2017 IEEE 37th International Conference on Distributed Computing Systems

Elastic Paxos: A Dynamic Atomic Multicast Protocol

Samuel Benz Fernando Pedone
Università della Svizzera Italiana (USI)
Switzerland

***Abstract*—Replication is a common technique used to design reliable distributed systems by masking defective components. To cope with the requirements of modern Internet applications, replication protocols must allow for throughput scalability and dynamic reconfiguration, that is, on-demand replacement or provisioning of system resources. This paper describes Elastic Paxos, a new dynamic atomic multicast protocol that fulfills these requirements. Elastic Paxos allows to dynamically add and remove resources to an online partially replicated state machine. We implemented Elastic Paxos and evaluated its performance in OpenStack, a cloud environment. We demonstrate its practicality to dynamically scale up and down a partially replicated data store with it and to reconfigure a distributed system.**

***Keywords*—atomic multicast; Paxos; scalability; dynamic reconfiguration**

I. Introduction

Today's on-demand computing resources, common in public cloud environments, provide operators of distributed systems with the possibility to react quickly to changes in application workload. Starting up new webservers once increased traffic is detected or switching off low utilized servers to save costs are common operations. Dynamically adding or removing resources when servers are stateful (e.g., databases), however, is much more challenging than reconfiguring stateless servers (e.g., webservers). In fact, building fault-tolerant (replicated) distributed services that provide strong consistency and scalable performance is a daunting task in itself. Further requiring these services to dynamically scale up and down resources introduces additional complexity.

Services are typically made scalable and fault-tolerant by means of state partitioning (sharding) and replication (e.g., [1], [2], [3]). But handling sharded and replicated data in a distributed environment is challenging if services are not willing to give up strong consistency. Strong consistency requires client requests to be ordered across shards and replicas. Atomic multicast is a communication abstraction that can help the design scalable and highly available stateful services [4], [5] by consistently ordering requests. Therefore, much of the complexity involved in designing scalable and fault-tolerant services is encapsulated by atomic multicast.

Nevertheless, existing atomic multicast protocols are *static*, in that creating new multicast groups at run time is not supported. Consequently, replicas must subscribe to multicast groups at initialization, and subscriptions and unsubscriptions can only be changed by stopping all replicas, redefining the subscriptions, and restarting the system. This paper presents Elastic Paxos, the first *dynamic* atomic multicast protocol. Elastic Paxos allows replicas to dynamically subscribe to and unsubscribe from atomic multicast groups.

Dynamic subscriptions in Elastic Paxos should not be confused with dynamic reconfiguration. In dynamic reconfiguration (e.g., [6], [7], [8], [9], [10]), the goal is to change the set of participants of a system (e.g., group membership). Elastic Paxos seeks to allow replicas to dynamically change the multicast groups they subscribe to, while the membership of the system may remain constant. Interestingly, we show in the paper that one can use dynamic subscriptions to reconfigure a system.

In brief, our dynamic atomic multicast protocol composes multiple sequences of Paxos [11], [12], where each sequence is referred to as an *atomic multicast stream*, to provide efficient message ordering. The protocol ensures that no two replicas order the same requests in different orders and allows to add and remove additional streams during run time. To illustrate the design of a scalable and highly available prototypical service, we consider a storage service (i.e., a key/value store). The storage is partitioned into disjoint partitions and each partition is replicated by a group of replicas. There is one atomic multicast stream per partition, which the replicas of the partition subscribe to, and one atomic multicast stream that is shared by all replicas. The storage service supports two types of operations: single-partition operations (i.e., get and put on a single key) and multi-partition operations (i.e., a consistent get range operation that returns all keys in a specified interval). Single-partition operations are multicast to the replicas of the partition that contains the accessed key; multi-partition operations are multicast to all replicas, using the shared atomic multicast stream.

This paper makes the following contributions. (i) We introduce Elastic Paxos, an atomic multicast protocol that supports dynamic subscriptions. (ii) We show how Elastic Paxos can be used to design strongly consistent, scalable and highly available dynamic services. (iii) We detail the implementation of our new protocol. (iv) We evaluate the performance of Elastic Paxos with three practical use cases: (a) How to dynamically remove bottlenecks of atomic broadcast with the online addition of resources. (b) How to split or combine shards of a partitioned data store. (c) How to reconfigure atomic broadcast.

The rest of this paper is structured as follows. Section II describes our system model and assumptions. Section III explains why system designers must care about atomic multicast as a middleware service and introduces how atomic multicast can be used to design scalable services. Section IV highlights why dynamic subscription is required, Section V presents the design of our protocol and Section VI explains how they were implemented. Section VII assesses the performance of the components, Section VIII presents related work and Section IX concludes this paper.

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.84

II. System model

We assume a distributed system composed of a set $\Pi = \{p_1, p_2, ...\}$ of interconnected processes that communicate through point-to-point message passing. Processes may fail by crashing and subsequently recover, but do not experience arbitrary behavior (i.e., no Byzantine failures). Processes are either *correct* or *faulty*. A correct process is eventually operational "forever" and can reliably exchange messages with other correct processes. A faulty process is a process that is not correct. In practice, "forever" means long enough for processes to make some progress (e.g., terminate one instance of consensus).

The protocols in this paper ensure safety under both asynchronous and synchronous execution periods. To ensure liveness, we assume the system is *partially synchronous* [13]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous, called the *Global Stabilization Time (GST)* [13], is unknown to the processes. When the system behaves asynchronously (i.e., before GST), there are no bounds on the time it takes for messages to be transmitted and actions to be executed; when the system behaves synchronously (i.e., after GST), such bounds exist but are unknown by the processes.

III. Background

Atomic multicast is a communication abstraction that helps design highly available and scalable applications.We start by defining atomic multicast and atomic broadcast (§III-A). Then, we explain how from atomic broadcast one can implement atomic multicast (§III-B). We conclude with the design of a highly available and scalable store service developed with atomic multicast, which will be used throughput the paper (§III-C).

A. Atomic multicast

Atomic multicast is an abstraction used by processes to communicate. It defines two communication primitives: *multicast*(S, m) and *deliver*(m). Client processes invoke *multicast*(S, m) to submit requests, encoded in message m, to the replica processes that subscribe to *atomic multicast stream* S. Replicas subscribe to one or more streams and deliver client requests with primitive *deliver*(m).

Intuitively, atomic multicast ensures that if a process delivers message m multicast to stream S, then all correct processes that subscribe to S also deliver m; and if processes p and q deliver messages m and m', then they deliver them in the same order. Atomic broadcast is a special case of atomic multicast where there is a single stream of messages to which all replicas subscribe.

B. From broadcast to multicast

Atomic multicast can be implemented with independent instances of Multi-Paxos, where each Multi-Paxos instance corresponds to an atomic multicast stream [5], [14]. Replicas can subscribe to one or more atomic multicast streams and a replica that subscribes to stream S_i becomes a learner in the Multi-Paxos instance associated with S_i.

Replicas that subscribe to multiple streams ensure ordered delivery of messages by implementing a deterministic round-robin procedure that merges messages ordered in different streams (*dMerge*).

To handle imbalanced traffic among streams and ensure that messages will not be delivered at the pace of the slowest stream, processes can skip Paxos executions in a stream. Periodically, the group coordinator of a stream calculates the number of executions that need to be skipped for the stream to reach a virtual maximum throughput, measured in Paxos executions per second. Then, the coordinator proposes in the next available execution to skip some executions [5].

C. Designing scalable services with atomic multicast

Designing services that are highly available and capable to scale throughput without giving up strong consistency is a daunting task. In this paper, we consider strongly consistent services that ensure linearizability. A service is linearizable if there is a permutation of the commands executed by the clients that respects (i) the service's sequential specification and (ii) the real-time precedence of commands [15].

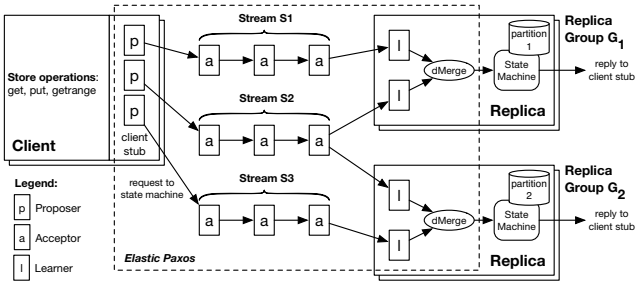

Fig. 1. Architecture overview of a highly available and scalable store service developed with atomic multicast.

State machine replication [16], [17] simplifies the problem of implementing highly available linearizable services by decomposing the *ordering* of requests across replicas from the *execution* of requests at each replica. Requests can be ordered using atomic broadcast and, as a consequence, service developers can focus on the execution of requests, which is the aspect most closely related to the service itself. State machine replication requires the execution of requests to be deterministic, so that when provided with the same sequence of requests, every replica will evolve through the same sequence of states and produce the same results.

State machine replication, however, does not lead to services that can scale throughput with the number of replicas. Increasing the number of replicas results in a service that tolerates more failures, but does not necessarily serve more clients per time unit. Several systems resort to state partitioning (i.e., sharding) to provide scalability (e.g., Calvin [18], H-Store [19]). Scalable performance and high availability can be obtained by partitioning the service state and replicating each partition with state machine replication. To submit a request for execution, the client atomically multicasts the request to the appropriate partitions [4]. Performance will scale as long as the state can be partitioned in such a way that most commands are executed by a single partition only.

Figure 1 illustrates a key-value store service developed with atomic multicast. There are commands to read and write single entries in the store (*get* and *put*) and to query multiple entries (*getrange*). Replicas in G_1 subscribe to streams S_1 and S_2 and replicas in G_2 subscribe to streams S_2 and S_3. Atomic multicast is implemented with Multi-Ring Paxos [5], which pipelines acceptors in a stream. The streams that a replica subscribes to are combined by the dMerge component.

IV. Dynamic Atomic Multicast

Atomic multicast is a suitable abstraction to build scalable distributed systems. But creating new groups during run time is not supported by existing atomic multicast systems. In this section, we motivate and define dynamic atomic multicast (§IV-A, §IV-B).

A. Motivation

Atomic multicast, as discussed in the previous section, relies on static subscriptions of replicas to streams, that is, subscriptions are defined at initialization and can only be changed by stopping all processes, redefining the subscriptions, and restarting the system.

In today's cloud environments, adding resources to and removing resources from an operational system without shutting it down is a desirable feature. Combining the benefits of atomic multicast and dynamic subscriptions at run time allows several practical use cases, as we describe next.

1) Vertical scalability: Although atomic broadcast is typically implemented with a single message stream, it can be also implemented with multiple streams, as long as all processes subscribe to all streams. When implemented with a single stream, the performance of atomic broadcast will be typically limited by the performance of the coordinator (CPU) or the acceptors (disk write performance) of the stream. However, replicas can increase the throughput of atomic broadcast by dynamically subscribing to multiple streams. In doing so, each stream contributes to the aggregated throughput of atomic broadcast.

2) Horizontal scalability: Scaling out a key-value store service can be achieved by horizontally partitioning (sharding) the service state. Partitioned state introduces the problem of how to ensure consistency of cross-partition queries. Paxos and other atomic broadcast algorithms ensure total order of commands within one partition (e.g., *get* and *put* commands), consistent cross-partition operations (e.g., *getrange*) must be coordinated using additional mechanisms, such as two-phase commit and synchronized clocks (e.g., [1]). Atomic multicast offers an alternative by ordering both single-partition and cross-partition commands, as needed (i.e., partial order). If replicas can dynamically subscribe to a new stream (i.e., a new partition), then a replicated data store can be repartitioned without service interruption.

3) Reconfiguration: Reconfiguration means changing the set of processes in a distributed system. It is used, for example, to replace a failed server or a server whose disk is full. Reconfiguring a replicated state machine has been considered before (e.g., [6], [7], [8], [9], [10]). In general, existing solutions consist in stopping processes in the current configuration (i.e., the running state machine), redefining the set of processes in the new configuration, and re-starting the processes in the new configuration [9].

In Paxos, the real challenge is reconfiguring the set of acceptors since these are the processes that store the state of Paxos (e.g., accepted values). Moreover, processes must know the set of acceptors of each consensus instance (i.e., system membership). Lamport [11] suggests to manage membership by making the set of acceptors part of the state of the system and handling membership changes as commands, which must also be ordered by consensus. Such a mechanism, however, prevents multiple consensus instances from executing concurrently, which limits performance [10].

Dynamic subscriptions offer an alternative approach to reconfiguring the acceptors in a single stream S_i. We first create a new stream S_i' with the new set of acceptors, then have the learners subscribe to S_i', and finally unsubscribe from S_i. Note that this approach does not impose any constraints on the intersection between S_i and S_i' (e.g., S_i and S_i' can be disjoint sets).

B. Dynamic atomic multicast

After arguing for dynamic subscriptions in atomic multicast, we extend the atomic multicast interface with two additional primitives: *subscribe_msg*(G, S) and *unsubscribe_msg*(G, S), which replicas in replication group G can use to subscribe to and unsubscribe from stream S. After replicas subscribe to stream S, they will eventually deliver messages multicast to S. Similarly, if replicas unsubscribe from S, they will eventually stop delivering messages multicast to S. In both cases, atomic multicast guarantees acyclic ordered delivery (see §III-A).

V. Elastic Paxos

In this section, we present an overview of the Elastic Paxos protocol (§V-A), describe Elastic Paxos in detail (§V-B) and introduce a few optimizations (§V-C).

A. Overview

We seek decentralized solutions that properly coordinate dynamic subscriptions in atomic multicast without relying on a single entity, such as an oracle that oversees all subscribe and unsubscribe requests. In the following, we provide an overview of our solution. We describe how a replica R in replication group G can subscribe to and unsubscribe from a stream.

Every replica in G starts with a subscription to a default stream, S_G. In order for R to subscribe to a new stream S_N, R must atomically broadcast request *subscribe_msg*(G, S_N) to (a) the new stream S_N; and (b) a stream S that R currently subscribes to (e.g., the default stream). Upon delivering the subscription request from S, the deterministic merger that executes at R spawns a new learner task at R for stream S_N. The new learner starts by recovering all messages in S_N until it reaches the subscribe request *subscribe_msg*(G, S_N).

When the subscribe request is ordered in both streams S and S_N, the merger determines the "merge point", that is, the instance after which the replica will start combining messages

from the new stream with messages from the currently subscribed streams. To avoid order violations, Elastic Paxos uses the *same instance* in both streams, computed as the maximum between the instances in which the subscribe request was delivered in each stream (see Figure 2). Intuitively, this works because the merge point is "aligned" at all subscribed streams.

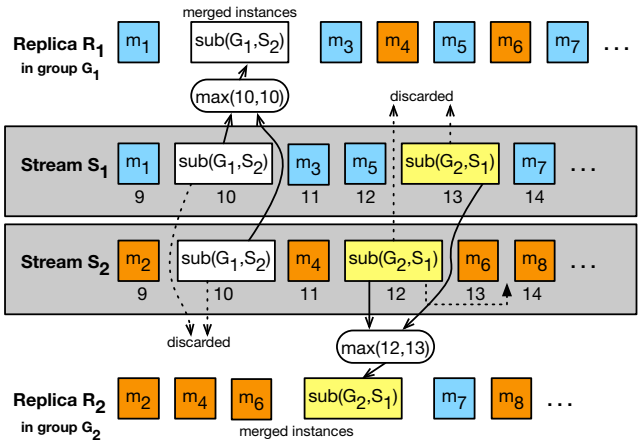

Fig. 2. How Elastic Paxos ensures acyclic ordering.

Unsubscriptions are simpler than subscriptions because there is already a total order among messages in all subscribed streams. Therefore, it is enough to broadcast a single *unsubscribe_msg(group,stream)* request to any of the subscribed streams. As soon as the request is delivered, the dMerge task removes the requested stream from the set of streams the replica subscribes to.

B. Detailed protocol

Algorithm 1 details how a replica R in replication group G subscribes to a new stream S_N. Every replica consists of multiple tasks. There is one dMerge task, and one learner task per subscribed stream. The dMerge task orders messages from the various streams a replica subscribes to and handles subscription and unsubscription requests. dMerge holds an array of stream queues (Q), from which it deterministically (round-robin) delivers decided values. Every stream queue is filled by a background learner task. When a replica subscribes to a new stream, one more learner task is created. This new learner will recover (Section VI) all decided values and put them in Q.

For every queue, dMerge keeps a pointer per stream (ptr) with the position of the last ordered value in the stream that has already been delivered to the application. The subscription point is the maximum stream position of the two subscription messages (i.e., the new stream and the currently subscribed stream). Round-robin delivery from the new stream will start in the round after the maximum stream position.

For the sake of simplicity, in Algorithm 1 a stream position corresponds to a Paxos instance. In our prototype, the stream position is not related to the decided Paxos instances. Since multiple values or skip messages can be decided in one Paxos instance (batching), in our prototype the pointer refers to a value, after discarding skip messages.

Algorithm 1 Replica R in G subscribes to stream S_N

```
1:  Initialization:
2:    Q[1..max_stream][1..max_instance] ← ⊥, ⊥, ...
3:    start task dMerge                          {init deterministic merge}
4:  task dMerge                                  {Deterministic merge}
5:    Initialization:
6:      Σ ← {S_G}        {set of subscribed streams, with default stream}
7:      start task Learner(S_G)                  {start the first learner}
8:      S ← S_G                                  {set first stream}
9:      ptr[S] ← 0                               {next instance in a stream}
10:   while forever do                           {round-robin delivery}
11:     ptr[S] ← ptr[S] + 1        {set pointer to next message in S}
12:     wait until Q[S][ptr[S]] ≠ ⊥
13:       v ← Q[S][ptr[S]]
14:     if v = subscribe_msg(G_x, S_x) and G_x = G then
15:       S_N ← S_x
16:       start task Learner(S_N)
17:       while Q[S_N][ptr[S_N]] ≠ v do {find same subscribe...}
18:         ptr[S_N] ← ptr[S_N] + 1      {...msg in both streams}
19:       merge_ptr ← max_ptr(ptr) + 1
20:       while ptr[S_N] < merge_ptr do          {align stream}
21:         ptr[S_N] ← ptr[S_N] + 1                {skip}
22:     else
23:       if v ≠ subscribe_msg(G_x, S_x) then
24:         deliver v        {v is ordered, pass it to the application}
25:     if ∀S ∈ Σ : ptr[S] = merge_ptr then
26:       Σ ← Σ ∪ {S_N}                {update current subscriptions}
27:       S ← first(Σ)   {after subscription start from first group}
28:     else
29:       S ← next(Σ)          {next group for round-robin delivery}
30: procedure max_ptr(ptr)
31:   // return maximum ptr[S] for all streams S in Σ
32:   x ← 0
33:   for each S ∈ Σ do
34:     if ptr[S] > x then x ← ptr[S]
35:   return x
36: procedure first(Σ)
37:   // return the first S in Σ
38: procedure next(Σ)
39:   // return the next (cyclic) S in Σ
40: task Learner(S)                              {Learner of stream S}
41:   Initialization:
42:     ptr[S] ← 0
43:     for i from 1 to max decided instance in S do
44:       Q[S][i] ← recover(i)      {recover all decided instances}
45:   upon deliver(v) do
46:     Q[S][i] ← v      {fill queue while Paxos instances get decided}
47:     i ← i + 1
```

C. Extensions and optimizations

As Algorithm 1 shows, after receiving a subscribe request, the dMerge task interrupts the handling of messages until the same request is received in the new stream. Since the dMerge task does not know where in the stream the missing subscription request is, the simplest approach is to scan all previous messages. This procedure can be optimized if the process that

triggers a subscription first broadcasts a hint to learners. Upon receiving such a hint (*prepare_msg*(G, S_N)), learners start scanning the new stream for subscription requests. To recover the stream in the background, we implemented a *prepare* message.

VI. Implementation

To evaluate the capabilities of Elastic Paxos, we extended the URingPaxos library[1] to handle dynamic subscriptions. URingPaxos implements Ring Paxos [20], a high throughput atomic broadcast protocol based on TCP. Further, it implements atomic multicast by combining multiple instances of Ring Paxos [14]. The library is written in Java with some performance critical sections in C (JNI). URingPaxos uses ZooKeeper [21] to store ring management and protocol configuration data. Elastic Paxos replaces the static deterministic merge procedure of URingPaxos with a new procedure (Algorithm 1).

To demonstrate Elastic Paxos in a real application, we extended a partitioned key/value store service [5] with operations to handle *subscribe* and *unsubscribe* events and support for dynamic scalability. Clients can submit *put*, *get*, and *getrange* commands to replicas. Replicas execute the commands to their in-memory data store and reply back directly to the client. Every replica belongs to one hash-partitioned shard of the whole state and every partition has a dedicated Paxos stream to order commands. To achieve linearizability for multi-partition operations, the replicas coordinate their executions with direct *signal* messages [4].

An important part to allow Elastic Paxos is recovery. The URingPaxos library has several mechanism built in to recover and trim Paxos acceptors log and coordinate replica checkpoints and state transfer [5], [22].

Further, we added support to OpenStack. A controller or a client can create or destroy virtual machines, forming additional streams depending on the currently measured application throughput. Adding a new stream from newly created virtual machines (three acceptors) takes approximately 60 seconds.

VII. Experimental Evaluation

In this section, we describe our experimental environment (§VII-A), explain our goals and methodology (§VII-B), and evaluate Elastic Paxos (§VII-C–VII-D–VII-E).

A. Experimental setup

All experiments were performed on SWITCHengines,[2] an IaaS cloud service for academics. The platform uses OpenStack to provide virtual machines and Ceph as a distributed parallel block storage, serving the virtual machines.

The hardware consists of 32 physical machines; 16 are dedicated for compute nodes and 16 act as storage nodes. Every node (Intel S2600GZ) has 256 GB of main memory. The distributed file system uses 128 4 TB (WD4000F9YZ) spinning drives and a replication factor of 3. During our experiments, approximately 500 other virtual machines were running on the cluster.

All virtual machines used in the experiments have 2 vCPU and 2 GB of memory. The network between these VMs is virtualized and tunneled between the physical nodes. Paxos acceptors and replicas are scheduled to different physical machines using the OpenStack anti-affinity host groups. Since the virtual machines do not provide local storage on real disk devices, all experiments were run in memory only.

Multi-Ring Paxos has two important parameters, λ and Δ_t. λ defines the maximum virtual system throughput per stream, measured in Paxos instances per second. Δ_t defines the sampling interval to compare the actual throughput in a stream and λ. In all experiments, λ is set to 4000 and Δ_t to $100ms$.

B. Objectives and methodology

We assess the behavior of Elastic Paxos under a range of different practical deployments, as described next.

- We evaluate the performance of Elastic Paxos when multiple Paxos streams are added dynamically to a set of replicas (§VII-C). This is important in practice whenever the ordering protocol is the bottleneck in a SMR setup.
- We assess how Elastic Paxos can be used with a partitioned key/value store application to dynamically re-partition the replicas under load (§VII-D). Re-partitioning is required whenever the replicas are the bottleneck (e.g., due to CPU saturation).
- We demonstrate how a set of Paxos acceptors can be reconfigured under full system load (§VII-E). This is useful to replace a failed acceptor or an acceptor that runs out of disk storage.

C. Vertical Scalability

In this experiment we demonstrate how Elastic Paxos can be used to dynamically add multiple streams to a single set of replicas.

Setup. We start the experiment with a client VM (5 threads per stream) that sends 32 kbyte values to two replica VMs. We limited the single stream throughput to 30% not to saturate the replicas at the beginning of the experiment. Every 15 seconds replicas subscribe to a new stream and immediately deliver new commands from the added stream. Every stream contains 3 acceptor VMs which are deployed as OpenStack Heat-AutoScaling groups. In this experiment, all VMs are started up from the beginning, but Heat-AutoScaling allows clients to boot up or shutdown the virtual machines that participate in the streams.

Results. Figure 3 shows the aggregated throughput at the replicas. The most visible impact is right after the subscribe message. This is due to the fact that we intentionally do not use the *prepare_msg* request (see V-C) to inform replicas about the changes. During recovery of the new stream, a number of messages are queued up in memory at the replicas and delivered right after the subscription process is over.

[1] https://github.com/sambenz/UringPaxos
[2] http://www.switch.ch/services/engines/

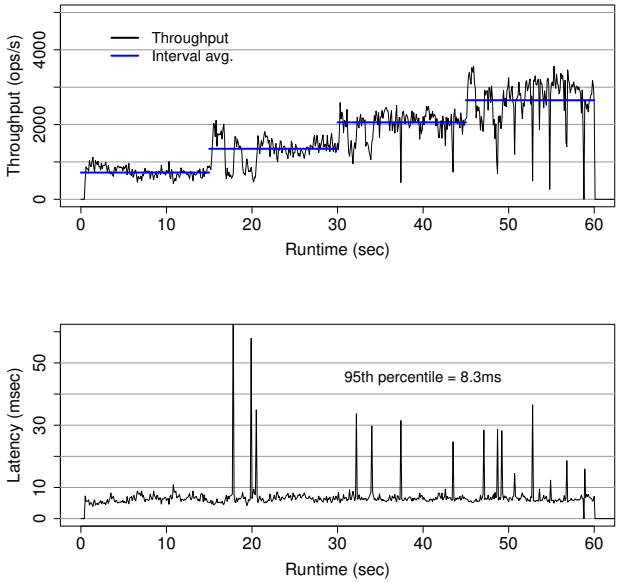

Fig. 3. Dynamically adding streams to a set of replicas to scale up the coordination layer. Every 15 seconds replicas subscribe to a new stream.

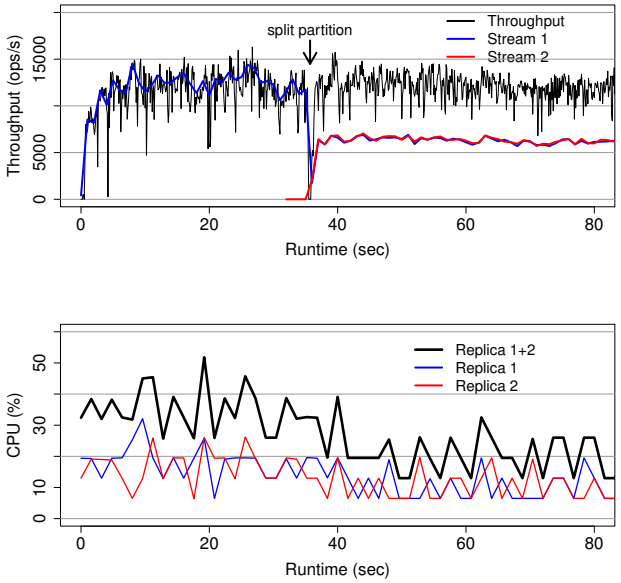

Fig. 4. Re-partitioning of a key/value store (75% peak load). After 35 seconds the throughput and CPU consumption at both replicas decreased.

The interval averages increasing from 735, 1498, 2391 to 2660 ops/s by adding additional streams. With 4 streams, this corresponds to an increase of 3.62 of the system throughput.

D. Horizontal Scalability

In this section we evaluate how Elastic Paxos can be used to dynamically scale out a partitioned key/value store. For this experiment we use the partitioned key/value store described in Section VI.

Setup. We start the experiment with a client VM (100 threads) that sends 1024-byte put commands to random keys. Two replica VMs apply these commands to their local in-memory storage and send back a command response to the client thread. Initially only one partition is present in the system and serves every request. Every partition is coordinated by a stream of 3 acceptor VMs. At 30 seconds, one of the replicas subscribes to a new stream with additional 3 acceptors and informs the whole system 5 seconds later about the partition change. The client is notified about the change in the partitioning by ZooKeeper and starts sending random commands to both partitions.

Results. Figure 4 shows the system throughput during re-partitioning under 75% peak load. The duration of the re-partitioning is 1 second and mainly caused by a client timeout. Commands from clients which are received by the wrong partition after the split are discarded. The clients will resend them after a timeout to the correct partition. The throughput after splitting the partition is half at every replica. Further, also the CPU consumption at every replica drops after the re-partitioning event. Therefore, both partitions could now clearly handle 100% more operations per second.

E. Reconfiguration

In this experiment we show how Elastic Paxos can be used to reconfigure a state machine under full system load. Since reconfiguration of atomic broadcast is a sub problem of reconfigure atomic multicast, we use dynamic subscriptions to replace the set of acting acceptors. Changing the set of acceptors is required, if for example they run out of disk space, one acceptor stable storage is not recoverable or to tolerate more failures (e.g., 5 instead of 3 acceptors). The goal of this experiment is to show, that dynamic subscription is an efficient solution to state machine reconfiguration.

Setup. We start the experiment with a client VM (60 threads) that sends 32 kbyte values to two replica VMs. These two replicas subscribe to the first stream which contains 3 acceptor VMs. After 40 seconds, we inform the replicas that we will add a second stream (with a *prepare_msg* request). After 45 seconds we let the replicas subscribe to the new stream containing 3 different acceptor VMs. Right after the subscribe message we submit a unsubscribe message to the original stream.

Results. Figure 5 shows the reconfiguration under full load of 550 Mbps. Since the replicas received a *prepare_msg* (see V-C), they can start up and recover the new stream in the background without blocking the main message execution. With this optimization, reconfiguration introduces no overhead.

VIII. Related Work

In this section, we briefly review related work on atomic multicast (§VIII-A), group membership (§VIII-B), and state machine reconfiguration (§VIII-C).

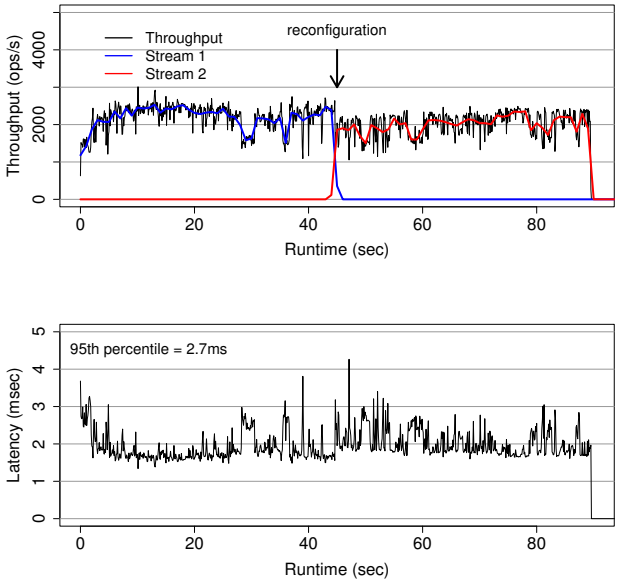

Fig. 5. State machine reconfiguration under full system load. At 45 seconds we replace the set of active acceptors with a new one.

A. Atomic multicast

Atomic multicast has been extensively studied in the literature [23]. In [6], a protocol is proposed for failure-free scenarios. To decide on the final timestamp of a message, each process in the set of message addresses locally chooses a timestamp, exchanges its chosen timestamps, deterministically agrees on one of them, and delivers messages according to the message's final timestamp. Several works have extended this algorithm to tolerate failures [24], [25], [26], [27], where the main idea is to replace failure-prone processes by fault-tolerant disjoint groups of processes, each group implementing the algorithm by means of state machine replication.

Atomic broadcast [28], [29] is a special case of atomic multicast where all messages are targeted to a single group. Paxos [11] is one of many ways to implement atomic broadcast (e.g., [30], [31], [32], [33], [34], [35]).

Elastic Paxos is based on Multi-Ring Paxos [14]. The algorithm to reconfigure the deterministic merge function is similar to [6]. The chosen timestamps for subscribe and unsubscribe messages are the stream positions which are persisted in the streams themselves. Therefore for reconfiguration, Elastic Paxos requires each involved stream (group) to be able to reach consensus.

Spread [36] implements a highly configurable group communication system, which supports the abstraction of process groups. Spread orders messages by the means of interconnected daemons that handle the communication in the system. Processes connect to a daemon to multicast and deliver messages. While the group abstraction is similar to the Totem Multi-Ring protocol [37], Totem uses timestamps to achieve global total order.

E-Cast [38] addresses similar problems like Elastic Paxos does. Compared to Elastic Paxos, E-Cast defines multicast as a stateful routing problem. E-Cast uses replicated routers (sequencers) to partially order messages and reconfigure the system, while Elastic Paxos uses the deterministic merge function in each replica. Therefore, Elastic Paxos does not require a global sequencer to order messages and reconfigure the system.

B. Group membership

Group membership has been an active field of research for decades, in the context of group communication protocols [39]. While in atomic broadcast total order is achieved by a sequence of individual consensus rounds, group communication protocols are based on a sequence of view changes. Group membership is a special case of the set membership problem, in which all processes decide on which non-faulty processes belong to the current set (view) [40].

In Elastic Paxos, the round-robin delivery order can be seen as a dynamic set of changing streams. While the total order within a stream is based on atomic broadcast, the deterministic merge function is based on a sequence of subscription changes, similar to view changes. Compared to group communication protocols, Elastic Paxos does not use view changes to remove faulty replicas, but to dynamically scale. Additionally, the subscriptions in Elastic Paxos are persisted in the streams, every recovering replica can re-learn all subscription changes.

Rollup [41] is a protocol designed for fast cluster membership updates. The main goal is to avoid disruptive behavior when the master or leader of a protocol is replaced. Since Elastic Paxos is based Paxos, frequent changes of the coordinator have an impact on performance. Compared to Rollup, Elastic Paxos is designed to scale atomic multicast groups rather than addressing fast replacement of the Paxos leader.

C. State machine reconfiguration

Changing the set of acting acceptors is discussed in [7], [9]. Elastic Paxos uses a different approach. It does not change the set of the acceptors itself, rather it replaces all of them by a new set (i.e., new stream).

Group communication protocols reconfigure the system to tolerate failures (e.g., process crashes). In general they use a fault-tolerant consensus algorithm to coordinate the view change. As already described, Elastic Paxos uses a similar way to add and remove new streams.

Similar to Elastic Paxos, SMART [10] uses different independent Paxos streams to reconfigure a replicated state machine. But, while SMART changes the set of replicas, Elastic Paxos keeps the replication group constant and changes the subscriptions. This allows Elastic Paxos, additionally to reconfiguration, also to scale by adding multiple Paxos streams to a single replication group. Adding a new replica to a replication group is part of Elastic Paxos's recovery procedure.

Eve [42] implements scalable state machine replication on multi-core servers, but it is static and does not allow reconfiguration. DynaStore [43] allows reconfiguration without consensus and can operate in a completely asynchronous system.However, compared to Elastic Paxos, DynaStore considers a strictly weaker model (i.e., read/write register instead of an arbitrary state machine).

IX. Conclusions

Using on-demand computing resources to elastically grow, shrink, or replace processes in a fault-tolerant distributed system requires dynamic replication protocols. Existing solutions often halt the system during reconfiguration. In this paper we propose a new dynamic atomic multicast algorithm, Elastic Paxos, which was designed for dynamically scaling up and down strongly consistent multicast. We showed the practicality of our argument by dynamically reconfiguring a distributed system. Moreover, the results of our experiments demonstrate both horizontal and vertical scalability of our proposed techniques deployed in a cloud environment.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and their suggestions to improve the paper. This work was supported in part by the Swiss National Science Foundation under grant number 146714.

References

[1] J. 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 *OSDI*, 2012.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *ACM SIGOPS OSR*, vol. 41. ACM, 2007, pp. 159–174.

[3] R. Padilha and F. Pedone, "Augustus: Scalable and Robust Storage for Cloud Applications," in *Eurosys*, 2013.

[4] C. E. Bezerra, F. Pedone, and R. van Renesse, "Scalable State-Machine Replication," in *DSN*, 2014.

[5] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with Atomic Multicast," in *Middleware*, 2014.

[6] K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, pp. 47–76, Feb. 1987.

[7] L. Lamport, D. Malkhi, and L. Zhou, "Stoppable paxos," *TechReport, Microsoft Research*, 2008.

[8] ——, "Vertical Paxos and Primary-backup Replication," in *PODC*, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 312–313.

[9] ——, "Reconfiguring a State Machine," *SIGACT News*, vol. 41, no. 1, pp. 63–73, Mar. 2010.

[10] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," *ACM SIGOPS OSR*, vol. 40, no. 4, pp. 103–115, 2006.

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

[12] R. D. Prisco, B. Lampson, and N. Lynch, "Revisiting the paxos algorithm," *Theoretical Computer Science*, vol. 243, no. 1–2, pp. 35–91, 2000.

[13] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[14] P. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," in *DSN*, 2012.

[15] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[18] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *SIGMOD*, 2012.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, 2008.

[20] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *DSN*, 2010.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," in *ATC*, 2010.

[22] S. Benz, L. Pacheco de Sousa, and F. Pedone, "Stretching Multi-Ring Paxos," in *ACM SAC*, 2015.

[23] X. Défago, A. Schiper, and P. Urbán, "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," *ACM Computing Surveys,*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[24] U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal, "Fault-Tolerant Total Order Multicast to Asynchronous Groups," in *SRDS*, 1998.

[25] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems," *Theor. Comput. Sci.*, vol. 254, no. 1-2, pp. 297–316, 2001.

[26] L. Rodrigues, R. Guerraoui, and A. Schiper, "Scalable atomic multicast," in *ICCCN*, 1998.

[27] N. Schiper and F. Pedone, "On the Inherent Cost of Atomic Broadcast and Multicast Algorithms in Wide Area Networks," in *ICDCN*, 2008.

[28] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[29] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, Department of Computer Science, Tech. Rep., 1994.

[30] I. Moraru, D. G. Andersen, and M. Kaminsky, "Egalitarian Paxos," in *SOSP*, 2012.

[31] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *ATC*, 2014, pp. 305–320.

[32] L. Lamport, "Generalized Consensus and Paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, Mar. 2005.

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

[34] D. Malkhi, M. Balakrishnan, J. D. Davis, V. Prabhakaran, and T. Wobber, "From paxos to CORFU: a flash-speed shared log," *ACM SIGOPS OSR*, vol. 46, no. 1, pp. 47–51, 2012.

[35] F. Pedone and A. Schiper, "Generic Broadcast," in *DISC, formerly WDAG*, Sep. 1999.

[36] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, "The Spread Toolkit: Architecture and Performance," Johns Hopkins University, Tech. Rep., 2004, cNDS-2004-1.

[37] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia, "The Totem Multiple-ring Ordering and Topology Maintenance Protocol," *ACM*, May 1998.

[38] P. Unterbrunner, G. Alonso, and D. Kossmann, "E-Cast: Elastic Multicast," ETH Zurich, Department of Computer Science, Tech. Rep., 2011.

[39] G. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys (CSUR)*, vol. 33, no. 4, pp. 427–469, 2001.

[40] A. Schiper and S. Toueg, "From set membership to group membership: A separation of concerns," *Dependable and Secure Computing, IEEE Transactions on*, vol. 3, no. 1, pp. 2–12, 2006.

[41] V. Gramoli, L. Bass, A. Fekete, and D. Sun, "Rollup: Non-Disruptive Rolling Upgrade with Fast Consensus-Based Dynamic Reconfigurations," 2016.

[42] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin *et al.*, "All about Eve: Execute-Verify Replication for Multi-Core Servers." in *OSDI*, vol. 12, 2012, pp. 237–250.

[43] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *Journal of the ACM*, vol. 58, no. 2, p. 7, 2011.