Multi-Ring Paxos

Parisa Jalili Marandi
University of Lugano (USI)
Switzerland

Marco Primi
University of Lugano (USI)
Switzerland

Fernando Pedone
University of Lugano (USI)
Switzerland

***Abstract*—This paper addresses the scalability of group communication protocols. Scalability has become an issue of prime importance as data centers become commonplace. By scalability we mean the ability to increase the throughput of a group communication protocol, measured in number of requests ordered per time unit, by adding resources (i.e., nodes). We claim that existing group communication protocols do not scale in this respect and introduce Multi-Ring Paxos, a protocol that orchestrates multiple instances of Ring Paxos in order to scale to a large number of nodes. In addition to presenting Multi-Ring Paxos, we describe a prototype of the system we have implemented and a detailed evaluation of its performance.**

I. Introduction

State-machine replication is a fundamental approach to building fault-tolerant services. By replicating a server in multiple nodes, clients will observe a highly available service despite the failure of one or more replicas. State-machine replication dictates how client requests must be executed against the replicated servers such that the behavior of the system is indistinguishable from the behavior of a single-server setup. In brief, state-machine replication requires requests to be executed by every replica (agreement) in the same order (total order) [1], [2]. These two requirements are encapsulated by group communication protocols, such as atomic broadcast or total order broadcast [3]. Encapsulating agreement and total order in group communication primitives has been proved quite convenient as it provides a separation of concerns: application designers can focus on the service to be replicated and system designers can focus on developing efficient group communication protocols.

Years of research on the design and implementation of group communication have resulted in efficient protocols (e.g., [4], [5], [6], [7]). Existing systems, however, offer poor scalability, if any. By scalability we mean the ability of a group communication system to increase throughput, measured in number of messages ordered per time unit, when resources (i.e., nodes) are added. Unfortunately, existing group communication systems do not behave like this: In general, increasing the number of nodes that participate in an atomic broadcast protocol improves its tolerance to failures, but not its throughput. We observe that the maximum throughput of an atomic broadcast protocol is typically determined by the capacity of the individual nodes that participate in the protocol (i.e., limited by each node's resources such as CPU and disk), not by the aggregated capacity of the nodes.

We illustrate our observation with Ring Paxos [5], a highly efficient atomic broadcast protocol. Ring Paxos is based on Paxos [8]. It orders messages (e.g., containing service requests issued by clients) by executing a sequence of consensus instances. In each consensus instance several messages can be ordered at once. The durability of a consensus instance is configurable: If a majority of *acceptors*, the nodes that accept a consensus decision, is always operational, then consensus decisions can be stored in the main memory of acceptors only (hereafter, "In-memory Ring Paxos"). Without such an assumption, consensus decisions must be written on the acceptors' disks (hereafter, "Recoverable Ring Paxos"). The maximum throughput of In-memory Ring Paxos is determined by what the CPU or the network interface of an acceptor can handle, whichever becomes a bottleneck first. In Recoverable Ring Paxos, the maximum throughput is limited by the bandwidth sustained by an acceptor's disks.

Figure 1 shows the performance of In-memory Ring Paxos and the performance of Recoverable Ring Paxos (see Section VI-A for setup description). In-memory Ring Paxos is CPU-bound: throughput can be increased until approximately 700 Mbps, when the *coordinator*, an acceptor with a distinguished role, reaches its maximum processing capacity. When this happens, even small increases in the coordinator load result in large increases in delivery latency. Recoverable Ring Paxos is bounded by the bandwidth of the acceptors' disks. At maximum throughput, around 400 Mbps, the acceptors approach the maximum number of consensus instances they can store on disk per time unit. Notice that at this point the coordinator has moderate processing load, around 60%. In either case, adding resources (i.e., acceptors) will not improve performance.

If executing requests is more costly than ordering them, then throughput will be dictated by the number of requests a server can execute per time unit, not by the number of requests that Ring Paxos can order. In such cases, one solution is to partition the service into sub-services (e.g., [9], [10]), each one replicated using state-machine replication. Requests involving a single partition are submitted to and executed by the involved partition only; requests

This work was supported in part by the Swiss National Science Foundation under grant number 200020_134553.

978-1-4673-1625-5/12/$31.00 ©2012 IEEE

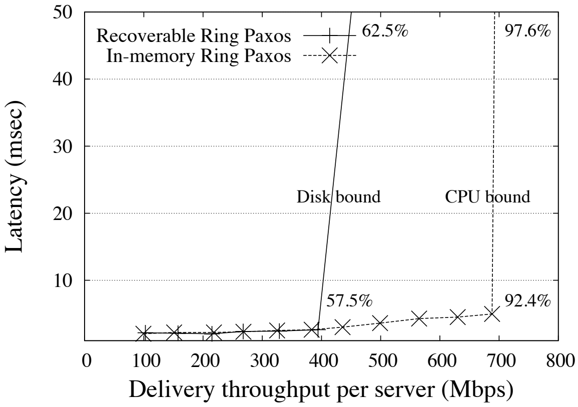

Figure 1. Performance of In-memory and Recoverable Ring Paxos.

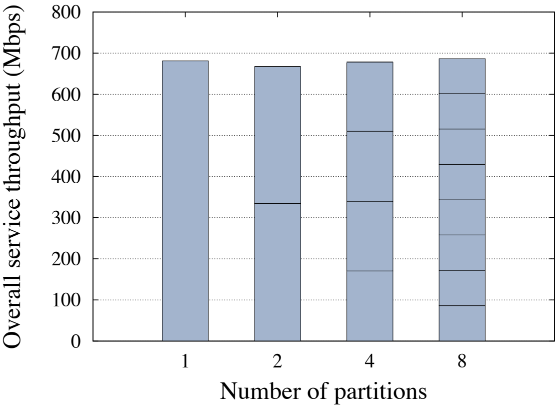

Figure 2. Performance of a partitioned service using In-memory Ring Paxos.

involving multiple partitions must be consistently ordered across partitions and executed by all involved partitions. As a result, if most requests affect a single partition (or few partitions), the scheme improves performance as the various partitions can execute requests in parallel. Ring Paxos can be configured to work with partitioned services by ordering all messages and selectively delivering them to the concerned partitions only [9]. By partitioning a service, the cost of executing requests can be distributed among partitions. But if a service can be partitioned into a large number of sub-services, then throughput may be limited by the overall number of requests that Ring Paxos can order and deliver to the various partitions, not by the capacity of the servers to execute the requests.

Figure 2 illustrates this case with a multi-partition service implemented using In-memory Ring Paxos. To emphasize our point, we assess the overall system throughput of a dummy service: delivered messages are simply discarded by the servers, that is, requests take no time to be executed. In this experiment, all submitted requests were single-partition and evenly distributed among partitions. The graph shows that the throughput of Ring Paxos does not increase as partitions and nodes (three per partition) are added. Instead, since the total throughput sustained by Ring Paxos is approximately the same for the various configurations, the more partitions a configuration has, the less throughput can be allocated to each partition.

In this paper we present Multi-Ring Paxos, a protocol that addresses the scalability of group communication protocols. Multi-Ring Paxos's key insight is to compose an unbounded number of parallel instances of Ring Paxos in order to scale throughput with the number of nodes. While the idea behind Multi-Ring Paxos is conceptually simple, its realization entailed non-obvious engineering decisions, which we discuss in the paper. Multi-Ring Paxos implements atomic multicast, a group communication abstraction whereby senders can atomically multicast messages to groups of receivers; atomic multicast ensures ordered message delivery for receivers that deliver messages in common. In brief, Multi-Ring Paxos assigns one instance of Ring Paxos to each group (or set of groups). Receivers that subscribe to a single group will have their messages ordered by the Ring Paxos instance responsible for this group. Receivers that subscribe to multiple groups will have multiple sources of messages and use a deterministic merge mechanism to ensure ordered delivery. Most of the complexity of Multi-Ring Paxos lies in its deterministic merge procedure, which accounts for dynamic load and imbalances among the various instances of Ring Paxos, without sacrificing performance or fault tolerance.

We have implemented a prototype of Multi-Ring Paxos using an open-source version of Ring Paxos [11] and conducted a series of experiments with our prototype. The results are encouraging: By composing eight instances of In-memory Ring Paxos, for example, we can reach an aggregated throughput of more than 5 Gbps, eight times the throughput of a single Ring Paxos instance. Recoverable Ring Paxos has similar scalability, linear in the number of Ring Paxos instances. In summary, Multi-Ring Paxos overcomes the limitations of existing group communication systems by composing independent instances of protocols, which individually do not scale. Although we have developed Multi-Ring Paxos using Ring Paxos, some of our ideas could be exploited with other atomic broadcast protocols.

The remainder of this paper is structured as follows. Section II presents the system model and some definitions used in the paper. Section III reviews Paxos and Ring Paxos. Section IV introduces Multi-Ring Paxos. Section V reviews related works and Section VI discusses the performance of Multi-Ring Paxos. Section VII concludes the paper.

II. System model and definitions

In this section we define our system model, present some group communication abstractions used throughout the paper, and illustrate the use of atomic multicast.

A. Processes and communication

We assume a distributed system composed of a set $\Pi = \{p_1, p_2, ...\}$ of interconnected processes. Processes may fail by crashing, but do not experience arbitrary behavior (i.e., no Byzantine failures). The network is mostly reliable and subject to small latencies, although load imbalances (e.g., peak demand) imposed on both nodes and the network may cause variations in processing and transmission delays. Communication can be one-to-one, through the primitives *send*(p, m) and *receive*(m), and one-to-many, through the primitives *ip-multicast*(g, m) and *ip-deliver*(m), where m is a message, p is a process, and g is the group of processes m is addressed to. Messages can be lost but not corrupted.

Our protocols ensure safety under both asynchronous and synchronous execution periods. The FLP impossibility result [12] states that under asynchronous assumptions consensus cannot be both safe and live. We thus assume that the system is *partially synchronous* [13], that is, it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [13], and it is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown. After GST nodes are either *correct* or *faulty*. A correct process is operational "forever" and can reliably exchange messages with other correct processes. This assumption is only needed to prove liveness properties about the system. In practice, "forever" means long enough for one instance of consensus to terminate.

B. Consensus, atomic broadcast and atomic multicast

Ring Paxos implements atomic broadcast as a sequence of consensus executions. Multi-Ring Paxos implements atomic multicast, a generalization of atomic broadcast. In the following we define these problems.

Consensus is defined by the primitives *propose*(v) and *decide*(v), where v is an arbitrary value. Consensus guarantees that (i) if a process decides v then some process proposed v *(uniform integrity)*; (ii) no two processes decide different values *(uniform agreement)*; and (iii) if one or more correct processes propose a value then eventually some value is decided by all correct processes *(termination)*.

Atomic broadcast is defined by the primitives *broadcast*(m) and *deliver*(m), where m is a message. Atomic broadcast guarantees that (i) if a process delivers m, then all correct processes deliver m *(uniform agreement)*; (ii) no two processes deliver any two messages in different orders *(uniform total order)*; and (iii) if a correct process broadcasts m, then all correct processes deliver m *(validity)*.

Atomic multicast implements the abstraction of groups $\Gamma = \{g_1, ..., g_\gamma\}$, where for each $g \in \Gamma$, $g \subseteq \Pi$. Consequently, processes may belong to one or more groups. If process $p \in g$, we say that p *subscribes* to group g.

Atomic multicast is defined by the primitives *multicast*(g, m) and *deliver*(m), and ensures that (i) if a process delivers m, then all correct processes in g deliver m *(uniform agreement)*; (ii) if processes p and q deliver messages m and m', then they deliver them in the same order *(uniform partial order)*; and (iii) if a correct process multicasts m to g, then all correct processes in g deliver m *(validity)*. If Γ is a singleton, then atomic multicast is equivalent to atomic broadcast.

C. A scalable service based on atomic multicast

To illustrate the implications and consequences of atomic broadcast and atomic multicast on the design of a replicated system, consider a simple database service accessed through requests insert(k), delete(k) and query(k_{min}, k_{max}), where the insert and delete requests involve a single key k and the query request returns every key k in the database such that $k_{min} \leq k \leq k_{max}$ [9].

If the service is replicated by means of state-machine replication, then each replica contains a full copy of the database and client requests can be propagated to the replicas using atomic broadcast. Consequently, every replica delivers and executes each request. A request is completed once the client receives the response from one of the replicas. The throughput of the service will be determined by the number of requests per time unit that can be either (a) ordered and delivered by the atomic broadcast primitive or (b) executed by the replicas, whichever bound is reached first.

Alternatively, one can divide the database into partitions $P_0, P_1, ..., P_l$ and replicate each partition using state-machine replication. Partition P_i is responsible for a range of database keys from the key space [14]. A request to insert or delete key k is multicast to the partition where k belongs; a query(k_{min}, k_{max}) request is multicast either to partition P_i, if k_{min} and k_{max} fall within P_i's range, or to all partitions otherwise. To map this scheme to atomic multicast, we can have one group g_i per partition P_i and a group g_{all} for all partitions. Each replica p in P_i belongs to g_i and g_{all}. Thus, except for queries that address all partitions, every replica delivers and executes requests that concern its partition only—a replica that delivers a query whose range does not fall within its partition simply discards it.

The throughput of each partition is limited by the requests per unit of time that (a) atomic multicast can order and deliver to the partition, and (b) the partition can execute. If requests access single partitions, then one can expect a system with n partitions to provide n times the throughput of a single-partition system, that is, a scalable system. In reality, as shown in Figure 2, this only happens if the group communication primitive itself scales: the number of messages per time unit ordered and delivered by the primitive grows with the size of the system. Multi-Ring Paxos is an atomic multicast primitive with this property.

III. Paxos and Ring Paxos

Ring Paxos is a variation of Paxos [8], optimized for clustered systems. In the following we briefly describe Paxos and Ring Paxos.

A. Paxos in a nutshell

Paxos distinguishes three roles: *proposers*, *acceptors*, and *learners*. A node can execute one or more roles simultaneously. In a client-server setup, clients act as proposers and servers as learners. A value is a command proposed by a client to be executed by the servers; the decided value is the next command to be executed. Each instance of Paxos proceeds in two phases: During Phase 1, the coordinator, a node among the acceptors, selects a unique round number *c-rnd* and asks a quorum Q_a (i.e., any majority) of acceptors to promise for it. By promising, an acceptor declares that, for that instance, it will reject any request (Phase 1 or 2) with round number less than *c-rnd*. Phase 1 is completed when Q_a confirms the promise to the coordinator. Notice that Phase 1 is independent of the value, therefore it can be pre-executed by the coordinator. If any acceptor already accepted a value for the current instance, it will return this value to the coordinator, together with the round number received when the value was accepted (*v-rnd*).

Once a coordinator completes Phase 1 successfully, it can proceed to Phase 2. Phase 2 messages contain a value and the coordinator must select it with the following rule: if no acceptor in Q_a accepted a value, the coordinator can select any value (i.e., the next client-submitted value). If however any of the acceptors returned a value in Phase 1, the coordinator is forced to execute Phase 2 with the value that has the highest round number *v-rnd* associated to it. In Phase 2 the coordinator sends a message containing a round number (the same used in Phase 1). When receiving such a request, the acceptors acknowledge it, unless they have already acknowledged another message (Phase 1 or 2) with a higher round number. Acceptors update their *c-rnd* and *v-rnd* variables with the round number in the message. When a quorum of acceptors accepts the same round number (Phase 2 acknowledgement), consensus terminates: the value is permanently bound to the instance, and nothing will change this decision. Thus, learners can deliver the value. Learners learn this decision either by monitoring the acceptors or by receiving a decision message from the coordinator.

As long as a nonfaulty coordinator is eventually selected, there is a majority quorum of nonfaulty acceptors, and at least one nonfaulty proposer, every consensus instance will eventually decide on a value. A failed coordinator is detected by the other nodes, which select a new coordinator. If the coordinator does not receive a response to its Phase 1 message it can re-send it, possibly with a bigger round number. The same is true for Phase 2, although if the coordinator wants to execute Phase 2 with a higher round number, it has to complete Phase 1 with that round number. Eventually the coordinator will receive a response or will suspect the failure of an acceptor.

B. Ring Paxos in a nutshell

Ring Paxos [5] differs from Paxos in a few aspects that make it more throughput efficient. The steps mentioned next refer to Figure 3, where Paxos's Phase 1 has been omitted.

- Acceptors are organized in a logical ring. The coordinator is one of the acceptors. Phase 1 and 2 messages are forwarded along the ring (Steps 3 and 4). Upon receiving a Phase 2 message, each acceptor appends its decision to the message so that the coordinator, at the end of the ring, can know the outcome (Step 5).
- Ring Paxos executes consensus on value IDs: for each client value, a unique identification number is selected by the coordinator. Consensus is executed on IDs which are usually significantly smaller than the real values.
- The coordinator makes use of ip-multicast. It triggers Phase 2 by multicasting a packet containing the client value, the associated ID, the round number and the instance number to all acceptors and learners (Step 3).
- The first acceptor in the ring creates a small message containing the round number, the ID and its own decision and forwards it along the logical ring.
- An additional acceptor check is required to guarantee safety. To accept a Phase 2 message, the acceptor must know the client value associated with the ID contained in the packet.
- Once consensus is reached, the coordinator can inform all the learners by just confirming that some value ID has been chosen. The learner will deliver the corresponding client value in the appropriate instance (Step 6). This information can be piggybacked on the next ip-multicast message.

Message losses may cause learners to receive the value proposed without the notification that it was accepted, the notification without the value, or none of them. Learners recover lost messages by inquiring other nodes. Ring Paxos assigns each learner to a preferential acceptor in the ring, to which the learner can ask lost messages. Lost Phase 1 and 2 messages are handled like in Paxos. The failure of a node (acceptor or coordinator) requires a new ring to be laid out.

IV. Multi-Ring Paxos

In this section, we present Multi-Ring Paxos. We also discuss failure handling, reconfigurations, extensions and optimizations. We argue for Multi-Ring Paxos correctness in the appendix.

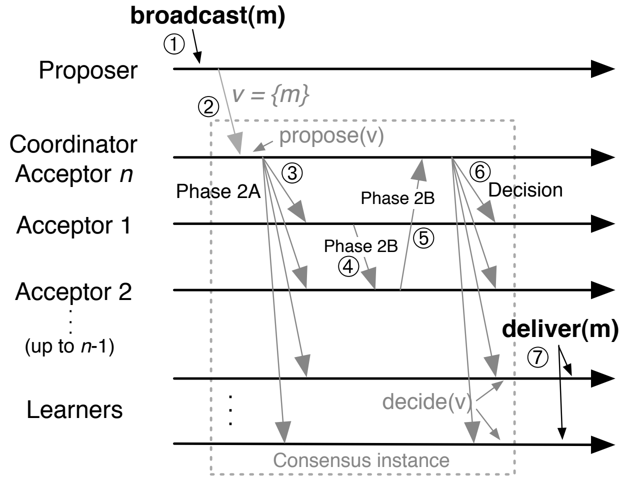

Figure 3. Ring Paxos: broadcast and delivery of message m (in black) and one consensus instance deciding on a single message (in gray).

A. Overview

Multi-Ring Paxos uses multiple independent instances of Ring Paxos to scale throughput without sacrificing response time—hereafter, we refer to a Ring Paxos instance as a "ring" and assume the existence of one ring per group (we revisit this issue in Section IV-D).

Learners subscribe to the groups they want to deliver messages from. Within a group, messages are ordered by the ring responsible for the group. If a learner subscribes to multiple groups, it uses a deterministic procedure to merge messages coming from different rings. Although deterministically merging messages from multiple rings is conceptually simple, its implementation has important performance consequences, as we now explain.

In general, learners implement the deterministic merge in round-robin fashion, delivering a fixed number of messages from each group they subscribe to in a pre-defined order. More precisely, each group has a unique identifier, totally ordered with any other group identifier. If a learner subscribes to groups $g_{l_1}, g_{l_2}, ..., g_{l_k}$, where $l_1 < l_2 < ... < l_k$, then the learner could first deliver M messages from g_{l_1}, then M messages from g_{l_2}, and so on, where M is a parameter of the algorithm. In order to guarantee ordered delivery, the learner may have to buffer messages that do not arrive in the expected pre-defined order.

This scheme has two drawbacks, which we illustrate with an example. Assume that a learner subscribes to groups g_1 and g_2, which generate messages at rates λ_1 and λ_2, respectively, where $\lambda_1 < \lambda_2$. First, the learner's delivery rate will be $2\lambda_1$, as opposed to the ideal $\lambda_1 + \lambda_2$. Second, the learner's buffer will grow at rate $\lambda_2 - \lambda_1$ and will eventually overflow.

One way to address the problems above is to define a value of M per group that accounts for different rates: If for each M_1 messages delivered for g_1, the learner delivers $M_2 = M_1\lambda_2/\lambda_1$ messages for g_2, then its total delivery rate will tend to the ideal. In the general case of a learner that subscribes to groups $g_{l_1}, g_{l_2}, ..., g_{l_k}$, it follows that $M_{l_1}/\lambda_{l_1} = M_{l_2}/\lambda_{l_2} = ... = M_{l_k}/\lambda_{l_k}$ must hold in order for the learner to deliver messages at the ideal rate of $\lambda_{l_1} + \lambda_{l_2} + ... + \lambda_{l_k}$.

Such a mechanism, however, requires estimating the message rate of each group and dynamically adapting this estimate during the execution. Moreover, to avoid buffer overflows, learners have to quickly adapt to changes in the message rate of a group. Our strategy does not require adapting to a group's message rate. Instead, we define λ, the *maximum expected message rate* of any group, a parameter of the system. The coordinator of each ring monitors the rate at which messages are generated in its group, denoted μ, and periodically compares λ to μ. If μ is lower than λ, the coordinator proposes enough "skip messages" to reach λ. Skip messages waste minimum bandwidth: they are small and many can be batched in a single consensus instance.

Figure 4 illustrates an execution of Multi-Ring Paxos with two groups. Learner 1 subscribes to group g_1; learner 2 subscribes to groups g_1 and g_2. Notice that after receiving message m_4 learner 2 cannot deliver it since it must first deliver one message from group g_2 to ensure order—in the execution, $M = 1$. Therefore, learner 2 buffers m_4. Since learner 1 only subscribes to g_1, it can deliver all messages it receives from Ring Paxos 1 as soon as it receives them. At some point, the coordinator of Ring Paxos 2 realizes its rate is below the expected rate and proposes to skip a message. As a consequence, learner 2 can deliver message m_4.

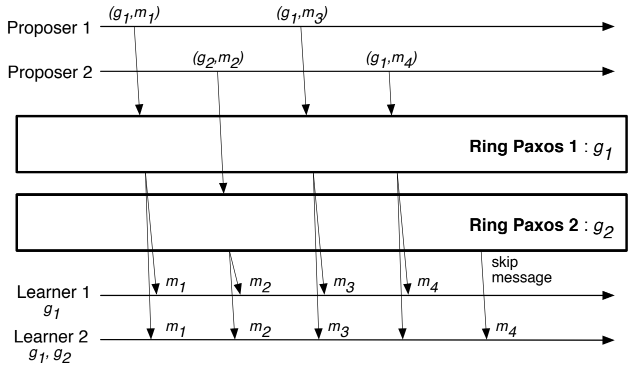

Figure 4. Muli-Ring Paxos execution with two rings and $M = 1$.

B. Algorithm in detail

Algorithm 1 details Multi-Ring Paxos. To multicast message m to group g, a proposer sends m to the coordinator of g (lines 3–4), which upon receiving m, proposes m in consensus instance k (lines 11–12). The acceptors execute consensus instances as in Ring Paxos (line 22; see also Figure 3). For simplicity, in Algorithm 1 one message is proposed per consensus instance. In our prototype, multiple messages are batched and proposed in a single instance.[1] Since consensus instances decide on batches of fixed size, if we set λ to be the maximum expected consensus rate, as opposed to the maximum expected message rate, we can easily determine λ since we know the maximum throughout of Ring Paxos.

```
 1: **Algorithm 1: Multi-Ring Paxos** (executed by process $p$)
 2: *Task 1 (proposer)*
 3: To multicast message $m$ to group $g$:
 4:     send $m$ to coordinator of $g$
 5: *Task 2 (coordinator)*
 6: *Initialization:*
 7:     $k \leftarrow 0$
 8:     $prev\_k \leftarrow 0$
 9:     set timer to expire at current time + $\Delta$
10: upon receiving $m$ from proposer
11:     propose$(k, m)$
12:     $k \leftarrow k + 1$
13: upon timer expires
14:     $\mu \leftarrow (k - prev\_k)/\Delta$
15:     if $\mu < \lambda$ then
16:         $skip \leftarrow prev\_k + \Delta\lambda$
17:         for $k \leftarrow k$ to $skip$ do
18:             propose$(k, \perp)$
19:     $prev\_k \leftarrow k$
20:     set timer to expire at current time + $\Delta$
21: *Task 3 (acceptor)*
22: execute consensus (Phases 1 and 2 of Ring Paxos)
23: *Task 4 (learner)*
24: *Initialization:*
25:     for $i \leftarrow 1$ to $\gamma$ do
26:         if $p \in g_i$ then $k_i \leftarrow 0$
27: repeat forever
28:     for $i \leftarrow 1$ to $\gamma$ do
29:         if $p \in g_i$ then
30:             repeat $M$ times
31:                 wait for decide$(k_i, v)$
32:                 if $v \neq \perp$ then deliver $v$
33:                 $k_i \leftarrow k_i + 1$
34: *Algorithm variables:*
35:     $k$ : current consensus instances in a group (coordinator)
36:     $prev\_k$ : value of $k$ at the beginning of an interval
37:     $\mu$ : number of consensus instances per time in a group
38:     $skip$ : consensus instances below optimum in last interval
39:     $k_i$ : the next consensus instance at group $g_i$ (learner)
40: *Algorithm parameters:*
41:     $\gamma$ : number of groups
42:     $\Delta$ : duration of an interval (i.e., time between samplings)
43:     $M$ : number of consecutive messages delivered for a group
44:     $\lambda$: expected number of consensus instances per $\Delta$
```

The coordinator sets a local timer (lines 9 and 20), which expires in intervals of Δ time units. In each interval, the coordinator computes μ, the number of consensus instances proposed in the interval (line 14). If μ is smaller than λ (line 15), the coordinator proposes enough *skip instances*, i.e., empty instances, to make up for the missing ones (lines 16–18). Notice that although in Algorithm 1 the coordinator executes a propose for each missing instance, in our prototype this is implemented much more efficiently by proposing a batch of instances using the same physical messages. The coordinator then sets the timer for the next interval (line 20).

Each learner keeps in variable k_i the number of the next consensus instance in which it will participate, for each group g_i to which the learner subscribes (lines 25–26). The procedure at the learner consists in deterministically delivering M messages (lines 30–32) multicast to each group g_i subscribed by the learner (lines 28–29). Since groups are totally ordered according to their unique identifiers, each two learners will round robin through the groups they subscribe to in the same order, and hence respect multicast order.

C. Failures and reconfigurations

Algorithm 1 assumes that rings guarantee progress individually. Therefore, for each ring, up to $f < n/2$ acceptors can fail, where n is the total number of acceptors in a Ring Paxos instance. To reduce response time, Ring Paxos keeps $f+1$ acceptors in the ring only [5]; the remaining acceptors are spares and could be shared by multiple rings in Multi-Ring Paxos, similarly to Cheap Paxos [15].

When an acceptor is suspected to have failed, its ring must be reconfigured, excluding the suspected acceptor and including a new one, from the spares. Until the ring is reconfigured, learners that subscribe to this ring cannot deliver messages broadcast to this ring and to any other ring the learner also subscribes. We assess the effects of reconfiguration in Section VI-F. Recovering from lost messages is done with retransmissions, as in Ring Paxos [5].

D. Extensions and optimizations

Algorithm 1 can be optimized for performance in a number of ways. As described in the previous sections, the coordinator does not propose a single message in a consensus instance, but a batch of messages. Moreover, multiple skip instances for an interval are executed together. Thus, the cost of executing any number of skip instances is the same as the cost of executing a single skip instance.

Another issue concerns the mapping of groups to rings (i.e., instances of Ring Paxos). If there are as many rings as groups, then we can have one group per ring—this is the setting used in our experiments. Alternatively, multiple groups can be mapped to the same ring. The drawback of such a setting is that some learners may receive messages from groups they do not subscribe to. Such messages will not be delivered to the application, but they waste the learner's incoming bandwidth and processor.

[1] A consensus instance is triggered when a batch is full or a timeout occurs. We use batches of 8 kB as this results in high throughput (see Marandi et al. [5] for more details).

While there are many strategies to address this issue (e.g., a simple one is to assign the busiest groups to different rings), we note that mapping γ groups to δ rings, where $\gamma > \delta$, is an optimization problem with implications that go beyond the scope of this paper [16].

V. Related work

Multi-Ring Paxos is an atomic multicast protocol. Differently from atomic broadcast, atomic multicast protocols can be made to scale under certain workloads. In the following we focus the discussion mostly on atomic multicast and review a few atomic broadcast protocols that share some similarities with Multi-Ring Paxos.

Although the literature on atomic broadcast protocols is vast [17], few atomic multicast algorithms have been proposed. Possibly, the first atomic multicast algorithm is due to D. Skeen, an algorithm for failure-free scenarios [18]. In Skeen's algorithm, the destination processes of a message m exchange timestamps and eventually decide on m's final timestamp. The destinations deliver messages according to the message's final timestamp. The algorithm scales under certain workloads since only the destinations of a message are involved in its ordering.

Several papers have proposed extensions to render Skeen's original algorithm fault tolerant [19], [20], [21], [22]. The basic idea behind these algorithms is to replace failure-prone processes by fault-tolerant groups of processes; each group implementing the logic in Skeen's algorithm by means of state-machine replication. Different algorithms have proposed different optimizations of this basic idea, all of which based on the assumption that groups do not intersect. An algorithm that departures from the previous proposals appears in [23]. The idea is to daisy-chain the set of destination groups of a message according to the unique group ids. The first group runs consensus to decide on the delivery of the message and then hands it over to the next group, and so on. Thus, the latency of a message depends on the number of destination groups.

Most previous work on atomic multicast had a theoretical focus. One notable exception is the Spread toolkit [6]. Spread is a highly configurable group communication system, which supports the abstraction of process groups. It relies on interconnected daemons, essentially the components that handle the physical communication in the system, to order messages. Participants connect to a daemon to multicast and deliver messages. The abstraction of groups in Spread, however, was not created for performance, but to easy application design. In Section VI we experimentally compare Multi-Ring Paxos and Spread.

Mencius is a protocol that implements state-machine replication in a wide-area network [24]. Mencius is a multi-leader protocol derived from Paxos. The idea is to partition the sequence of consensus instances among the leaders to amortize the load and better balance the bandwidth available at the leaders. Similarly to Multi-Ring Paxos, leaders can account for load imbalances by proposing skip instances of consensus. Differently from Multi-Ring Paxos, Mencius does not implement the abstraction of groups; it is essentially an atomic broadcast protocol.

Multi-Ring Paxos's deterministic merge is conceptually similar to the work proposed in [25], which aims to totally order message streams in a widely distributed publish-subscribe system. Differently from Multi-Ring Paxos merge scheme, the mechanism proposed in [25] uses approximately synchronized clocks to estimate the expected message rates of all publishers and then merge messages throughout the network in the same way.

VI. Performance evaluation

In this section we evaluate Multi-Ring Paxos experimentally. In the first set of experiments, reported in Section VI-B, we assess the effect of two important configurations on the performance of Multi-Ring Paxos. To this end the values of the parameters involved in the design of the protocol are kept constant. Then the effects of λ, Δ, and M, the three main parameters, are evaluated in Sections VI-C, VI-D, and VI-E while keeping the configuration of Multi-Ring Paxos constant. In Section VI-F, we investigate the effect of coordinator failures on the performance of the system.

A. Experimental setup

We ran the experiments in a cluster of Dell SC1435 servers equipped with 2 dual-core AMD-Opteron 2.0 GHz CPUs and 4 GB of main memory. The servers are interconnected through an HP ProCurve2900-48G Gigabit switch (0.1 msec of round-trip time). In all the experiments, unless specified otherwise, λ, Δ, and M were set to 9000 consensus instances per interval, 1 millisecond, and 1 message, respectively. The size of application-level messages was 8 kB in all the experiments.

In all experiments each group has a dedicated ring. Recoverable Multi-Ring Paxos uses buffered disk writes. Thus, in the experiments both In-memory and Recoverable Multi-Ring Paxos assume that a majority of acceptors is operational during each consensus instance. To remove peaks in latency due to flushes to disk, we report the average latency after discarding the 5% highest values. When analyzing throughput, we report the aggregated throughput of the system, which combines the throughput of each group. Our prototype uses an open-source version of Ring Paxos [11].

B. Performance of Multi-Ring Paxos

Depending on the number of learners and groups, there can be many configurations of Multi-Ring Paxos. Two extreme cases are when (1) each learner subscribes to only one group and (2) each learner subscribes to all groups. The first case assesses the scalability of Multi-Ring Paxos since throughput is not limited by the incoming bandwidth of a

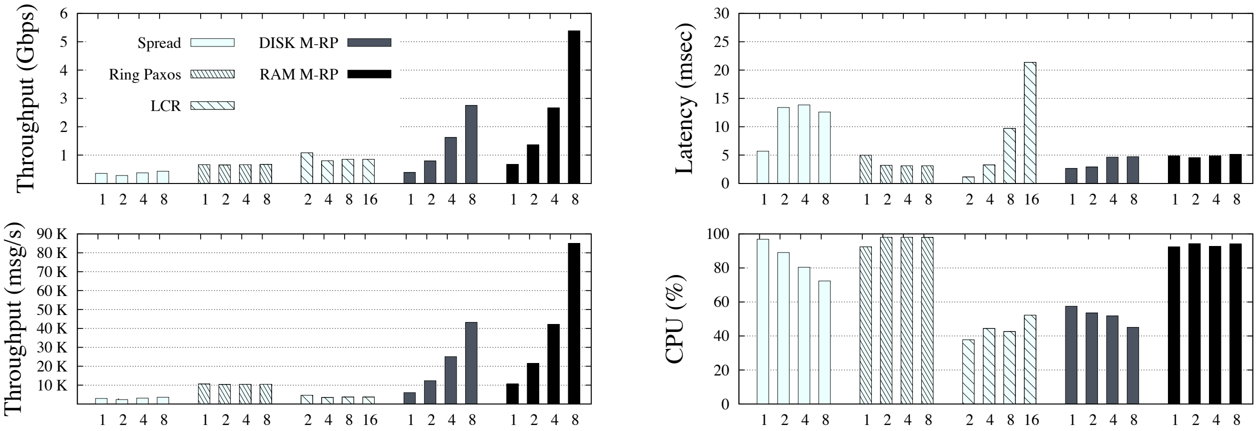

Figure 5. Performance of In-memory Multi-Ring Paxos (RAM M-RP) and Recoverable Multi-Ring Paxos (DISK M-RP), compared with Spread, Ring Paxos and LCR. The x-axis shows number of partitions for RAM M-RP, DISK M-RP and Ring Paxos; number of daemons/groups for Spread; and number of nodes in the ring of LCR. There are 2 acceptors per partition in RAM M-RP and DISK M-RP, and a fixed number of 2 acceptors in Ring Paxos. The CPU of the most-loaded node is reported, which for RAM M-RP, DISK M-RP and Ring Paxos is the coordinator. In Spread messages are 16 kB long and in LCR they are 32 kB long.

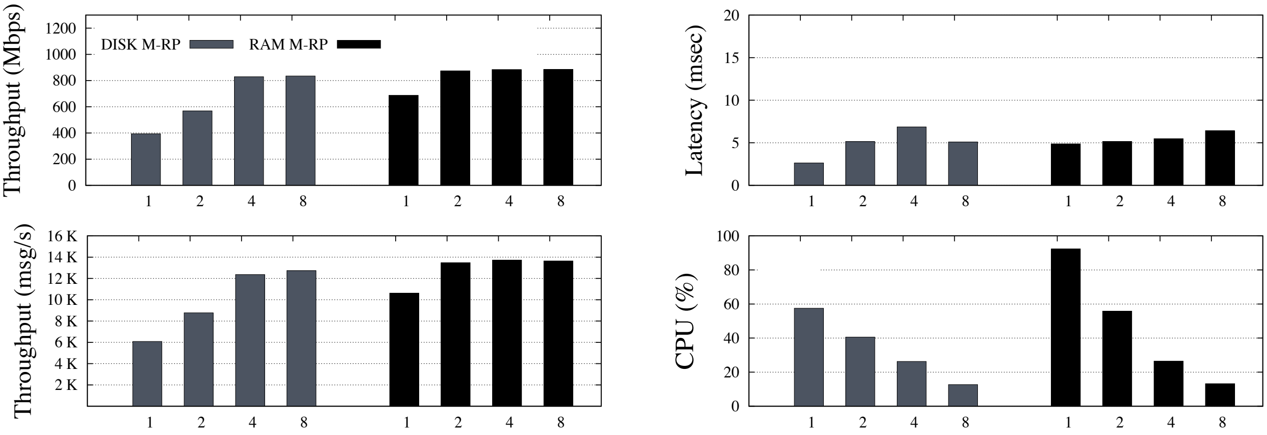

Figure 6. Performance of Multi-Ring Paxos when each learner subscribes to all groups.

learner. The second case assesses the ability of learners to combine messages from multiple rings.

When each learner subscribes to only one group (see Figure 5), the throughput of the learner is bound by the maximum throughput of the ring in charge of the learner's group. This is because before the learner uses up its local resources, the coordinator of each ring in In-memory Multi-Ring Paxos saturates its CPU and the acceptors in Recoverable Multi-Ring Paxos reach their maximum disk bandwidth (see also Figure 1). The throughput of both Multi-Ring Paxos protocols grows linearly with the number of partitions, peaking at more than 5 Gbps with In-memory Multi-Ring Paxos and about 3 Gbps with Recoverable Multi-Ring Paxos. As a reference, we also present the performance of Spread, Ring Paxos, and LCR. The first two systems implement the abstraction of groups but do not scale with the number of groups. LCR is a high performance atomic broadcast protocol and does not implement groups.

Figure 6 shows the performance of Multi-Ring Paxos when learners subscribe to all groups. For both Multi-Ring Paxos protocols, with one ring the bottleneck is the single Ring Paxos instance. As groups (i.e., rings) are added, the aggregate throughput of the various rings eventually saturates the learners' incoming links. To reach the maximum capacity of a learner, In-memory Multi-Ring Paxos needs two rings and Recoverable Multi-Ring Paxos needs three rings. This experiment illustrates how Multi-Ring Paxos can combine multiple "slow" atomic broadcast protocols (e.g., due to disk writes) to build a much faster protocol.

C. *The effect of Δ on Multi-Ring Paxos*

Recalling from Section IV-B, Δ is the interval in which the coordinator of a ring samples the number of executed consensus instances to then check for the need of skip

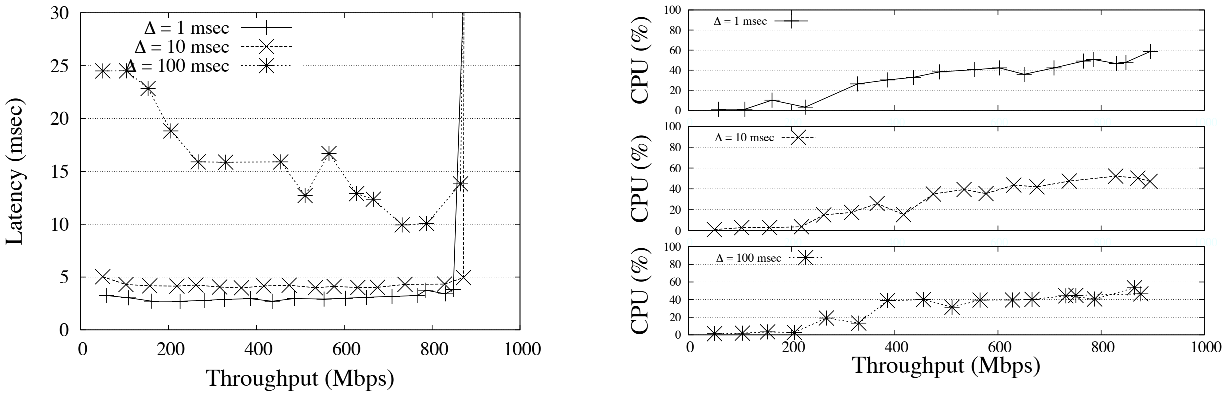

Figure 7. The effect of Δ on Multi-Ring Paxos. Latency versus throughput (left) and CPU at the coordinator of one of the rings (right).

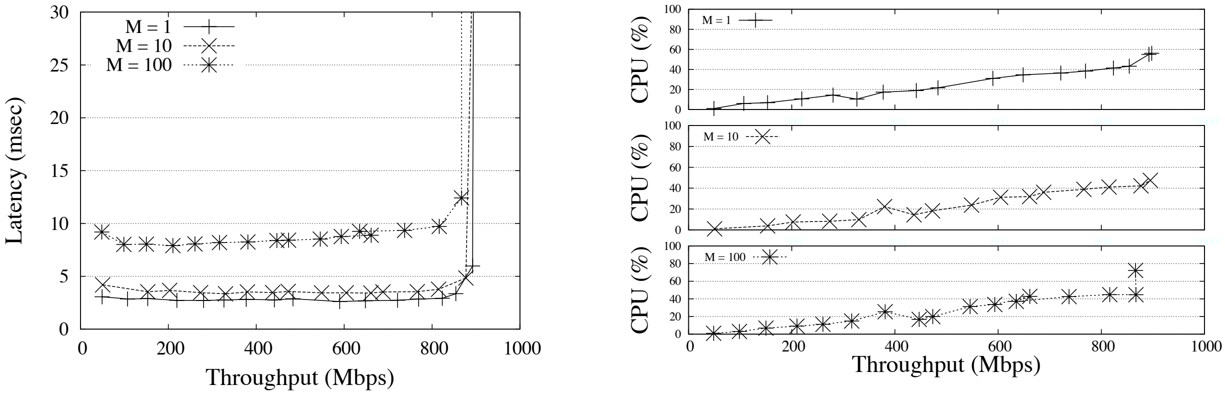

Figure 8. The effect of M on Multi-Ring Paxos. Latency versus throughput (left) and corresponding CPU usage in the learner (right).

instances. The value assigned to Δ should be big enough to avoid unnecessary samplings and checks, and small enough to allow quick corrections in the rate of the ring. To investigate the effects of Δ, we deployed an In-memory Multi-Ring Paxos configuration with two rings and one learner that subscribed to both rings. Messages were created at the same rate in both rings and this rate did not change over time.

From the graph on the left of Figure 7, a large Δ results in high latency at the learners, suggesting that small values are preferred. Notice that even though each ring has the same rate, small variations in the transmission and handling of messages can lead to the buffering of messages at the learners and increased latency. For large values of Δ (e.g., 100 milliseconds), latency decreases with the throughput since fewer skip instances are needed. The graphs on the right of Figure 7 assess the processing cost of Δ. First, the maximum throughput is not affected by Δ, as all configurations reach the approximately same maximum. Second, small values of Δ have no effect on the CPU usage of the coordinator. Therefore, choosing small values of Δ is justifiable.

D. The effect of M on Multi-Ring Paxos

We evaluate next the effect of M in the execution. We recall that M is the number of consensus instances that a learner handles at a time from each ring it expects messages from. In these experiments, we have deployed an In-memory Multi-Ring Paxos with two rings, and one learner that receives messages from both of them. As the left graph of Figure 8 implies, by augmenting the value of M, the average latency increases. The reason is that while M instances of a ring are handled in the learner, instances of other rings are buffered and delayed. As M increases, this delay increases and so does the average latency. As it is evident in Figure 8 (right side), M has no effect on the throughput and CPU usage of the learner. Therefore, a small M is a good choice.

E. The effect of λ on Multi-Ring Paxos

If a learner subscribes to several groups, each with a different message rate, slow groups will delay the delivery of messages multicast to faster groups, and therefore negatively affect the latency and overall throughput observed

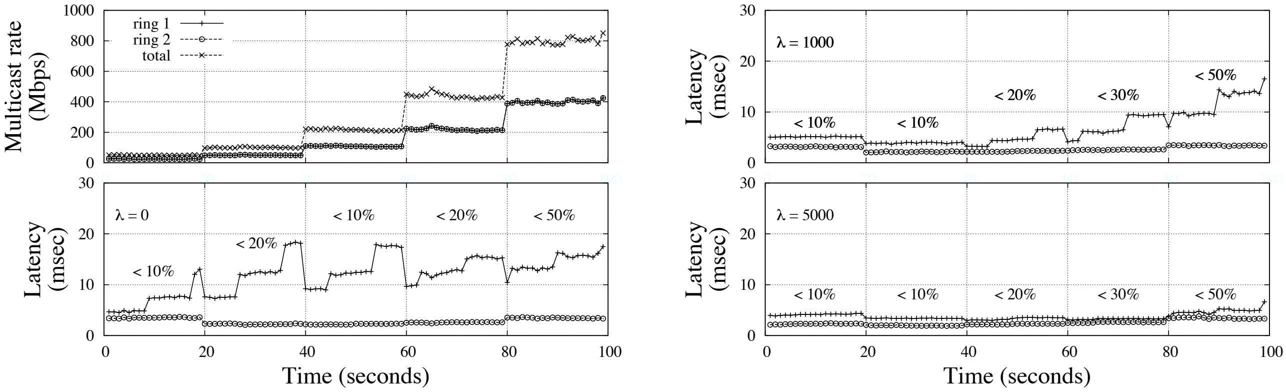

Figure 9. The effect of λ when the rates of the rings are constant and equal (percentages show load at ring coordinators).

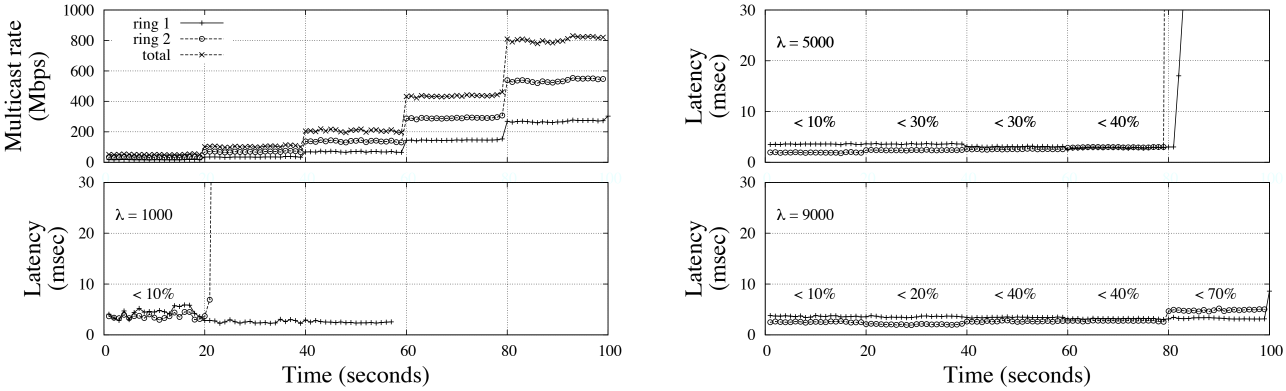

Figure 10. The effect of λ when the rates of the rings are constant and one is twice the other (percentages show load at ring coordinators).

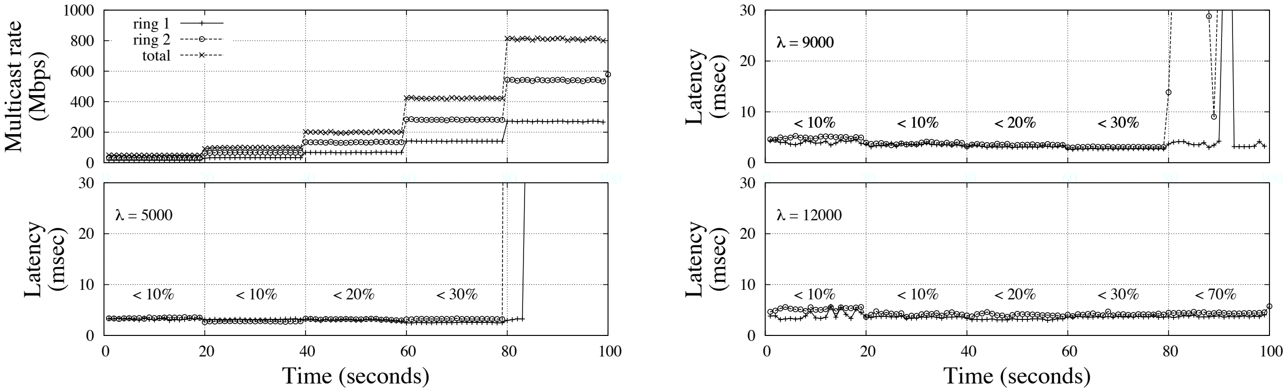

Figure 11. The effect of λ when the rates vary over time and in average one is twice the other (percentages show load at ring coordinators).

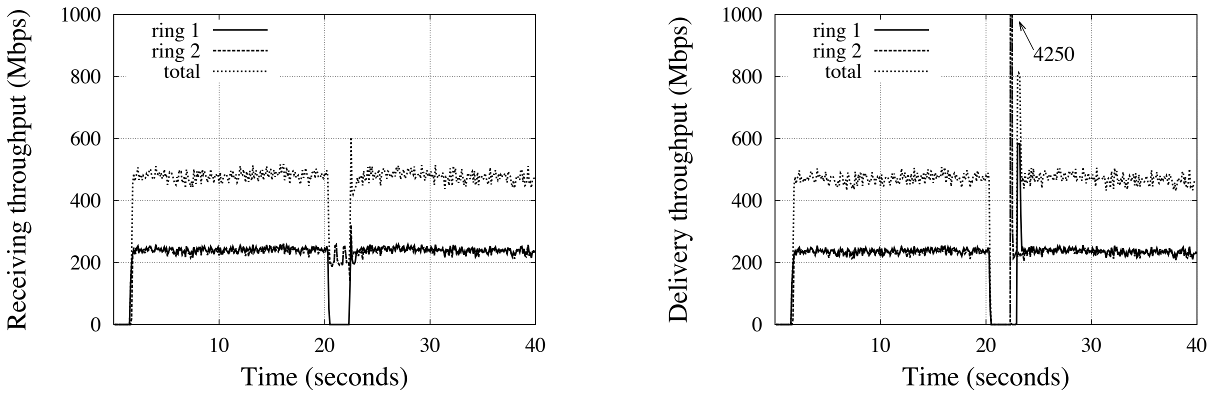

Figure 12. The effect of a coordinator failure in a learner of In-memory Multi-Ring Paxos.

by the learners. Multi-Ring Paxos copes with these issues by skipping consensus instances and by carefully setting λ, the maximum expected consensus rate of any group. In the following, we investigate the effect of λ on the system.

We have conducted three sets of experiments using In-memory Multi-Ring Paxos with two rings and one learner. In the first experiment (see Figure 9) proposers multicast messages to the two groups at a fixed and equal rate. In the second experiment (see Figure 10) the ratio of multicast messages to one of the groups is twice the other, though the multicast rate is constant in both groups throughout the execution. In the last experiment (see Figure 11), not only the ratio of multicast messages to one of the groups is twice the other, but their submission rates oscillates over the time such that the average is the same as in the previous experiment. In all the cases, we increase the multicast rate every 20 seconds. In all the figures, the top left graph shows the individual multicast rate per group and the total multicast rate in the system.

In Figure 9, we initially set λ to 0 (i.e., no mechanism to skip consensus instances). Even though the group rates are the same, even under low rates the traffic from the rings gets “out-of-sync” at the learner and messages have to be buffered, a phenomenon that the learner does not recover from. With λ equal to 1000, latency remains stable with higher loads, but the problem still exists at very high load. With λ set to 5000 the problem is solved. Figure 10 illustrates the problem when the learner buffers overflow (i.e., $\lambda = 1000$ after 20 seconds and $\lambda = 5000$ after 80 seconds). A buffer overflow brings the learner to a halt since it cannot deliver buffered messages and new messages keep arriving. A large value of λ is enough to handle the most extreme loads in this experiment. Figure 11 shows a similar situation, which is only solved when λ is set to 12000. Skipping up to 12000 consensus instances in an interval of one second, where each instance decides on messages of 8 kB, corresponds to “skipping” up to 750 Mb of data per second, approximately the maximum throughput achieved by a ring. We recall that all such instances are skipped using a single consensus execution.

F. The effect of discontinued communication

We now investigate the effect of discontinued communication (e.g., due to a coordinator failure) in Multi-Ring Paxos. Our experiment consisted in deploying two rings and a learner that listened to these rings. Each ring generates messages with the same constant rate of approximately 4000 messages per second in average. In steady state, the learner receives and delivers approximately 500 Mbps of data (see Figure 12). After 20 seconds we stop the coordinator of ring 1, bringing the receiving throughout from this ring at the learner to zero. Although messages still arrive at the learner from ring 2, the learner buffers such messages as it cannot execute its deterministic merge procedure. The result is that the delivery throughput at the learner drops to zero (graph on the right of Figure 12). Notice that after ring 1 stops, the incoming throughput from ring 2 decreases, as the learner does not acknowledge the delivery of messages from group 2 to the node that multicasts to ring 2 and this one slows down its sending rate.

Three seconds later the execution at ring 1 proceeds. We forced a restart after three seconds to emphasize the effects of the discontinuity of traffic. In reality, it takes much less time to detect the failure of a coordinator and replace it with an operational acceptor. When the coordinator of the first ring starts, it notices that no consensus instances were decided in the last intervals and proposes to skip multiple consensus instances. As a result, the learner delivers all messages it has enqueued, momentarily leading to a high peak in the delivery throughput. Then the execution proceeds as normal.

Acknowledgements

We wish to thank Carlos Eduardo Bezerra, Daniele Sciascia, and the anonymous reviewers for their help and suggestions to improve the paper.

VII. Conclusions

This paper presented Multi-Ring Paxos, a protocol that combines multiple instances of Ring Paxos to implement atomic multicast. While atomic broadcast induces a total order on the delivery of messages, atomic multicast induces a partial order. Multi-Ring Paxos exploits the abstraction of groups in a different way than previous atomic multicast algorithms: In Multi-Ring Paxos, messages are addressed to a single group only, but processes can subscribe to multiple groups. In all atomic multicast algorithms we are aware of, messages can be multicast to multiple groups, and often groups cannot intersect. Finally, although Multi-Ring Paxos uses Ring Paxos as its ordering protocol within a group, one could use any atomic broadcast protocol within a group, a conjecture that we plan to investigate in the future.

References

[1] 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.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[3] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, 2nd ed. Addison-Wesley, 1993, ch. 5.

[4] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, pp. 5:1–5:32, July 2010.

[5] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 527 –536.

[6] 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.

[7] K. P. Birman and R. van Renesse, *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Press, 1994.

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

[9] P. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *International Conference on Dependable Systems and Networks (DSN)*, 2011.

[10] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proc. VLDB Endow.*, vol. 3, pp. 48–57, 2010.

[11] http://libpaxos.sourceforge.net.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty processor," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[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] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[15] L. Lamport and M. Massa, "Cheap Paxos," in *International Conference on Dependable Systems and Networks (DSN)*, 2004, pp. 307–314.

[16] M. Adler, Z. Ge, J. F. Kurose, D. F. Towsley, and S. Zabele, "Channelization problem in large scale data dissemination," in *Proceedings of the Ninth International Conference on Network Protocols*, 2001, pp. 100–109.

[17] 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.

[18] 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.

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

[20] N. Schiper and F. Pedone, "On the inherent cost of atomic broadcast and multicast in wide area networks," in *International conference on Distributed computing and networking (ICDCN)*, 2008, pp. 147–157.

[21] J. Fritzke, U., P. Ingels, A. Mostefaoui, and M. Raynal, "Fault-tolerant total order multicast to asynchronous groups," in *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, 1998, pp. 228–234.

[22] L. Rodrigues, R. Guerraoui, and A. Schiper, "Scalable atomic multicast," in *International Conference on Computer Communications and Networks*, 1998, pp. 840–847.

[23] C. Delporte-Gallet and H. Fauconnier, "Fault-tolerant genuine atomic multicast to multiple groups," in *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, 2000, pp. 107–122.

[24] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, 2008, pp. 369–384.

[25] M. K. Aguilera and R. E. Strom, "Efficient atomic broadcast using deterministic merge," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (PODC)*, 2000, pp. 209–218.