Leveraging Sharding in the Design of Scalable Replication Protocols

Hussam Abu-Libdeh
Dept. of Computer Science
Cornell University
hussam@cs.cornell.edu

Robbert van Renesse
Dept. of Computer Science
Cornell University
rvr@cs.cornell.edu

Ymir Vigfusson
School of Computer Science & CRESS
Reykjavik University
ymir@ru.is

Abstract

Most if not all datacenter services use sharding and replication for scalability and reliability. Shards are more-or-less independent of one another and individually replicated. In this paper, we challenge this design philosophy and present a replication protocol where the shards interact with one another: A protocol running within shards ensures linearizable consistency, while the shards interact in order to improve availability. We provide a specification for the protocol, prove its safety, analyze its liveness and availability properties, and evaluate a working implementation.

1 Introduction

Datacenter services are usually scaled out by horizontally partitioning their data into multiple more-or-less independent shards that are then replicated for enhanced availability and failure tolerance. With replication comes the question of consistency. A strongly consistent replicated system behaves, logically, identical to its unreplicated counterpart. However, many large-scale fault-tolerant services used in datacenters today provide weaker consistency guarantees such as eventual [42, 4] or causal [27] consistency, which provide easy scale-out and predictable performance in the face of crash failures or overload. While relaxed consistency is useful in many contexts, programming against weakly consistent services is difficult.

Strongly consistent services have recently made a comeback with Spanner, Megastore and Scatter [11, 5, 16] having previously been considered hard to implement, unscalable or expensive to operate. These services are carefully designed to provide availability and performance, even though the replicas must coordinate all operations among themselves.

However, a prominent challenge brought on by scale is configuration management. Node reconfiguration in strongly consistent services usually relies on an *external* replicated and failure-tolerant centralized configuration management service (CCM), such as Chubby and ZooKeeper [10, 20]. Internally, most CCMs use a state machine replication protocol, such as Paxos or Zab [24, 31].

In this paper, we ask if strongly consistent services can provide flexible and easy reconfiguration by leveraging the shards themselves. The idea is to make the system shards serve a dual purpose: to store application state, and to manage the configuration of other shards. Furthermore, the mechanism can make weaker availability assumptions about the replicated shards than modern CCMs can make for their internal consensus protocols.

Elastic replication. We develop these ideas into a new scheme for replicating sharded services that we call *elastic replication*. The main contribution of elastic replication is that reconfiguration is both simple and adaptable. A replicated object's configuration is separated from its state, and the object can only be reconfigured by dictation from an external entity. This independence of state and configuration relieves the need for solving consensus to reconfigure the service.

Our protocol makes the following minimal assumptions about the availability of shards at any point in time.

(A1) Each shard has at least one non-faulty replica.
(A2) There is at least one shard with no faulty replicas.

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1.
http://dx.doi.org/10.1145/2523616.2523623

A novelty of elastic replication is that shards are used to reconfigure each other in response to failure without the need to use a configuration master, as in Vertical Paxos [25] or other existing systems. Shards are organized into monitoring/reconfiguration rings called *elastic bands*. Each shard belongs to one ring and monitors the successive shard on the ring. Because of this design, and the use of shards to issue new configurations, reconfiguration does not require running an instance of a consensus algorithm as long as there exists at least one non-faulty shard in the band (A2). External intervention is required only if all shards are faulty.

Elastic replication offers the following benefits, which we will demonstrate throughout the paper:

- *Customizable consistency.* Strong consistency guarantees such as linearizability [19] are provided to applications that need them, while other applications receive improved timeliness, particularly in the face of network partitioning, in exchange for weaker guarantees [12, 30, 43, 42].
- *Minimal cost of replication.* In order to survive f failures, no more than $f+1$ replicas are required, and message and computational overheads are low.
- *Robust consistency.* Consistency does not depend on accurate detection of failures — pinging methods may lead to false positives in systems that lack real-time guarantees [37].
- *Smooth reconfiguration.* Reconfiguration does not violate consistency guarantees [34], and is both fast and straightforward.

Our paper makes the following contributions:

- We specify a new distributed replication protocol with strong consistency guarantees;
- We formally prove the safety of our protocol and analyze its scalability;
- Through experimental analysis, we demonstrate that the protocol we show that the protocol can be used to build scalable and highly reliable services that support easy reconfiguration.

2 Background and Related Work

The replication and consistency semantics of distributed systems is an active field of research. Much of the related work will be discussed throughout the paper, however we lay some preliminaries here.

Replication. Modern datacenter services rely primarily on two approaches to replication: Primary-backup and quorum intersection protocols. Primary-backup [1, 9] is used in systems such as GFS and HDFS [15, 38]. A primary replica receives all updates, orders them, and forwards them in FIFO order to the non-faulty backup replicas. In case of an unresponsive primary, another replica may become primary when reconfigured by a configuration management service. If the original primary was mistakenly suspected of having failed, a client may read the result of some update operation to the original primary that is not applied, and never will be, to the new primary, resulting in *divergence*. In order to avoid the likelihood of divergence, clients track the current active configuration as maintained by the configuration manager.

Quorum Intersection protocols are particularly useful for put/get-type Key-Value Stores such as Amazon's Dynamo and Apache Cassandra [13, 22]. A put operation, accompanied by a timestamp, is sent to a "put-quorum," while a get operation reads from a "get-quorum" (and returns the result with the highest timestamp). By making quorums smaller than the entire set of replicas, availability and performance are achieved. By guaranteeing that any put-quorum and get-quorum intersect, a get can be guaranteed to see the latest completed put operation. However, divergence can still occur in the case of dynamic reconfiguration of replicas as quorums may temporarily fail to intersect [13].

Strong consistency. Relaxed consistency guarantees have enabled the large-scale distributed systems of today's cloud computing environments. However, the lack of consistency is not suitable for all applications. Even for current cloud applications, developers usually pretend they are dealing with consistent data regardless of the consistency guarantees provided, simply because it is often hard to program against relaxed consistency.

Current strong consistency protocols rely on majority voting techniques where $2f+1$ replicas are required to tolerate f failures. Using $2f+1$ replicas, as is the case in Paxos and Quorum Replication systems, is not only expensive in terms of resources, but it is also harder to ensure that the larger number of replicas fail independently. Also, reconfiguration—important to service expansion, migration, or software updates in the cloud—is difficult while maintaining strong consistency.

Failure detection. Primary-backup assumes a fail-stop model with perfect failure detection. In practice, timeouts are often used to detect failures in these protocols and the choice of timeout values results in a trade-off between the liveness and safety of the system. Elastic replication only assumes crash-failures with imperfect failure detection, and the choice of timeout values does not impact safety. Even with a centralized configuration manager like Zookeeper or Chubby there could be a delay until all clients learn of a new configuration as demonstrated in Figure 1. In this experiment, we used Zookeeper to manage the configuration of a replicated service, forced a configuration change, and measured the

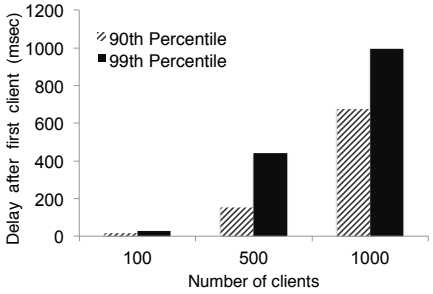

Figure 1: Delay between first client being notified of a configuration change and the 90th and 99th percentile being notified. This delay can result in divergence or inconsistency.

latency until 90% and 99% of the clients get notified of the new configuration after the first client is notified. Late notifications about changes could result in inconsistencies if clients interact with a defunct primary or an old replica. As we'll later show, elastic replication uses a technique called *wedging* to ensure this never happens.

Reconfiguration. Various scalable peer-to-peer storage systems have proposed separating configuration from storage for improved consistency [33, 32, 26, 28]. Similarly, Vertical Paxos (VP) [25] uses an auxiliary configuration master to reconfigure primary-backup or read/write quorums. Another such approach is Dynamic Service Replication [7], a mixture of consensus and virtual synchrony-based [8] replication strategies. There are two important differences between these works and elastic replication. First, they rely on a majority-based consensus protocol for reconfiguration, whereas elastic replication does not. Second, they use consensus not only to agree on the sequence of configurations, but also to agree on the initial application state in each configuration. In elastic replication, replicas can be reconfigured without updating their application state.

In cloud services, centralized configuration services such as Chubby [10] and ZooKeeper [20] are increasingly common. Note that these configuration services use majority-based consensus protocols to serialize configurations, while the systems they manage rarely provide strong consistency guarantees. Also, many of these services provide only atomic read/write operations, while elastic replication supports the full deterministic state machine model. Elastic replication is best considered a scalable instantiation of the Replicated State Machine (RSM) approach [23, 36] in which a collection of replicas apply the same operations in the same order.

Our work is similar to Vertical Paxos in that the replicated state is separate from the configuration. However, unlike Vertical Paxos, we do not rely on an external configuration master to reconfigure. Additionally, we do not require running an instance of the consensus protocol to reconfigure. Furthermore, our approach is novel in its use of shards to monitor and reconfigure each other.

Scatter [16] is another system that is related to our design. Scatter uses a ring of shards and provides linearizable consistency. Like elastic replication, Scatter also spreads the task of managing shard configurations on other shards. However, there are important differences between Scatter and elastic replication. Scatter uses Paxos in each shard, and reconfiguring a shard is done via a two-phase commit transaction that runs across the shards. This is not the case in our protocol. Additionally, compared to elastic replication, Scatter requires almost double the number of replicas ($2f+1$ instead of $f+1$).

3 Elastic Replication

In this section, we specify, refine, and compare elastic replication to other replication protocols. A formal proof of the protocol's safety is provided in the attached appendix.

Environment Model. An asynchronous environment is assumed with no bounds on processing times or message delays. Failures are assumed to be either crash or omission failures (although the method can be generalized to Byzantine failures as well [40]).

We develop elastic replication in steps. First, we show how a single shard can be replicated assuming a sequence of configurations that is defined *a priori*. Next, we show how collections of shards can manage the configurations of one another. Our discussion starts with a high-level specification which is then refined. Section 3.5 describes an actual implementation of elastic replication.

3.1 Replicating Single Shards

We model the state of a shard as a finite history of operations: $\mathscr{H} = o_1 :: o_2 :: o_3 :: ...$ where $\mathscr{H}$ is initially empty.

The only operation possible on this high-level shard is to add a sequence of operations to its history. A shard can then be specified as follows (the operator $\cdot$ concatenates two sequences):

specification `Shard`:
  transition `apply`(S):
    action: $\mathscr{H} := \mathscr{H} \cdot S$

To make the shard highly available, we use replication and dynamically change the configuration of replicas in order to deal with crash failures and unresponsiveness. For now, assume the existence of an unbounded sequence of configurations $\mathscr{C} = C_1 :: C_2 :: C_3 :: ....$ The

boolean function $succ(C, C')$ evaluates to `true` if and only if C' directly follows C in $\mathscr{C}$. Each configuration C_i consists of the following:

- $C_i.replicas$: a set of replicas; and
- $C_i.orderer$: a designated replica in $C_i.replicas$.

For simplicity, we will initially assume that the replicas of any two configurations are disjoint. Later, we will drop this impractical assumption. We call replicas of the same configuration *peers*. In order to tolerate up to f failures, each configuration needs at least $f+1$ independently failing replicas. A replica r has the following state:

- *r.conf*: the configuration this replica belongs to;
- *r.orderer*: the orderer of this configuration;
- *r.mode*: is either `PENDING`, `ACTIVE`, or `IMMUTABLE`. Initially all replicas in C_1 are `ACTIVE`, while replicas in other configurations are all `PENDING`;
- *r.history*: a sequence of operations $o_1 :: o_2 :: ....$ In practice, replicas maintain a running state, but this model is easier to understand;
- *r.stable*: the length of a prefix of the history that r knows to be *persistent* (initially 0). The inequality $0 \leq r.stable \leq length(r.history)$ always holds.

Let $gcp(C)$ be the greatest common prefix of the histories of the replicas of configuration C. In the appendix we formally show that the greatest common prefix of the histories of a configuration's replicas is persistent.

Figure 2 shows five atomic transitions that are allowed:

1. `addOp`(r,o): if r is an active orderer of a configuration, it is allowed to add an operation o to its history;
2. `adoptHistory`(r): a non-immutable replica r may adopt the history of the orderer in its configuration as long as the orderer has a stable prefix that is at least as long as that of the replica;
3. `learnPersistence`(r,s): an active replica r may extend its stable prefix up to the greatest common prefix of all histories of its peers;
4. `wedgeState`(r): an active replica r may cease operation, giving way to the next configuration making progress;
5. `inheritHistory`(r,r'): a pending replica r may assume the history of an immutable replica r' in the prior configuration and become active.

The transitions specify what actions are safe (ensure persistence), but not when or in what order to do them.

specification `Replicas`:
  transition `addOp`(r,o):
    precondition:
      $r.mode = \texttt{ACTIVE} \wedge r = r.orderer$
    action:
      $r.history := r.history :: o$
  transition `adoptHistory`(r):
    precondition:
      $r.mode \neq \texttt{IMMUTABLE} \wedge$
      $r.orderer.mode \neq \texttt{PENDING} \wedge$
      $r.history \neq r.orderer.history \wedge$
      $r.stable \leq r.orderer.stable$
    action:
      $r.history := r.orderer.history$
      $r.stable := r.orderer.stable$
  transition `learnPersistence`(r,s):
    precondition:
      $r.mode = \texttt{ACTIVE} \wedge r.stable < s \leq gcp(r.conf)$
    action:
      $r.stable := s$
  transition `wedgeState`(r):
    precondition:
      $r.mode = \texttt{ACTIVE}$
    action:
      $r.mode := \texttt{IMMUTABLE}$
  transition `inheritHistory`(r,r'):
    precondition:
      $r.mode = \texttt{PENDING} \wedge succ(r'.conf, r.conf) \wedge$
      $r'.mode = \texttt{IMMUTABLE}$
    action:
      $r.mode := \texttt{ACTIVE}$
      $r.history := r'.history$
      $r.stable := r'.stable$

Figure 2: Specification of replicas.

We now sketch in a more operational manner when non-faulty replicas perform transitions in the specification above.

Clients send operations to an active orderer, which will add each received operation to its history (possibly filtering out duplicates). This corresponds to the `addOp` transition. Note that we are not ruling out that there exist multiple active orderers of different configurations. However, as we will prove in the appendix, only one configuration at a time will be composed of all active replicas.

The goal of an active orderer is to get its peers to accept its history. Upon becoming active, and upon adding operations to its history, the orderer will notify its peers. Corresponding to the `adoptHistory` transition, a non-immutable replica will adopt the history of the orderer if the orderer has a stable prefix that is at least as long as its own stable prefix —it is an invari-

ant that replicas never truncate their stable prefix. Note that our specification does not say how operations are distributed, providing flexibility to the implementation.

As soon as an operation is in the history of all peers of a configuration, it is persistent. Replicas can only act upon an operation once they learn it is persistent. The way this could be done is as follows: After receiving a request to adopt the history of the orderer, the replica returns the length of the common prefix of its history with that of the orderer. If an orderer receives a response from all its peers, it can calculate the minimum and increase its stable prefix accordingly (corresponding to the `learnPersistence` transition). The orderer would then notify its peers, who can then update their stable prefix as well (also `learnPersistence`).

If an active replica suspects that one of its peers is faulty, it goes into immutable mode (`wedgeState` transition). Once immutable, the state of a replica r can no longer change and only operations already in *r.history* can become persistent in its configuration. Replicas in the subsequent configuration can transition from pending to active mode using the history of an immutable replica (the `inheritHistory` transition), as described below, and continue to make progress.

3.2 Liveness

As we have indicated, if an active replica suspects that one of its peers is faulty, it becomes immutable. For this, we need to assume that correct replicas eventually suspect faulty peers, something that can be implemented with a simple pinging protocol. As safety does not assume accurate failure detection, the pinging protocol can use aggressive timeouts in order to detect failures quickly.

Theoretically, liveness of a single shard depends on there existing a configuration in $\mathscr{C}$ consisting only of correct replicas that never suspect one another. Such an assumption is necessary in an asynchronous environment; indeed, if we could show liveness without such an assumption, we would violate the FLP impossibility result [14]. For now, we will argue that a shard can make progress given this assumption.

Let C be the first configuration in $\mathscr{C}$ with correct replicas that never suspect one another of failure. If C is the first configuration in $\mathscr{C}$, then liveness follows easily: All replicas are active and none of them will become immutable by our assumptions. If C is not the first in $\mathscr{C}$, then let $\mathscr{C}'$ be the sequence of configurations before C. The first configuration in $\mathscr{C}'$ has at least one correct replica that will become immutable, and thus the pending replicas in the next configuration in $\mathscr{C}'$ can become active and inherit its history. This continues until finally there is a correct replica in the predecessor configuration to C that becomes immutable. Thus all replicas in C can become active, and then it is clear that it can make progress from then on, as none of these replicas will ever become immutable by assumption.

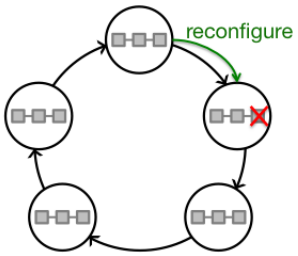

Figure 3: Example of an elastic band. Each (replicated) shard is sequenced by its predecessor on the band as indicated by the arrows. The sequencer of a shard issues new configurations when needed.

3.3 Elastic Bands

Thus far we have assumed that there is a single replicated shard, and that its unbounded sequence of configurations is determined *a priori*. We will now drop these assumptions and instead assume a dynamic collection of shards $\mathscr{X}$. Logically, each shard $x \in \mathscr{X}$ has a *sequencer* that determines its sequence of configurations. A sequencer can be further subdivided into two distinct functions: *sequencing mechanism* and *policy*.

The sequencing mechanism aspect ensures that for every shard there is a sequence of configurations that never branches. The policy function determines when the configuration needs to be changed and what the new configuration should be in accordance with the deployment policy. In this section, we will focus only on the sequencing mechanism as discussion of deployment policies is beyond the scope of this paper.

In elastic replication, the sequencing of configurations of a shard x is accomplished by *another shard* x'. Shards are organized into one or more circular *elastic bands*, with the configuration of any shard x on a band being sequenced by exactly one other shard x' on the same band; its predecessor shard in the band. As a shard can have only one sequencer, a shard cannot be on two different bands at once. Also, as shards cannot sequence themselves, each band has at least two shards on it. Figure 3 depicts an elastic band.

Each shard has dual responsibilities: to store and manipulate the application-specific state of its replicated state machine, and to issue new configurations for its successor shard in the band when needed —e.g. in response to failure.

We will now describe the shard sequencing interface. Let $x.id$ be a unique identifier for shard x. The sequencing interface then consists of two operations:

- $s.putConf(x.id, config)$: this makes s the sequencer of x, with *config* the current configuration of x;
- $s.getConf(x.id)$: if shard s has the configuration of x, it will stop being the sequencer for x and return the configuration of x. Otherwise, s will return an error.

A configuration itself consists of a tuple $\langle id, index, locations, orderer \rangle$. Here *id* is the shard identifier, *index* is the index in its sequence of configurations, *locations* is a set of host identifiers (e.g. TCP/IP addresses), and *orderer* is the host in *locations* that runs the orderer replica.

It is easy to insert into and remove shards from a band. Doing so would be under the control of a configuration management agent m, itself an object in a shard. Consider shards x and x' on a band, with x the sequencer of x'. To insert a shard y in between x and x', m first obtains the configuration of x' from x using *getConf*, and then puts the configuration into y using *putConf*. Then m puts the configuration of y into x using *putConf*. This concludes inserting a new shard into the band. Note that even though shard y is the new sequencer of x', shard x' does not need to be notified of the change since sequencing is a one-way relationship. Additionally, note that m *can* only do this if it has the configuration of y, and *should* only do this if y does not have a sequencer already.

To remove y, m first gets the configuration x' from y, and then puts this configuration into x, making x the sequencer of x'. Finally, m can get the configuration of y from x, and destroy y or put its configuration at a new sequencer.

For liveness, we assume that at any time (A1) for any band, each shard contains at least one correct replica, and that (A2) each band has at least one shard that has no faulty replicas. (A1) is a standard safety assumption made by all replication protocols, and it is satisfied by having $f+1$ replicas per shard if f replicas can fail simultaneously. (A2) is required because shards are reconfigured by other shards on the same band rather than an external centralized configuration management service (CCM). In systems where shards are reconfigured by a CCM, (A2) is replaced by an assumption:

(A3) The centralized configuration manager does not fail.

In section 4.1 we will provide an exact formulation of the probability that (A1) and (A2) are met, compare that to the probability that (A3) is met, and show that organizing shards into elastic bands can result in higher reliability compared to using a CCM.

specification `Replicas++`:
  transition `inheritHistory2`(r,C):
    precondition:
      $r.mode \neq$ `PENDING` $\wedge\, succ(r.conf, C) \wedge$
      $r \in C.replicas$
    action:
      $r.mode :=$ `ACTIVE`
      $r.conf := C$
  transition `inheritHistory3`(r,r'):
    precondition:
      $r.mode =$ `PENDING` $\wedge\, r'.conf = r.conf \wedge$
      $r'.mode \neq$ `PENDING`
    action:
      $r.mode :=$ `ACTIVE`
      $r.history := r'.history$
      $r.stable := r'.stable$

Figure 4: Additional transitions allowed for replicas.

3.4 Practical Considerations

To simplify presentation, we have assumed that successive configurations do not overlap, that is, they have no replicas in common. We now show how we can drop this impractical assumption by adding two more transitions to the specification, shown in Figure 4.

The transition `inheritHistory2`(r,C) allows a replica that is not pending to move from configuration $r.conf$ to the successor configuration C. It inherits its own history, and does not become immutable. This, however, presents a problem, as other replicas in C may now not have an immutable replica in the previous configuration to inherit state from. The extended specification solves this by allowing pending replicas to inherit state from non-pending peers in the same configuration (transition `inheritHistory3`(r,r')).

Another impractical simplification we made for ease of specification is that replicas maintain the entire history of operations. However, instead of the stable part of the history, replicas can maintain a *running* state. This is possible because the stable part of the history is persistent and never has to be undone. Replicas will need to queue the remaining operations until they are known to be stable or maintain a running state with a log of "undo" operations in order to be able to roll back. The latter would make it possible for applications to support weakly consistent semantics where updates are exposed quickly but are not known to be persistent.

Finally, we have not optimized read-only operations in any way—so far they have been treated as any other command. In practice, read-only operations do not have to become persistent as they do not affect the running state of an application. In case linearizable semantics are needed, a replica can handle a read-only command as

soon as the update operations that precede it are known to be persistent. If sequential consistency suffices, read-only operations can be handled upon arrival at a replica.

3.5 Implementation

As mentioned in Section 3.1, our specification does not include how operations are distributed to replicas. In our implementation we organized replicas within a shard in a chain akin to chain replication (CR). We chose this construction due to the simplicity of CR and its high throughput properties.

The orderer is the head of the chain, and it sends operations along the chain. Each replica on the chain adds the operations to its history. When the tail replica adds an operation to its history, all replicas have the operations in their histories and thus the operation is persistent. This allows the tail to respond to the client. The tail then sends an acknowledgment along the chain in the other direction so that other replicas can also learn that the operation is persistent and apply the operation to their state.

While similar, there are some important differences between elastic replication (ER) and the original CR:

- in ER, when a replica is unresponsive, a new chain is configured, possibly with a new collection of replicas. In CR, faulty replicas are removed from a chain and new replicas added to the end of the chain. The reason behind this difference is that CR assumes a fail-stop model [35] in which failures are detected accurately, whereas ER assumes that crashes cannot be detected accurately—a weaker assumption;
- in ER, read-only operations have to be ordered just like any other updates, whereas in CR clients can read directly from the tail. Again, this is because of the lack of accurate failure detection in our model: if the tail were incorrectly suspected of failure and the chain reconfigured, the tail's state would become stale without it knowing that its configuration is no longer current. In ER, old configuration are wedged and even read operations cannot be serviced by it. This apparent inefficiency of ER can be addressed by using *leases* [17] to obtain the same read efficiency as CR.

There are some subtle issues to consider when a chain is reconfigured, because each chain must satisfy the CR invariant that a server to the left in a chain has a longer history than a server to the right [41]. ER thus ensures that when a chain is reconfigured, the replicas that were in the old configuration appear at the beginning (the left) in the new chain *in the same order as in the old configuration*, while new replicas appear at the end of the chain with an initially empty history.

Our shard implementation is similar to CRC Shuttle implementation of Byzantine CR. However, Byzantine CR relies on an external configuration master, while ER delegates reconfiguration to other shards on the elastic band.

For increased efficiency, we can exploit parallel access and allow clients to read from any replica, at the cost of a weaker semantics. If clients read the stable running state of any replica, they know the state is persistent but possibly stale. If clients read the full history of any replica, then they are not guaranteed that the state is persistent or that the order of operations stays the same. It is up to the application programmer to know whether such weaker semantics are acceptable.

3.6 Reconfiguration Discussion

Thus far we have described the transitions involved in reconfiguring a shard and the sequencing relationship between shards in an elastic band. We now elaborate on reconfiguration in elastic replication.

Reconfiguration prompts several natural questions: How is a new shard configuration chosen? How is the new replica membership set? How is the next configuration orderer selected? These concerns are matters of *policy* rather than of mechanism. Elastic replication separates mechanism from policy [6] and is only focused on mechanism issues. Thus, any configuration with any set of replicas and any orderer could be chosen to reconfigure an existing shard without compromising correctness. In fact, the sequencing shard could be reading a predetermined list of configurations off a disk and it would not impact system safety.

In our implementation, when a replica is being unresponsive, we immediately restore service by wedging the replica's current configuration and issuing a new configuration. The new configuration is simply the old configuration with the unresponsive replica removed. The overhead is the same as that of any update operation on the sequencer shard. Next, in order to restore the original fault tolerance, a new replica is allocated, placed at the end of the chain, and set to inherit the state of a replica in the current configuration in the background. Finally, once the new replica has the same persistent history as a replica in the current configuration, the sequencer wedges the current shard configuration and issues a new configuration with the new replica at the tail of the chain of the old configuration.

By construction, replicas in a new configuration can inherit the state of any wedged replica in the predecessor configuration. In our implementation, replicas of a new shard configuration issue an anycast request to all

	PB/CR	QI	Paxos	Vertical Paxos	CRC Shuttle	ER
minimum # replicas needed	$f+1$	$2f+1$	$2f+1$	$f+1$	$f+1$	$f+1$
strongest consistency provided	linearizable	atomic R/W	linearizable	linearizable	linearizable	linearizable
requires accurate fault detection	yes	no	no	no	no	no
uses timeouts for liveness	yes	no	yes	yes	yes	yes
requires external configuration	no	no	no	yes	yes	no
signatures required	none	none	none	none	CRC	none

Table 1: Comparison of Primary-Backup (PB), Chain Replication (CR), Quorum Intersection (QI), Paxos, Vertical Paxos, Byzantine Chain Replication (CRC Shuttle), and Elastic Replication (ER) protocols.

replicas of the predecessor configuration and adopt the history of the first responder.

If a sequencer fails during reconfiguration, system safety is not compromised. This is because elastic replication requires all replicas of a configuration to be active for any replica to change its persistent history. Replicas of a configuration become active only by successfully inheriting the history of a replica in a prior configuration. Thus, the worst thing a failed sequencer can do is to neglect to notify all the new replicas of a configuration to inherit the state of their predecessor configuration which would result in an inactive shard configuration.

Elastic replication assumes that clients learn about shards and their configuration through out-of-band routing methods. Clients include the expected configuration *id* with their requests. If a replica receives a request with an out-of-date configuration *id*, it ignores the request and responds to the client with its current configuration.

3.7 Comparing ER with Other Protocols

We now briefly discuss how elastic replication (ER) compares to other crash-tolerant replication protocols such as Primary-Backup (PB) [1, 9], Chain Replication (CR) [41], asynchronous consensus techniques such as Paxos [24], Vertical Paxos [25], and Quorum Intersection (QI) techniques that provide strong consistency such as [39, 18, 3]. We also compare against the CRC Shuttle implementation of Byzantine Chain Replication which only tolerates crash and omission failures [40]. A summary is provided in Table 1.

PB and CR depend on accurate failure detection (*aka* fail-stop [35]). However, in practice, failure detection is implemented using timeouts. If timeouts are chosen conservatively, the probability of mistakes is small, but in the case of a failure, service availability is substantial. If timeouts are chosen aggressively short, outages are short, but mistakes are common and lead to inconsistencies (multiple primaries in the case of PB, or multiple chains in the case of CR).

Paxos, Vertical Paxos, CRC Shuttle, and ER do not suffer from this safety defect since they do not depend on failure detection being accurate. However, they all rely on failure detection for liveness. In Paxos, a faulty leader must eventually be replaced by another causing a so-called *view change*, while in Vertical Paxos, CRC Shuttle and ER, a faulty replica must eventually be replaced, leading to a reconfiguration. The time to recover from a failure depends much on the timeout used to detect failures. If chosen conservatively, outages can be substantial in either case. But aggressively chosen timers can lead to starvation. In practice, timers should be adaptive so they are as short as practical, but no shorter. Compared to ER, Paxos requires $2f+1$ replicas instead of only $f+1$[1]. Vertical Paxos requires only $f+1$ replicas for the read/write quorums but it relies on an auxiliary configuration master for reconfiguration. Byzantine Chain Replication provides two protocols: CRC Shuttle (which tolerates crash and omission failures) and HMAC Shuttle (which tolerates arbitrary failures). The specification of CRC Shuttle is similar to a single shard in ER, however it relies on an external configuration manager for reconfiguration while ER does not.

Finally, QI techniques do not rely on accurate failure detection and can completely mask failures altogether, requiring no timeouts. They are, however, limited in what operations they can support, and typically support only read and overwrite operations on file-like objects, and provide atomic read/write register consistency. See [21] for a comparison of quorum replication to other replication techniques. Also, consistent read operations are relatively expensive, as they require two round-trips.

Like Paxos and other consensus techniques, QI protocols are based on majority voting and thus require a relatively large number of replicas. As in Paxos, techniques exist to make f of the replicas light-weight [29]. Nonetheless, such light-weight replicas still require independently failing resources (for a total of $2f+1$). On the other hand, ER leverages the situation that there are many shards, each having $f+1$ replicas. replication.

Any of these techniques can be made scalable through partitioning of data, as both linearizability and atomic

[1] In Paxos terms, there are $2f+1$ acceptors and $f+1$ learners that are typically co-located with acceptors. As acceptors have to maintain state for every command in the history, we consider them replicas.

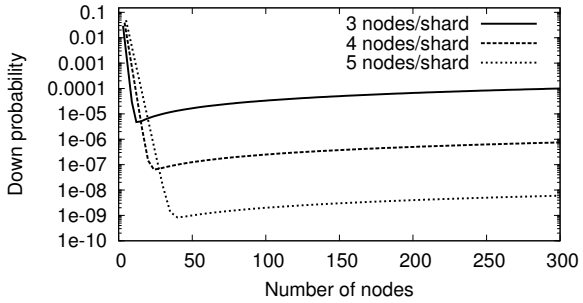

Figure 5: Probability that the liveness condition is not met (band requires external intervention) as a function of the number of nodes and size of shards. Shard replicas are assumed to be up with probability $p = 0.99$.

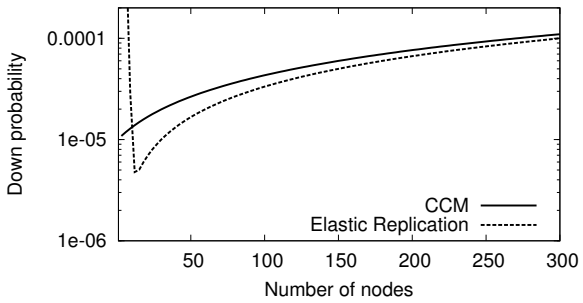

Figure 6: Probability that a system gets stuck requiring external intervention. An elastic replication deployment is compared to a collection of shards managed by a CCM comprised of 5 replicas. Each shard has 3 replicas. All nodes are assumed to have 99% uptime.

R/W register semantics are composable. Also, all these techniques support weak forms of consistency for improved efficiency and liveness, for example by allowing any replica to be read and buffering write operations at any replica [43].

4 Evaluation

In this section we evaluate elastic replication analytically and experimentally. First, we analytically formulate the probability that the liveness condition for elastic replication is met, show its implication on system reliability, and compare the reliability of elastic replication with traditional methods using CCMs. Next, we benchmark a prototype distributed key-value store built using elastic replication .

4.1 Liveness Condition

In elastic replication, we require for liveness that (A1) for any band, each shard contains at least one correct replica, and that (A2) each band has at least one shard that has no faulty replicas. This leads to an interesting trade-off: the smaller the band, the more likely that assumption (A1) holds, whereas the larger the band, the more likely that assumption (A2) holds.

Another variable involved here is n, the number of replicas in a shard. It suffers from a similar trade-off. The larger n is, the more likely that at least one replica is correct, but the less likely that there exists a shard consisting of only correct replicas.

Let p stand for the probability that a particular replica is up, and let all configurations of all shards have a different set of replicas with independent failure probabilities. Let $n = f + 1$ be the number of replicas in a shard. Then let $P_c = p^n$ be the probability that a shard is *correct* (all replicas are up). Let $P_s = \sum_{i=1}^{n-1} \binom{n}{i} p^i (1-p)^{n-i} = 1 - (1-p)^n - P_c$ stand for the probability that a shard is *safe* (between 1 and $n-1$ replicas are up). If N stands for the number of shards on the band, then the probability that at least one shard has only correct replicas while the other shards each have at least one correct replica is:

$$P(p,n,N) = \sum_{i=1}^{N} \binom{N}{i} P_c^i \cdot P_s^{N-i} \tag{1}$$

This is exactly the probability that the liveness condition is met, and typically one would require that $P(p,n,N) \geq (1-\varepsilon)$ for some small constant ε. We analyze this probability in more detail in Appendix B. If the liveness condition is not met, the system would require external intervention to get "unstuck".

To demonstrate the feasibility of meeting the liveness condition, Figure 5 plots $1 - P(0.99,n,N)$ for various shard sizes (n) and number of shards (N). The sharp dip followed by slow rise seen in Figure 5 demonstrates the tension between shard size and number of shards on the probability of meeting the liveness condition.

4.2 Elastic Replication vs. CCM

Elastic replication delegates the task of managing configurations to the shards in the system. Is this more reliable than using a CCM to manage the configuration of shards directly?

A sharded service managed by a CCM still has to meet liveness condition (A1), but condition (A2) is replaced with (A3): the centralized configuration manager does not fail. A replicated CCM running a quorum-based consensus protocol requires more than half of its replicas to be up for liveness. Let m be the number of nodes in a CCM and p the probability that a replica is up, then

$$Q(p,m) = \sum_{i=\lfloor 1+m/2 \rfloor}^{m} \binom{m}{i} p^i \cdot (1-p)^{m-i}$$

	Read	Write
Throughput (ops/sec)	16,385	12,894
Latency (msec)	6.64	8.94

Table 2: Performance of a single stand-alone instance of our example storage server servicing read/write requests of 2KB objects from a network client and storing them in an Erlang DETS on disk.

is the probability that a replicated CCM is up. Then the probability that a CCM-managed system does not require external intervention, is precisely probability that (A1) and (A3) are met:

$$Q(p,m)\cdot\left(1-\sum_{i=1}^{N}\binom{N}{i}(1-p)^{i\cdot n}\cdot(1-(1-p)^{n})^{N-i}\right)$$

Figure 6 compares the likelihood that an external intervention is required (the liveness condition is not met) for an elastic band and a CCM-managed system. As the figure shows, a single self-managing elastic band is more reliable than a CCM for systems with more than 10 replicas.

4.3 Experimental setup

We now aim to highlight the performance characteristics of elastic replication . To that extent, we developed an application library that implements multiple replication protocols. The library defines a state machine interface that user modules implement. The interface is comprised of functions to handle client commands as well as to initialize, export, and import the user module's state.

The library is written in Erlang [2], and our following examples build a simple distributed service that stores and retrieves binary data to/from disk. The service is implemented as a server module that relies on our library for replication and sharding. Our focus is on the protocol layer, so the implementation relies on Erlang's built-in Disk Term Storage abstraction (DETS) for storage[2]. To put our results in perspective, Table 2 shows the performance profile of running a stand-alone instance of our system on a single server servicing read and write requests for 2KB objects from a network client.

Our experiments ran on a 25-machine cluster. Each machine had an Intel Xeon Quad Core processor, 4GB of RAM, a 250GB SATA drive, and a quad port PCIe Intel Gigabit Ethernet card. The machines ran 64-bit Ubuntu 12.04 and Erlang R16B. The machines were connected via a Gigabit Ethernet network. Graph results represent the average of 10 runs and the error bars represent a standard deviation unit.

[2]DETS tables are also used in the Mnesia distributed DBMS

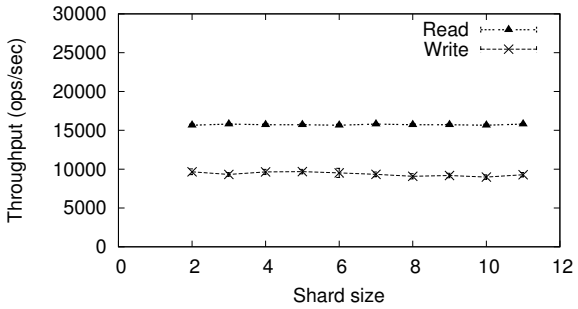

Figure 7: Average read/write throughput of per shard for elastic replication as a function of the number of replicas.

4.4 Single Shard Performance

We first evaluate the performance of a single shard in isolation as a function of the number of its replicas. Figure 7 shows the average read and write throughput for a single shard processing requests from 100 concurrent clients storing and retrieving 2KB objects. Our implementation of elastic replication is based on chain replication and thus exhibits similar throughput characteristics. The throughput of a replicated shard closely matches that of a single unreplicated server.

A concern with elastic replication is that in order to guarantee linearizable consistency in the presence of failures, requests must be propagated to all replicas in a configuration before they get executed. This applies to both read and write requests. How costly is it to propagate read requests to all replicas instead of just one?

To mitigate that potential cost, our implementation tags each request with the index number of the configuration that the client assumes the shard is in. Before forwarding a command, each elastic replica checks that the tagged version number matches its own. If not, the request is dropped and the client is notified of the new configuration. With that, non-tail replicas handle read requests by just checking the request tag before forwarding it to their chain successor (without adding the request to the local unstable history queue or executing). When the tail receives a request, it checks the tag and responds to the client. This is a slight deviation from the original specification of elastic replication which is agnostic to read/write request, but it results in a performance benefit without compromising semantics.

In our implementation, the cost of piping read requests to replicas was dominated by the cost of reading from the disk store at the tail. To demonstrate the cost of propagating read requests, we made our clients send "ping read" requests to the replicated shard. These requests are treated by the replication protocol as a normal read request, but the underlying server responds without engaging the disk store. Figures 8 and 9 show the latency and throughput of handling ping read requests

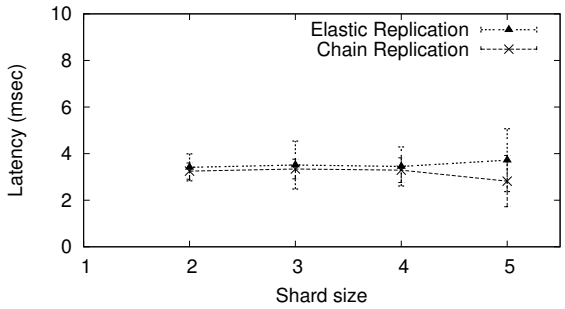

Figure 8: 95th percentile latency for *ping read* requests per shard as a function of the number replicas.

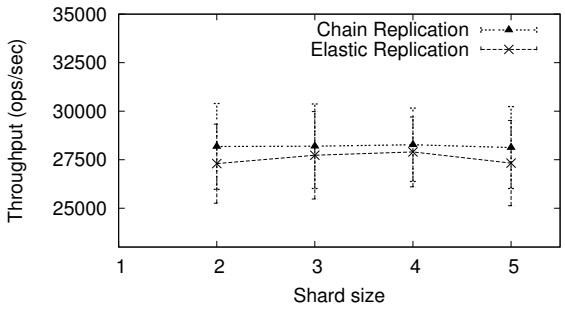

Figure 9: Average throughput of *ping read* requests per shard as a function of the number replicas.

using both elastic replication and chain replication. As the figures demonstrate, the overhead of piping read requests through the chain is small. Elastic replication performs comparably to chain replication. The added cost is well within the margin of error.

The main takeaway here is that elastic replication has the same high performance characteristics of chain replication with minimal overhead cost.

4.5 Band Maintenance

Because a shard's configuration is separated from the actual state, reconfiguration in elastic replication is cheap. Figure 10 shows a simple elastic band being reconfigured to add/remove a shard every 20 seconds. This operation is trivial and barely has any impact on system throughput beyond the effects of adding or removing a shard.

Another technique for reconfiguring an elastic band is by way of merging/splitting shards. This allows part of the state of a large shard to be transfered to a new shard. Figure 11 shows the impact of splitting/merging a shard every 20 seconds on an elastic band. The cost of merging shards is more considerable than just removing them because of state transfer. However, in our implementation, state is transfered in the background to mitigate the potential high cost.

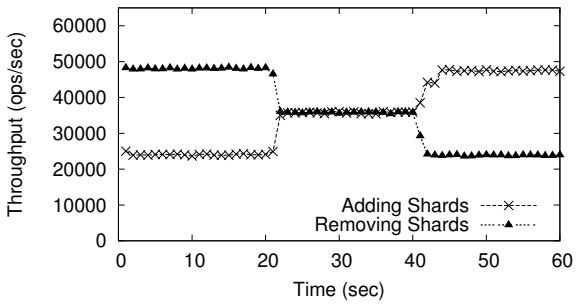

Figure 10: Impact of reconfiguration to add/remove shards on aggregate throughput. A partition was added/removed to the system every 20 seconds. When adding shards, the band originally had 2 shards, and when removing it started with 4 shards.

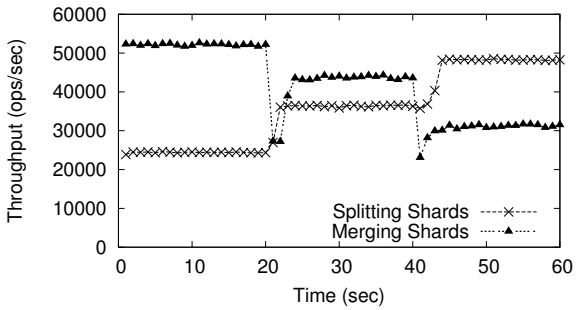

Figure 11: Impact of reconfiguration to split/merge shards on aggregate throughput. A partition was added/removed to the system every 20 seconds. When adding shards, the band originally had 2 shards, and when removing it started with 4 shards.

4.6 Shard Maintenance

When a replica is being unresponsive, the configuration that contains that replica needs to be modified. In order to restore service immediately, the new configuration is simply the old configuration with the unresponsive replica removed. The overhead is the same as that of any update operation on the sequencer shard. Next, in order to restore the original fault tolerance, a new replica needs to be allocated. One would take into account placement for optimal diversity, and load balancing considerations. Then a second reconfiguration needs to take place to add the new replica to the configuration.

Figure 12 shows the impact of adding/removing replicas to a single shard through reconfiguration. The temporary drop in throughput is due to the original configuration being wedged. Because the choice of timeout values does not affect system safety, clients were using long 5 second timeouts to retry their requests. Requests to the old configuration (before adding/removing replicas) would timeout. When a client's request times out, it tries to refresh its configuration information by broadcasting a request to all the replicas of the old shard configuration. By our assumptions, at least one replica of the old

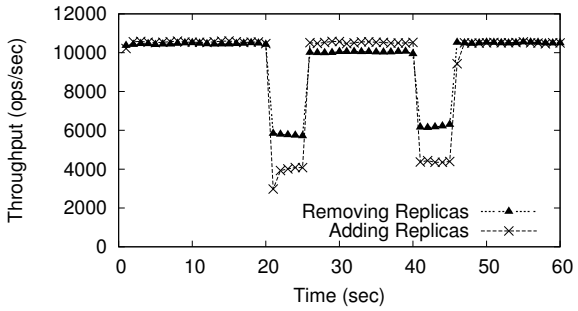

Figure 12: Impact of reconfiguration to add/remove replicas to a shard. Replicas were added/removed every 20 seconds.

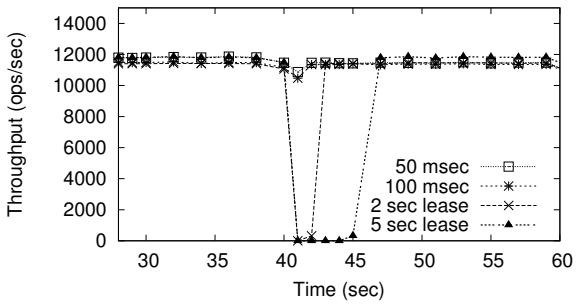

Figure 13: Failure tolerance within a shard with various timeouts. Failure was induced at the 40th second. Shorter timeouts only result in shorter downtime without compromising safety.

configuration would still be alive and would know of the new configuration and responds to the client. Once the client receives a response detailing the new shard configuration, it retries its request.

4.7 Failure Tolerance

Replicas in each shard monitor the replicas in the shard they are sequencing. When replica failure is suspected, the wedge command is issued to the victim shard by both the suspecting replicas in the same shard and the sequencer shard. This is done to prevent the shard from accepting any new client requests and guarantee that safety is not violated. Next, the sequencer issues a reconfiguration of the wedged shard and spins up new replicas if needed. Figure 13 highlights the fact that because elastic Replication guarantees safety independent of timeout values, aggressively small timeout values can used for failure detection without compromising strong consistency.

5 Conclusion

Elastic replication is a new replication technique for sharded datacenter services. It supports rapid reconfiguration without the use of a centralized configuration manager. Elastic replication is novel in that replicated shards are responsible for each others' configurations. Each shard uses only $f+1$ replicas to tolerate up to f crash failures at a time. Elastic replication guarantees strong consistency in the presence of reconfiguration and without relying on accurate failure detection. We have specified our new protocol, analyzed its reliability properties, and benchmarked a prototype implementation. We believe elastic replication is a simple mechanism for providing strong consistency, high availability and fast reconfiguration to today's large scale services.

Acknowledgments

We would like to thank Haoyan Geng, Ken Birman, our shepherd Steve Gribble, and the anonymous reviewers for their valuable feedback. This work was funded, in part, by grants from DARPA, NSF, ARPAe, Amazon.com and Microsoft Corporation. The last author was partially supported by Grant-of-Excellence #120032011 from the Icelandic Research Fund.

Appendix A: Safety

We show that the `Replicas` specification refines the high-level `Shard` specification. The collective states of the replicas maps to the state of the shard as follows: $\mathscr{H} = \max_{C \in \mathscr{C}} gcp(C)$. The following invariants are easily proven:

- For any replica r, $gcp(\text{r.}conf)$ is a prefix of $r.history$;
- `PENDING` replicas have an empty history;
- If a configuration C contains a `PENDING` replica, then $gcp(\text{C})$ is empty;
- Replicas cannot truncate their stable prefix;
- For any configuration that has an `ACTIVE` or `IMMUTABLE` replica, all the predecessor configurations have at least one `IMMUTABLE` replica.

Unlike other replication protocols such as Primary-Backup, Chain Replication, or State Machine Replication, it is *not* an invariant that, given two histories of two different replicas in a configuration, one is a prefix of the other. However, it is the case that the stable prefix of any replica is always a prefix of the histories of the other replicas.
An important invariant that holds over the states of replicas is the following:

Stable Prefix Invariant: The stable prefix of a replica's history is a prefix of the history of each of its non-`PENDING` peers.

Proof. Initially, all histories are empty, and so the invariant holds in the initial state. We will now show that each transition maintains the invariant.

1. `addOp`(r,o): This transition does not change the stable prefix of any replica;
2. `adoptHistory`(r): This transition copies the history and stable prefix from the orderer. The orderer's stable prefix satisfies the invariant, and thus the transition maintains the invariant;
3. `learnPersistence`(r,s): This transition, as a precondition, requires that s corresponds to a prefix of all histories of the peers, and thus evidently enforces the invariant for the state after the transition;
4. `wedgeState`(r): This transition does not change the stable prefix of any replica;
5. `inheritHistory`(r,r'): This transition can only happen to replicas that are in `PENDING` mode. Let $C = r.conf$ and $C' = r'.conf$. Observe that the `learnPersistence` transition can only happen in a configuration that has no `PENDING` replicas, as the history of a `PENDING` replica is empty. Thus we know that the `learnPersistence` transition cannot have occurred at replicas of C. Also note that the initial configuration has no `PENDING` replicas, and thus C not the initial configuration.

   All non-`PENDING` replicas must thus have obtained their history from an immutable replica in C' (`inheritHistory` transition) or from the orderer of C (`adoptHistory` transition). As the `addOp` transition does not affect the invariant, we will for simplicity of exposition assume that no operations have arrived at the orderer. Because the orderer must have obtained its history from an immutable replica of C' *all* histories at non-`PENDING` replicas are from immutable replicas in C'. Since the invariant holds in C', and since the histories of `PENDING` replicas are empty, the invariant must also hold in C after this transition.

As none of the possible transitions result in a state that violate the property, the property is invariant. ■

Lemma 1. *For any configuration C and any non-*`PENDING` *replica r in the successor configuration of C, there exists an* `IMMUTABLE` *replica in C whose history is a prefix of $r.history$.*

Proof. If r is the orderer, then its history must have first been inherited from an immutable replica in C, and then possibly extended with new operations (using `addOp` transitions). Because no transition can truncate the history of the orderer, the lemma holds. If r is not the orderer, then its history was either directly inherited from an immutable replica in C, or it was adopted from the orderer. In either case, the lemma holds. ■

Corollary 2. *For any configuration C and any non-*`PENDING` *replica r in the successor configuration of C, $gcp(C)$ is a prefix of $r.history$.*

Corollary 3. *For any two configurations C and C', C before C' in $\mathscr{C}$, and r a non-*`PENDING` *replica of C', $gcp(C)$ is a prefix of $r.history$.*

Lemma 4. *For any two configurations C and C', either $gcp(C)$ is a prefix of C' or vice versa.*

Proof. Note that if either C or C' contains `PENDING` replicas, then its *gcp* is empty and the lemma holds trivially. If neither contains `PENDING` replicas, then the lemma follows from the corollary above. ■

Recall the refinement mapping $\mathscr{H} = \max_{C \in \mathscr{C}} gcp(C)$. Lemma 4 shows that the refinement mapping is well-defined, in that there is a unique maximum `gcp` among the configurations.

Theorem 5. *Specification* `Replicas` *refines specification* `Shard`.

Proof. By induction on the number of transitions. For the initial state, all replica's histories are empty and thus map to the empty shard history $\mathscr{H}$. We now show that each `Replicas` transition corresponds to a transition in `Shard` or leaves the state of `Shard` unchanged (a so-called *stutter*). A transition at a replica r in configuration C may change $gcp(C)$, but cannot cause an inconsistency in $\max_{C \in \mathscr{C}} gcp(C)$ because of Lemma 4. Examining each transition:

1. `addOp`(r,o): If the transition extends $\max_{C \in \mathscr{C}} gcp(C)$ by adding o, then the transition corresponds to `apply`$([o])$. Otherwise it is a stutter;

2. `adoptHistory`(r): Because r adopts the history of another replica, it cannot cause $gcp(r.conf)$ to be truncated. If the transition causes a sequence S of operations to become persistent, the transition corresponds to `apply`(S). Otherwise the transition is a stutter;

3. `learnPersistence`(r,s): This transition does not affect the refinement mapping, and thus is a stutter transition;

4. `wedgeState`(r): This transition does not affect any histories, and therefore is a stutter;

5. `inheritHistory`(r,r'): r starts out with an empty history, and thus the transition cannot cause $gcp(C)$ to be truncated. If the transition causes a sequence S of operations to become persistent, the transition corresponds to `apply`(S). Otherwise the transition is a stutter. ■

Appendix B: Liveness

We now analyze the probability of meeting the liveness conditions in more detail. For the sake of analysis, we first assume a failure model in which replicas are either up or down, and that each replica is up with probability $\frac{1}{2} \leq p < 1$ independently of other replicas in the system.

When we run our protocol on a band with a large number of nodes T, we must balance the number of shards N and per shard size $n = T/N$. Treading the balance is not easy: when shards are small, it is more likely that all replicas of an individual shard fail (violating A1), whereas when shards are few and large, there is greater chance of each shard suffering a replica failure and becoming wedged (violating A2). Let us define these violations as bad events $\overline{A1}$: all members of some shard fail, and $\overline{A2}$: every group suffers at least one node failure. In the parlance of Section 4.1, we wish to maximize $P(p,n,N) = 1 - \Pr[\overline{A1} \text{ or } \overline{A2}]$.

Theorem 6. *There is a number $c > 0$ depending only on p such that $Pr[\overline{A1} \text{ or } \overline{A2}] \to 0$ when $n = c \log T$ and $T \to \infty$.*

Here, $\log(\cdot)$ denotes the natural logarithm of a number. The result implies that to minimize the chance of wedging or complete configuration failure, the shard size be $n = \Theta(\log T)$ and there should be $N = \Theta\left(\frac{T}{\log T}\right)$ shards on each band.

Proof. By definition, $\Pr[\overline{A1}] = (1-p^n)^N$ and $\Pr[\overline{A2}] = 1-(1-(1-p)^n)^N$. We observe that by varying n and N, these probabilities undergo a phase transition between converging to 0 or 1 in the limit of large T. We show that when n is $\Theta(\log T)$, the two probabilities simultaneously converge to zero, and so $\Pr[\overline{A1} \text{ or } \overline{A2}] \leq \Pr[\overline{A1}] + \Pr[\overline{A2}] \to 0$ as $T \to \infty$.

Let $\alpha = \log \frac{1}{1-p}$ and $\beta = \log \frac{1}{p}$. Choose c such that $\frac{1}{\alpha} \leq c \leq \frac{1}{\beta}$. This is always possible because $p \geq \max\{\frac{1}{2}, 1-p\}$. Assume $n = c \log T$ and thus $N = \frac{T}{c \log T}$. We obtain the following.

$$\begin{aligned}
\lim_{T\to\infty} \Pr[\overline{A2}] &= \lim_{T\to\infty} \left(1 - p^{c\log T}\right)^{\frac{T}{c\log T}} \\
&= \lim_{T\to\infty} \left(1 - \frac{1}{T^{c\beta}}\right)^{\frac{T^{c\beta}\cdot T^{1-c\beta}}{c\log T}} \\
&= \lim_{T\to\infty} e^{-\frac{T^{1-c\beta}}{c\log T}} = 0
\end{aligned}$$

since $\lim_{k\to\infty}\left(1-\frac{1}{k}\right)^k = \frac{1}{e}$ and $1 - c\beta > 0$.

Similarly,

$$\begin{aligned}
\lim_{T\to\infty} \Pr[\overline{A1}] &= 1 - \lim_{T\to\infty} \left(1 - (1-p)^{c\log T}\right)^{\frac{T}{c\log T}} \\
&= 1 - \lim_{T\to\infty} \left(1 - \frac{1}{T^{c\alpha}}\right)^{\frac{T^{c\alpha}\cdot T^{1-c\alpha}}{c\log T}} \\
&= 1 - \lim_{T\to\infty} e^{-\frac{T^{1-c\alpha}}{c\log T}} = 0
\end{aligned}$$

since $1 - c\alpha < 0$. ■

Whereas the proof shows the asymptotic behavior of elastic bands, Eq. (1) provides an exact probability for when elastic replication requires external intervention. However, neither expression is convenient to use in practice. Below, we give a closed form estimate on the probability of impeding liveness of the protocol as we vary the parameters p, n and N. The expression can guide operators in configuring elastic replication that fit the shard set-up and failure probability of their servers.

Using $\left(1-\frac{1}{k}\right)^k \leq \frac{1}{e}$ and $\left(1-\frac{1}{k}\right)^{k-1} \geq \frac{1}{e}$ for $k \geq 2$ from basic calculus, we can derive the following lower bound on $P(p,n,N)$ using the techniques from the previous proof.

Corollary 7. *Let $p \geq \frac{1}{2}$ and set $\alpha = \log \frac{1}{1-p}$ and $\beta = \log \frac{1}{p}$. If the number of shards follows $N = c \log T$ and each shard has $n = \frac{T}{c\log T}$ replicas for some c such that $\frac{1}{\alpha} \leq c \leq \frac{1}{\beta}$ and large number of nodes T, then:*

$$P(p,n,N) \geq e^{-\frac{T}{(T^{c\alpha}-1)c\log T}} - e^{-\frac{T^{1-c\beta}}{c\log T}} = e^{-\frac{n}{(nN)^{c\alpha}-1}} - e^{-\frac{n}{(nN)^{c\beta}}}.$$

□

References

[1] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proc. of the 2nd Int. Conf. on Software Engineering (ICSE'76)*, pages 627–644, San Francisco, CA, Oct. 1976. IEEE.

[2] J. Armstrong. The development of Erlang. In *Proc. of the SIGPLAN Int. Conf. on Functional Programming*, pages 196–203. ACM Press, 1997.

[3] H. Attiya, A. Bar Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121–132, 1995.

[4] P. Bailis and A. Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *CACM*, 56(5):55–63, May 2013.

[5] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234. www.cidrdb.org, 2011.

[6] N. Belaramani, J. Zheng, A. Nayte, M. Dahlin, and R. Grimm. PADS: A Policy Architecture for building Distributed Storage systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[7] K. Birman, D. Malkhi, and R. Van Renesse. Virtually Synchronous Methodology for Dynamic Service Replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.

[8] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, Austin, TX, Nov. 1987. ACM Press.

[9] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed systems (2nd Ed.)*. ACM Press/Addison-Wesley, New York, NY, 1993.

[10] M. Burrows. The Chubby Lock Service for loosely-coupled distributed systems. In *7th Symposium on Operating System Design and Implementation*, Seattle, WA, Nov. 2006.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally-distributed database. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, Oct. 2012. USENIX.

[12] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, Sept. 1985.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of 21st Symposium on Operating Systems Principles*, 2007.

[14] M. Fischer, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[15] S. Ghermawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[16] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, Oct. 2011.

[17] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the Twelfth ACM Symp. on Operating Systems Principles*, pages 202–210, Litchfield Park, AZ, Nov. 1989.

[18] M. Herlihy. A quorum consensus replication method for abstract data types. *Trans. on Computer Systems*, 4(1):32–53, Feb. 1986.

[19] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[21] R. Jimenéz-Peris and M. Patiño-Martínez. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, Sept. 2003.

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

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[24] L. Lamport. The part-time parliament. *Trans. on Computer Systems*, 16(2):133–169, 1998.

[25] L. Lamport, D. Malkhi, and L. Zhou. Brief announcement: Vertical Paxos and Primary-Backup replication. In *Proc. of the 28th ACM Symp. on Principles of Distributed Computing*, Aug. 2009.

[26] L. Lamport and M. Massa. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, Washington, DC, 2004. IEEE Computer Society.

[27] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Scalable causal

consistency for wide-area storage with COPS. In *Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, Oct. 2011.

[28] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT Laboratory for Computer Science, June 2004.

[29] J.-F. Paris. Voting with witnesses: A consistency scheme for replicated files. In *Proc. of the 6th Int. Conf. on Distributed Computer Systems*. IEEE, 1986.

[30] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proc. of the 1991 ACM SIGMOD Int Conf. on Management of Data*, pages 377–386, 1991.

[31] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 2. ACM, 2008.

[32] R. Rodrigues and B. Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Technical Report MIT-LCS-TR-992, MIT Laboratory for Computer Science, Dec. 2003.

[33] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *Proc. of the 10th ACM SIGOPS Eur. Workshop*, Sept. 2002.

[34] S. Sankararaman, B.-G. Chun, C. Yatin, and S. Shenker. Key consistency in DHTs. Technical Report UCB/EECS-2005-21, UC Berkeley, 2005.

[35] R. Schlichting and F. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *Trans. on Computer Systems*, 1(3):222–238, Aug. 1983.

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

[37] T. Shafaat, M. Moser, A. Ghodsi, T. Schütt, S. Haridi, and A. Reinefeld. On consistency of data in structured overlay networks. In *Proceedings of the 3rd CoreGRID Integration Workshop*, Apr. 2008.

[38] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[39] R. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proc. of COMPCON 78 Spring*, pages 88–93, Washington, D.C., Feb. 1978. IEEE Computer Society.

[40] R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. In *OPODIS*, Rome, Italy, December 2012.

[41] R. van Renesse and F. Schneider. Chain Replication for supporting high throughput and availability. In *6th Symp. on Operating Systems Design and Implementation (OSDI '04)*, Dec. 2004.

[42] W. Vogels. Eventually consistent. *ACM Queue*, 6(6), Dec. 2008.

[43] H. Yu and A. Vahdat. The cost and limits of availability for replicated services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, Oct. 2001.