protocol messages as soon as they receive them and they broadcast messages every d time to all participants;
4) Besides upgrades, all other local actions are processed with zero time passing;
5) Every participant of a new configuration is a participant since at least e time before the configuration was proposed.

While assumptions (1)–(2) ensure that Rollup terminate, assumptions (3)–(5) allow us to simplify the proof by relying on the analysis of RDS to analyze the time complexity of Rollup. Finally, starting from time ℓ there is only one Rollup executed at a time. Note that this latter assumption is reasonable given that after ℓ, messages get delivered within d time and Rollup is intended to be invoked by the cloud operator rather than by individual clients. The only missing property needed by the RDS reconfiguration analysis is that there exist in any active configuration, at least one read and one write quorum with all their members as non failed. The following lemma implies this property.

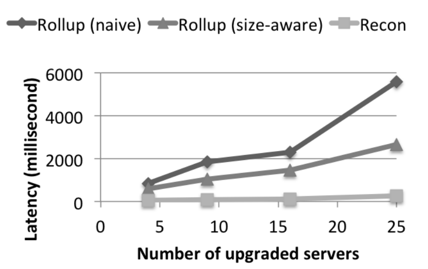
Lemma 3: In any active configuration, all the members of at least one read quorum and one write quorum have not failed.

Proof: We proceed by contradiction. We consider the case $n \geq 4$ as the case $n \leq 3$ is straightforward. Assume that there exists one active configuration c where no read and write quorums have all their members non failed. By the definition of our biquorum system (Section IV-A), there are $q = \sqrt{m}$ read (resp. write) quorums that are pair-wise disjoint in c, hence there must be at least q failures for none of the read (resp. write) quorums to have all their members non failed. As $m = \lfloor \sqrt{n} \rfloor^2$, we need to have at least $q = \lfloor \sqrt{n} \rfloor$ failures. This contradicts the assumption that there are at most $\lfloor \sqrt{n} \rfloor - 1$ failures in any active configuration when $n \geq 4$, leading to the result. ■

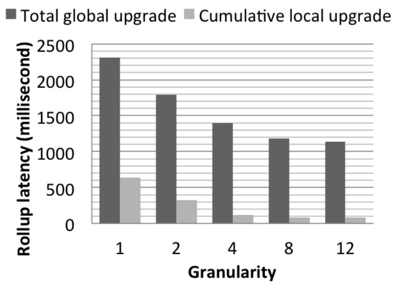
We can now analyze the time it takes for Rollup to complete by reusing the time complexity of the RDS reconfiguration.

Theorem 4: A Rollup request with granularity g invoked at time r is acknowledged at the latest at time $\max(r, \ell + e + 2d) + \lceil \frac{n}{g} \rceil (5d + u) + 7d$.

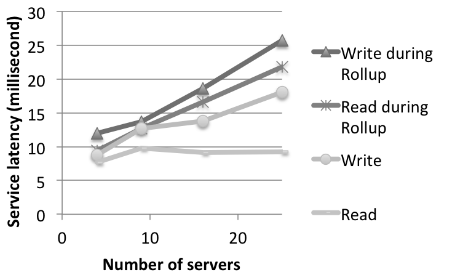
Proof: By examination of the code in Algorithm 1, we know that Rollup consists of a loop with $\lceil \frac{n}{g} \rceil$ iterations, each containing one reconfiguration (line 15) and g parallel local upgrades (line 16). By assumption (2), we know that all these g upgrades complete within u time. We know by Theorem 6.3 of [11] that if a node issues the reconfiguration from c to c' at time r, then the reconfiguration completes by



(a) The importance of the granularity g on the scalability of the rolling upgrade (the naive Rollup has $g = 1$ while the size-aware Rollup has $g = \sqrt{n}$)



(b) Rolling upgrade time as well as its cumulative local upgrade time on a 16-server system as the granularity of Rollup increases: cumulative local upgrade time decreases as the granularity grows, allowing more servers to locally upgrade in parallel



(c) Comparison of the service latency with and without concurrently executing Rollup reconfigurations – while the traffic of Rollup reconfigurations impact request response, read requests alone scale well with the system size

Fig. 4. Evaluation of Rollup on our cluster of 35 physical machines

time $\max(r, \ell+e+2d)+5d$. Note that this result holds even if the node that invokes the reconfiguration is different from the one that issued the previous successful reconfiguration to c, as it is the case in Rollup. Hence, by assumption (4) we know that if a node invokes a Rollup request at time r then the first iteration of the loop is complete by time $\max(r, \ell+e+2d)+5d+u$. As $\max(r, \ell+e+2d)+5d+u > r$, we know that all subsequent reconfigurations will only take $5d$ to execute. Since there are $\lceil \frac{n}{g} \rceil - 1$ remaining loop iterations, it will take $(\lceil \frac{n}{g} \rceil - 1)(5d+u)$ time to execute them. Summing up the time of the subsequent iterations with the time taken by the first iteration leads to $\max(r, \ell+e+2d) + \lceil \frac{n}{g} \rceil (5d+u)$. Accounting for the final reconfiguration (line 20) and the two messages necessary to invoke and acknowledge the Rollup request, the Rollup request invoked at time r is acknowledged at the latest at time $\max(r, \ell+e+2d) + \lceil \frac{n}{g} \rceil (5d+u) + 7d$. ■

The service operations terminate provided that new configurations get decided not too frequently, meaning that the CRUD operations do not get redirected to an infinite sequence of new configurations as specified by the recon-spacing assumption of RDS [11]. Section V shows that this assumption was always met experimentally in Rollup: recon (45–249 ms) is slower than key-value store operations (8–24 ms) so that operations restart at most once. Building upon the analysis of RDS, a Rollup operation takes between 2δ, with δ a lower bound on the message delay, in case of a get of a confirmed value (which is known optimal for any atomic read [16]) and $8d+\varepsilon$, where ε is a constant independent from the message delay, in case of a put, update or delete operation concurrent with a reconfiguration.

V. Experimental Analysis of Rollup

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. We ran a series of experiments using from 1 to 50 client machines issuing requests treated by from 4 to 49 servers to evaluate the service and its rolling upgrade. Each point on the graph in terms of reads (get) and writes (put/delete/update) performance is averaged over 50 values while each point representing Rollup (rollup/reconfiguration) performance is averaged over 20 values.

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 45 ms for 4 servers to 249 ms 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 $\lceil\frac{n-g}{g}\rceil$ 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 18 KiB 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 16 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. First, 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. Second, it is clear that the gain of parallelizing 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. Note that the impact of the granularity on fault tolerance has recently been studied elsewhere [55].

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.

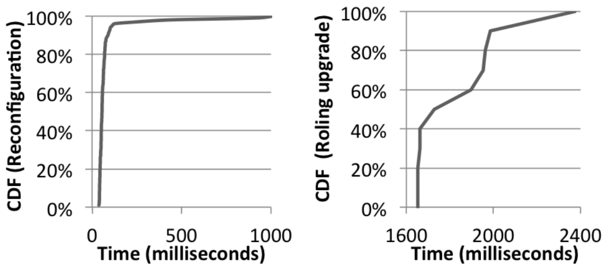


Fig. 5. Cumulative Distribution Function of the reconfiguration and rolling upgrade latencies on Amazon EC2

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 8 ms without Rollup and 11 ms with Rollup (37% increase) whereas at $n = 25$ the average latency is 13 ms without Rollup and 24 ms during Rollup (85% 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 [47].

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 biquorum 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 largest 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. The mean latency over all these runs was 82.41 ms, while the minimum

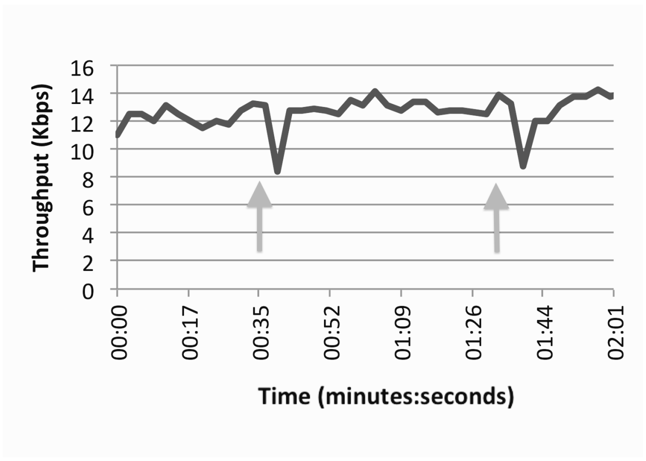


Fig. 6. Rollup slows down but does not disrupt the service on Amazon EC2

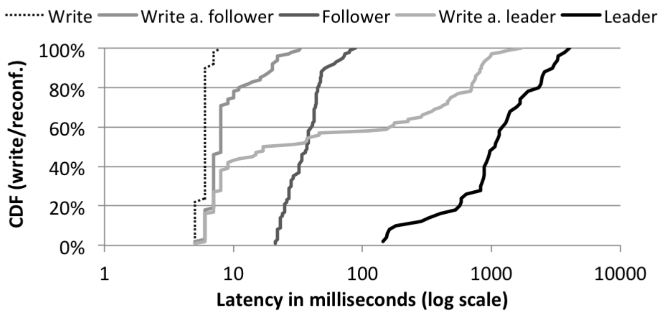


Fig. 7. The CDF of the Zookeeper latency in logscale while executing a baseline write (write), the write that follows a follower reconfiguration (write after follower), the total time of a follower reconfiguration followed by a write (follower), the write that follows a leader reconfiguration (writer after leader) and the total time of a leader reconfiguration followed by a write (leader)

latency was 38 ms and the median was 56 ms. We also ran 10 rolling upgrades and observed an average latency of 2003.8 ms between a minimum of 1652 ms and a maximum of 3161 ms, confirming that Rollup scales well to 49 servers.

While consensus-based storages often fail to scale to large numbers of participants due to many messages overloading a primary [29], we achieve scalability by avoiding consensus for the most frequent operations (reads/writes) and executing it exclusively during 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 [25]. Another scalable protocol, called Giraffe, is especially designed to cope with the lack of scalability of Zookeeper by distributing its tree-based structure into multiple trees [52].

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 [54]). Our key-value store is different from Zookeeper's as it only offers strongly consistent reads/writes and our goal was not to compete against Zookeeper's key-value store but simply to show non-disruption. We believe Zookeeper reads and writes could be more efficient as our key-value store is a prototype not yet at production stage. 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 [54, Fig.6].

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 [54] as the baseline as this protocol is at the heart of recent proposals like Raft [46] and CoreOS etcd's membership managements [50] used by academics and industrials. All values presented below are averaged over at least 50 runs on our Amazon EC2 testbed.

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 [54]. In order to totally order the subsequently installed configurations, Zookeeper uses the Zab atomic broadcast protocol [31] whereas Rollup uses the Paxos consensus protocol [32]. We evaluated the performance of the Zookeeper reconfiguration [54] that is part of the public release Zookeeper-107[3] and found that its latency can be of different orders of magnitude depending on the server being reconfigured. 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.

B. Invoking writes to evaluate Zookeeper's reconfiguration

Zookeeper's reconfiguration may return before its completion, which confirms that the completion is not known

[3]https://issues.apache.org/jira/browse/ZOOKEEPER-107

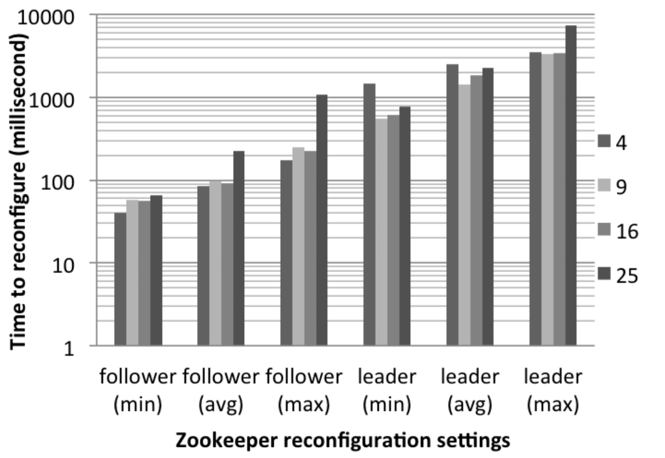


Fig. 8. Zookeeper reconfiguration performance varies greatly (note the log scale on the y axis) depending on the role of the affected servers (results are given for reconfiguring by removing a follower or removing a primary, a.k.a. leader, on 4, 9, 16, and 25 servers)

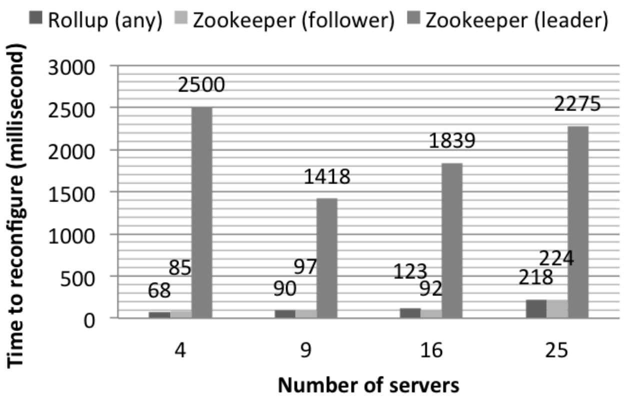


Fig. 9. The Zookeeper reconfiguration removing a primary (or leader) is one order of magnitude slower than other reconfigurations including the one based on Rollup (local upgrade time is not taken into account)

by the participants [54]. To measure the latency of reconfiguration we thus measured the time between the invocation of a reconfiguration request and the response time of an immediate subsequent write (create) operation. Because the writes are strongly consistent and have to contact the primary, we can guarantee that their response indicates that the primary was responsive. This is particularly useful to capture the time during which a leader reconfiguration prevents consistent requests from completing.

Figures 7 depicts the CDF of the latencies of (1) a write not preceded by any reconfiguration invocation, (2) a write preceded by a follower reconfiguration invocation, (3) a write preceded by a primary reconfiguration invocation, (4) a follower reconfiguration followed by a write and a leader reconfiguration followed by a write on Zookeeper with 5 servers. We observed similar results for 9, 15 and 25 servers. Note that the $\langle \text{median}, \text{average} \rangle$ pairs we observed for writes, writes after follower reconfiguration invocation, follower reconfigurations, writes after leader reconfiguration invocation and leader reconfigurations are: $\langle 6,6 \rangle$, $\langle 37,38 \rangle$, $\langle 8,10 \rangle$, $\langle 29,276 \rangle$ and $\langle 1070,1356 \rangle$, respectively. We observe that the writes take much longer to execute after a reconfiguration (notice the log scale on the x-axis). In particular, a write executes 178× slower after a leader reconfiguration than if no reconfiguration were preceding it. This difference with Rollup read/write latencies of Figure 4(c) is explained by Zookeeper's disruption.

C. *Latency of Zookeeper's reconfiguration*

We now measure the reconfiguration latency. Note first that the Zookeeper write latency, as measured above, is usually very low (6 ms) but a reconfiguration may delay it significantly. As mentioned before, we measure the reconfiguration latency by measuring the time it takes between a reconfiguration invocation and the response of an immediate subsequent write operation.

Figure 8 uses log-scale on the y-axis to depict the 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 request immediately following the reconfiguration response as explained previously.

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. 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 [54].

D. *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 9).

VII. Comparing Large-Scale Rolling Upgades

This section complements the experimental evaluation of the duration of primary-based reconfiguration by computing the time of a primary-based rolling upgrade. To analyze the

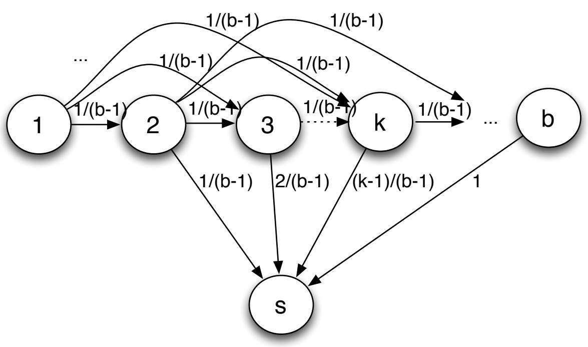


Fig. 10. The Markov chain used to compute the expected duration of the primary-based rolling upgrade: when the primary gets reconfigured, the probability for the new primary to be elected in any next batch is $\frac{1}{b-1}$ while the probability to "move" from some batch k to any previous batch (reaching the absorbing state s) is $\frac{k-1}{b-1}$

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 $b = \frac{n}{g}$ batches of servers at a time until all n servers are upgraded. (For the sake of simplicity in the analysis, let n 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 $D_{Primary}$ 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).

$$\begin{aligned} D_{Primary}(b) &= d_P X + d_B(b-X). \\ E[D_{Primary}(b)] &= E[d_P X + d_B(b-X)] \\ &= d_P E[X] + d_B(b - E[X]). \qquad (1) \end{aligned}$$

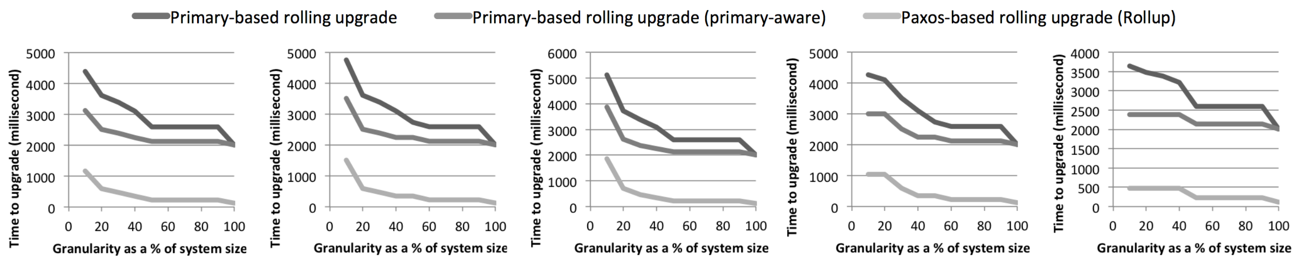
The process can be described as a discrete time Markov chain whose states are the potential batches with a primary, 1 to b, plus an absorbing state s as depicted in Figure 10. A transition is labeled with the probability of one node of the destination batch to be elected as the new primary. Initially, the primary is thrown uniformly at random in one batch, so the initial vector of $b+1$ coordinates is $V_0 = (1/b, ..., 1/b, 0)$. Then the primary moves from the batch k, where it belongs, to one of the next (to-be-upgraded) batches with probability $\frac{1}{b-1}$ or to anyone of the previous (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, we consider the case where $n = 25$, $g = \sqrt{n} = 5$ and $b = \frac{n}{g} = 5$. As we reported in the previous section (cf. Figure 9), 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, $d_P = 2008$ ms and $d_B = 124.5$ ms on average. By contrast, Rollup achieves any reconfiguration in time $d = 117$ ms on average. Finally, we can deduce from Equation 1 that the duration in this case is $E[D_{Primary}(5)] = 2008 \times 1.641406 + 124.5 \times 3.358594 = 3714$ whereas the duration to execute Rollup is $E[D_{Rollup}(5)] = 117 \times 5 = 585$, which is more than $6\times$ faster than a primary-based rolling upgrade. Note that even in the case where the leader gets explicitly changed before any upgrade, would the duration be at least $7.5 \times b$ milliseconds longer than with Rollup.

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 9, 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 11 simulates the expected durations of a rolling upgrade based on primary/backup reconfiguration and a rolling upgrade based on our reconfiguration (Paxos-based). We tested two primary/backup reconfigurations one where the primary is not known (Primary-based) and may be reconfigured multiple times (as we modelled with the Markov chain in Section VII-A) and another where the primary is known (Primary-aware) so that we can ensure it gets reconfigured exactly once. 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. 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. Note that even the Primary-aware approach remains still substancially slower than Rollup's: in the same setting ($n = 4$ and $g = 1$), our Paxos-based approach is still $5\times$ faster than the Primary-aware approach due to the overhead induced by reconfiguring one primary. On average, our Paxos-based approach is $8\times$ and $10\times$ faster than the Primary-aware and Primary-based approaches, respectively.



(a) n=100 (b) n=25 (c) n=16 (d) n=9 (e) n=4

Fig. 11. Comparison (with $\frac{n}{10} \leq g \leq n$) of the expected duration of a Primary-based rolling upgrade against our Paxos-based rolling upgrade and a variant of the primary-based rolling upgrade, called Primary-aware, where the primary is known and is upgraded exactly once (the local upgrade time is not counted in these results)

VIII. Related work

Rolling upgrade is not a novel research challenge as it had impact in distributed computing fifteen years ago [8]. There are few automated solutions for rolling upgrade as most production tools [48], [56], [51], [41] still require manual updates to check for bugs in order to prevent various software failures [14] mostly induced by mixed version [45].

Data Guard [47] 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 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 [13] 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 [15]. 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 [2] 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 [18], however, Paxos is generally used to ensure validity and agreement anytime while guaranteeing termination if specific conditions meet [32]. The consensus-less solutions also assume a majority of correct participants to guarantee liveness but usually require service requests to traverse the directed acyclic graph of partially ordered configurations before completing [3].

Thanks to its ability to totally order configurations [53], consensus-based reconfiguration has attracted lots of attention [40], [43], [42], [24], [10], [11], [54], however, it usually requires several message delays to achieve basic operations or garbage collect old configurations [22]. Other solutions target reconfiguration in the more general context of Byzantine failures [42]. The Reconfigurable Distributed Storage (RDS) was formally proved correct [11] using the IOA formalism [38], [19] but was never experimented on load-optimal quorum systems. Variants of this algorithm, including DSR [6], uses a majority for simplicity reasons but maintains the virtual synchrony property. The Rollup reconfiguration is the first generalization to a key-value store we know of.

An appealing feature for designing cluster membership protocols in a centralized way is avoiding the *Paxos anomaly* [7], [29]: Paxos decides on individual proposed values, potentially violating dependencies between values proposed by the same requester. In Zookeeper [29], 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: if two updates occur on the same key-value pair concurrently, then they will both succeed and none of them will be discarded.

By contrast, replicated state machines [34] aim at totally ordering all commands, be they read/write storage operations or more complex operations like reconfiguration, CAS, etc. This is the case of SMART [37] that builds upon Paxos, Zookeeper that uses Zab, and CoreOS etcd that uses Raft, to totally order write requests with respect to reconfiguration requests. Rollup illustrates that partial order is key to non-disruption: there are actually no needs for deciding upon the order between reconfigurations and updates. One could extend Rollup's storage service by offering distributed locking, multi-object transactions and CAS through its built-in consensus. In this case, it is necessary to distinguish these services so that upgrades would not disrupt the storage service.

Raft [46] describes a consensus 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 [50]. Note that write requests are forcefully forwarded to the leader both in CoreOS etcd[4] and in Zookeeper's reconfiguration [54]. Virtual Paxos (VP) [33] is a class of protocols that use a master for reconfiguration that could speedup Rollup. VP could benefit from local reads and high fault-tolerance if it has a primary as a read quorum but would then suffer from disruptions during primary failovers. Our service requests do not need any leader or master, hence the service is never disrupted.

Reconfiguration in partitioned key-value stores is also known to be a difficult problem. MongoDB uses auto-scaling to allow a user to adjust the number of partitions of the dataset, however, it requires the administrator to choose a shard key that cannot be adapted at runtime.[5] Morphus is as far as we know the first attempt to remedy this problem and has just been published at the time of writing [21]. Morphus employs an isolation phase where the operation log stops being executed by secondary servers during the data transfer required by a shard reconfiguration. Other key-value stores organize shards in a ring-based topology for the sake of scalability [23], [4], [1]. Scatter [23] reconfigures a shard using a two-phase commit transaction across shards. CATS [4] uses Paxos but requires subsequent messages during an extra view installation phase before data can be transferred. Elastic replication [1] reconfigures locally by creating new chains, however, all operations have to be ordered and a reconfiguration may disrupt them.

IX. Conclusion

The main appeal of rolling upgrade is to provide availability. 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. To achieve low load, each successive configuration of Rollup tolerates only $\sqrt{n}-1$ failures when $n \geq 4$. Rollup proved efficient as its reconfiguration is non-disruptive and its service scalable. Moreover, its reconfiguration protocol is one order of magnitude faster than Zookeeper's reconfiguration of the primary. Finally, we showed that a quorum-based rolling upgrade can speedup a primary-based rolling upgrade by $8\times$ even when the leader is carefully upgraded only once. Although, Rollup upgrades a stateful service like a storage without disruption, it does not solve the problem of implementing a non-disruptive upgrade for a distributed locking service. To our knowledge, this problem remains open.

Acknowledgements

We wish to thank Michael Troup for pointing out the delay between Zookeeper reconfiguration and write acknowledgements 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.

[4]https://coreos.com/docs/cluster-management/scaling/etcd-optimal-cluster-size/.

[5]http://docs.mongodb.org/manual/faq/sharding/#can-i-change-the-shard-key-after-sharding-a-collection.

References

[1] Hussam Abu-Libdeh, Robbert van Renesse, and Ymir Vigfusson. Leveraging sharding in the design of scalable replication protocols. In *SOCC*, pages 12:1–12:16, 2013.

[2] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–108, 2010.

[3] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):1–32, 2011.

[4] Cosmin Arad, Tallat M. Shafaat, and Seif Haridi. CATS: A linearizable and self-organizing key-value store. In *SOCC*, pages 1–2, 2013.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[6] Ken Birman. *Guide to Reliable Distributed Systems - Building High-Assurance Applications and Cloud-Hosted Services*. Texts in computer science. Springer, 2012.

[7] Ken Birman, Dahlia Malkhi, and Robbert van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.

[8] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, page 7, 2000.

[9] Shun Yan Cheung, Mostafa H. Ammar, and Mustaque Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):582–592, December 1992.

[10] Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alexander A. Shvartsman. Reconfigurable distributed storage for dynamic networks. In *OPODIS*, volume 3974 of *LNCS*, pages 351–365, Apr. 2005.

[11] Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alexander A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *J. of Parallel and Distributed Computing*, 69(1):100–116, jan 2009.

[12] Nicolas de Condorcet. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Imprimerie Royale, 1785.

[13] Tudor Dumitraş and Priya Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Middleware*, pages 18:1–18:20, 2009.

[14] Tudor Dumitraş, Priya Narasimhan, and Eli Tilevich. To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. In *OOPSLA*, pages 865–876, 2010.

[15] Tudor Dumitraş, Jiaqi Tan, Zhengheng Gho, and Priya Narasimhan. No more hotdependencies: Toward dependency-agnostic online upgrades in distributed systems. In *HotDep*, 2007.

[16] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *PODC*, pages 236–245, 2004.

[17] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. Development and deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[18] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[19] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A language for specifying, programming, and validating distributed systems*, 2001.

[20] Chryssis Georgiou, Nicolas Nicolaou, Alexander C. Russell, and Alexander A. Shvartsman. Towards feasible implementations of low-latency multi-writer atomic registers. In *NCA*, pages 75–82. IEEE, Aug. 2011.

[21] Mainak Ghosh, Wenting Wang, Gopalakrishna Holla, and Indranil Gupta. Morphus: Supporting online reconfigurations in sharded NoSQL systems. In *ICAC*, 2015.

[22] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *DSN*, pages 259–268, 2003.

[23] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *SOSP*, pages 15–28, 2011.

[24] Vincent Gramoli. RAMBO III: speeding up the reconfiguration of an atomic memory service, 2004. MS thesis. Université Paris-Sud.

[25] Vincent Gramoli, Emmanuelle Anceaume, and Antonino Virgillito. SQUARE: Scalable quorum-based atomic memory with local reconfiguration. In *SAC*, pages 574–579. ACM, March 2007.

[26] Vassos Hadzilacos and Sam Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report TR94-1425, Department of Computer Science, Cornell University, May 1994.

[27] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst.*, 4(1):32–53, 1986.

[28] Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *ATC*, pages 11–11. USENIX, 2010.

[30] Flavio Junqueira and Benjamin Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, Nov. 2013.

[31] Flavio Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, pages 245–256, 2011.

[32] Leslie Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, 1998.

[33] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *PODC*, pages 312–313, 2009.

[34] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

[35] Leslie Lamport and Mike Massa. Cheap Paxos. In *DSN*, pages 307–, 2004.

[36] Hyunyoung Lee, Jennifer L. Welch, and Nitin H. Vaidya. Location tracking using quorums in mobile ad hoc networks. *Ad Hoc Networks*, 1(4):371–381, 2003.

[37] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *EuroSys*, pages 103–115, 2006.

[38] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[39] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

[40] Nancy A. Lynch and Alexander A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *DISC*, pages 173–190, 2002.

[41] IBM. Tivoli Endpoint Manager. http://www-01.ibm.com/software/tivoli/solutions/endpoint/.

[42] Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic byzantine storage. In *DSN*, pages 325–334, 2004.

[43] Peter M. Musial and Alexander A. Shvartsman. Implementing a reconfigurable atomic memory service for dynamic networks. In *Proceedings of 18th International Parallel and Distributed Symposium*, page 208b, 2004.

[44] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447, April 1998.

[45] Iulian Neamtiu and Tudor Dumitraş. Cloud software upgrades: Challenges and opportunities. In *Proceeding of the IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 1–10, 2011.

[46] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *ATC*, pages 305–319, Philadelphia, PA, 2014. USENIX.

[47] Oracle Corporation. Database rolling upgrade using data guard SQL apply. Maximum availability architecture, December 2008.

[48] Oracle Corporation. Realizing the economic benefits of oracle enterprise manager ops center, 2012. http://www.oracle.com/us/products/enterprise-manager/economic-benefits-ops-center-185035.pdf.

[49] David Peleg and Avishai Wool. The availability of quorum systems. *Inf. Comput.*, 123(2):210–223, 1995.

[50] Brandon Philips. Distributed configuration data with etcd, 2013. https://coreos.com/blog/distributed-configuration-with-etcd/.

[51] Microsoft Corporation. Windows Server Update Services. http://technet.microsoft.com/en-us/windowsserver/bb332157.aspx.

[52] Xuanhua Shi, Haohong Lin, Hai Jin, Bing Bing Zhou, Zuoning Yin, Sheng Di, and Song Wu. GIRAFFE: A scalable distributed coordination service for large-scale systems. In *CLUSTER*, pages 38–47. IEEE, 2014.

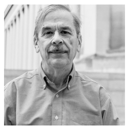
[53] Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. In *LADIS*. ACM, 2010.

[54] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *ATC*, pages 39–39. USENIX, 2012.

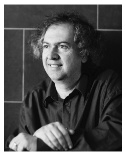
[55] Wei Sun, Daniel Guimarans, Alan Fekete, Vincent Gramoli, and Liming Zhu. Multi-objective optimisation for rolling upgrade allowing for failures in clouds. In *SRDS*, 2015.

[56] VMware Inc. vSphere Update Manager. http://www.vmware.com/products/datacenter-virtualization/vsphere/update-manager.html.

Vincent Gramoli is an academic at the University of Sydney and a senior researcher at NICTA, National ICT Australia. Vincent started his research on the topic of reconfigurable atomic memory while visiting the University of Connecticut and MIT (USA). He worked on probabilistic distributed algorithms at Université of Rennes where he received his PhD, INRIA (France) and Cornell University (USA), and contributed to the development of the transactional memory stack at EPFL and University of Neuchâtel (Switzerland). He is the main author of Synchrobench.



Len Bass received his PhD in Computer Science from Purdue University in 1970 under supervision of Paul Ruel Young with the thesis, entitled "Hierarchies based on computational complexity and irregularities of class determining measured sets". Bass was appointed Professor of Computer Science at the University of Rhode Island in 1970. In 1986 he moved to the Software Engineering Institute at the Carnegie Mellon University, where he started as head of the user-interface software group, and later focussed on analysis of software architectures. Since 2011 he is Senior Principal Researcher at NICTA, National ICT Australia.



Alan Fekete is a Professor of Enterprise Software Systems within the School of Information Technologies at the University of Sydney. His undergraduate education was at the University of Sydney, and his doctorate was earned in the mathematics department of Harvard University. He has been an academic at the University of Sydney since 1988, and he was promoted to Professor from 2010. Alan is a member of ACM and ACS, and of the IEEE Computer Society. He has been recognized as a Distinguished Scientist by ACM for "significant accomplishments in, and impact on, the computing field".

Daniel W. Sun is a researcher in Software Systems Research Group, National ICT Australia, and a conjoint lecturer in The University of New South Wales. After he was awarded his Ph.D. in Japan Advanced Institute of Science and Technology, he worked for NEC Central Labs as a researcher and then an assistant manager from 2008 to 2012. His research interests include largely parallel and distributed systems, real-time system, Big Data, and cyber security.