Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

Ridge: high-throughput, low-latency atomic multicast

Carlos Eduardo Bezerra[1,2], Daniel Cason[1,3], Fernando Pedone[1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brazil
[3] Instituto de Computação, Universidade Estadual de Campinas, Brazil

Abstract

It has been shown that the highest throughput for broadcasting messages in a point-to-point network is achieved with a ring topology. Although several ring-based group communication protocols have benefited from this observation, broadcasting messages along a ring overlay may lead to high latencies: In a system with n processes, at least $n-1$ communication steps are necessary for all processes to deliver a message. In this work, we argue that it is possible to reach optimal throughput without resorting to a ring topology (or to ip-multicast, typically unavailable in wide-area networks). This can be done by routing messages through different paths, while carefully using the available bandwidth at each process, resulting in a significantly lower latency for every message (potentially a single communication step). Based on this idea, we propose Ridge, a Paxos-based atomic multicast protocol where each message is initially forwarded to a single destination, the *distributor*, whose responsibility is to propagate the message to all other destinations. To utilize all bandwidth available in the system, processes alternate in the role of distributor. By doing this, the maximum system throughput matches that of ring-based protocols, with a latency that is not significantly dependent on the size of the system. Finally, we show that Ridge can also deliver messages optimistically, with even lower latency.

Report Info

Published
April 2015

Number
USI-INF-TR-2015/01

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Many modern online services have strict availability and scalability requirements. Highly available services remain operational despite node crashes and datacenter disasters. Scalable services can boost performance by adding system components and thereby accommodate increased load (e.g., as a result of new clients joining the system). However, designing highly available and scalable services is challenging. Some approaches have responded to this challenge by weakening the service guarantees offered to the clients, in what is generally known as *weak consistency* systems; some other approaches, known as *strong consistency* systems, have considered efficient infrastructures to provide high availability and scalability without compromising service guarantees.

Although weak consistency can lead to highly scalable and available systems, it is not applicable to all services and when it is applicable, it may penalize the service users, who must cope with non-intuitive service behavior. Strong consistency does not suffer from these shortcomings but requires ordering requests across the system. Ordering requests in a distributed system is a challenge that has received much attention from the scientific community [1]. While some works have focused on reducing the latency of message ordering protocols (e.g., [2–6]), several other proposals have aimed at increasing their throughput (e.g., [7–12]). Ridge, the protocol that we have designed and implemented, strives to achieve high throughput without sacrificing latency, a compromise that is often present in existing high throughput total order protocols.

In [8], it has been shown that the maximum throughput in a point-to-point network may be achieved by disposing processes in a ring overlay: To broadcast a message m, each process sends m to the next process in the ring. The authors consider a system composed of n processes running in rounds. Each process can send one message per round to any number of processes while it can receive only one message per round (communication can be done simultaneously in both directions). Each process can also receive in a round a message that the process sends to itself. As a consequence, the maximum throughput in messages per round that can be achieved with a ring overlay is $n/(n-1)$, if there are n senders, or 1, otherwise.

The prospects of achieving throughput optimality with a ring overlay have motivated the design of several protocols such as LCR [8], Ring Paxos [10], Multi-Ring Paxos [12, 13], and Spread [14] (some of which coupled with ip-multicast communication). All these protocols achieve very high throughput, but are subject to an inherent limitation of ring overlays: the latency to deliver messages is proportional to the number of processes in the system. In this paper, we set out to investigate whether more latency efficient message dissemination techniques can promise the same throughput optimality of a ring overlay. We show here that it is possible to reach optimal throughput without resorting to a ring topology or to ip-multicast, which is usually unavailable in wide-area networks. Based on this result, we introduce Ridge, a Paxos-based atomic multicast protocol where each message is initially forwarded to a single destination, the *distributor*, whose responsibility is to propagate the message to all other destinations. To use all bandwidth available in the system, processes alternate in the role of distributor. As a result, Ridge's maximum throughput matches the throughput of ring-based protocols, with a latency that does not significantly depend on the system size. We also show that the ideas that motivate Ridge can be combined with other optimizations to reduce latency, such as optimistic delivery of messages.

The paper makes the following contributions: (a) we introduce Ridge, a high-throughput, latency-efficient atomic multicast protocol; (b) we reason about Ridge's theoretical maximum performance; (c) we provide a detailed experimental evaluation of Ridge's performance and compare it to the performance of other ordering protocols.

The remainder of the paper is structured as follows. Section 2 describes the system model. Section 3 recalls the Paxos protocol. Sections 4 and 5 describe Ridge and analyze its performance analytically. Section 6 assesses Ridge's performance experimentally. Section 7 surveys related work and Section 8 concludes the paper.

2 System model and definitions

We consider a distributed system composed of a set $\mathscr{P} = \{p_0, p_1, ...\}$ of processes that communicate through message passing and do not have access to a shared memory or a global clock. Processes are either *correct*, if they never fail, or *faulty*, otherwise. Processes, however, do not experience arbitrary behavior (i.e., no Byzantine failures). Processes communicate by message passing, using primitives $send(p, m)$ and $receive(m)$, where m is a message and p is the process m is addressed to. If sender and receiver are correct, then every message sent is eventually received.

To implement atomic multicast, we make use of *consensus*. 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; (ii) no two processes decide different values; and (iii) if one (or more) correct process proposes a value then eventually some value is decided by all correct processes.

Since it is impossible to solve consensus in an asynchronous system [15], we assume that the system is *partially synchronous* [16]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [16], 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 to the processes.

3 Background

Paxos [17] is a fault-tolerant consensus algorithm and the basis for Ridge. We describe next how a value is decided in a single consensus instance. Paxos distinguishes three roles: *proposers*, *acceptors*, and *learners*. Each process can play one or more of these roles simultaneously. Proposers propose values, acceptors choose values, and learners learn what value was decided. Hereafter, A denotes the set of acceptors, L the set of learners, and Q a *majority quorum* of acceptors, that is, a subset of A of size $\lceil(|A|+1)/2\rceil$.

The execution of one consensus instance proceeds in a sequence of *rounds*, each identified by a number. For each round, one process (typically one of the proposers) acts as *coordinator* of the round. To propose a value, proposers send the value to the coordinator. The coordinator maintains two variables: (a) *c-rnd* is the highest-numbered round that the coordinator has started; and (b) *c-val* is the value that the coordinator has picked for round *c-rnd*. Acceptors maintain three variables: (a) *rnd* is the highest-numbered round in which the acceptor has participated; (b) *v-rnd* is the highest-numbered round in which the acceptor has cast a vote—it follows that $rnd \leq v\text{-}rnd$ always holds; and (c) *v-val* is the value voted by the acceptor in round *v-rnd*.

Algorithm 1 Paxos

```
Initialization:
    $c\text{-}rnd \leftarrow 0; rnd \leftarrow 0; v\text{-}rnd \leftarrow 0$
    $c\text{-}val \leftarrow null; v\text{-}val \leftarrow null$
Task 1 (coordinator)
when receiving value $v$ from proposer
    $c\text{-}rnd \leftarrow$ unique value higher than current $c\text{-}rnd$
    for all $acc \in A$ do send ($acc$, ⟨PHASE 1A, $c\text{-}rnd$⟩)
Task 2 (acceptor)
when receiving ⟨PHASE 1A, $c\text{-}rnd$⟩ from coordinator
    if $c\text{-}rnd > rnd$ then
        $rnd \leftarrow c\text{-}rnd$
        send (coordinator, ⟨PHASE 1B, $rnd, v\text{-}rnd, v\text{-}val$⟩)
Task 3 (coordinator)
when receiving ⟨PHASE 1B, $rnd, v\text{-}rnd, v\text{-}val$⟩ from $Q$, such that $rnd = c\text{-}rnd$
    $h \leftarrow$ highest $v\text{-}rnd$ value received
    $V \leftarrow$ set of ⟨$v\text{-}rnd, v\text{-}val$⟩ received with $v\text{-}rnd = h$
    if $h = 0$ then $c\text{-}val \leftarrow v$
    else $c\text{-}val \leftarrow$ the only $v\text{-}val$ in $V$
    for all $acc \in A$ do send ($acc$, ⟨PHASE 2A, $c\text{-}rnd, c\text{-}val$⟩)
Task 4 (acceptor)
when receiving ⟨PHASE 2A, $c\text{-}rnd, c\text{-}val$⟩ from coordinator
    if $c\text{-}rnd \geq rnd$ then
        $v\text{-}rnd \leftarrow c\text{-}rnd$
        $v\text{-}val \leftarrow c\text{-}val$
        send (coordinator, ⟨PHASE 2B, $v\text{-}rnd, v\text{-}val$⟩)
Task 5 (coordinator)
when receiving ⟨PHASE 2B, $v\text{-}rnd, v\text{-}val$⟩ from $Q$
    if for all received messages: $v\text{-}rnd = c\text{-}rnd$ then
        for all $learner \in L$ do send ($learner$, ⟨DECISION, $v\text{-}val$⟩)
```

Algorithm 1 provides an overview of Paxos. The algorithm has two phases. To execute Phase 1, the coordinator picks a round number *c-rnd* greater than any value it has picked so far, and sends it to the acceptors (Task 1). Upon receiving such a message (Task 2), an acceptor checks whether the round number received from the coordinator is greater than any round number it has received so far; if that is the case, the acceptor “promises" not to accept any future message with a round smaller than *c-rnd*. The acceptor then replies to the coordinator with the highest-numbered round in which it has cast a vote, if any, and the value it voted for. Notice that no value is proposed to the acceptors in Phase 1.

The coordinator starts Phase 2 after receiving a reply from a quorum (Task 3). Before proposing a value in Phase 2, the coordinator checks whether some acceptor has already cast a vote in a previous round. If not, the coordinator can use the value received from the proposer. If one or more acceptors have cast votes in previous rounds, the coordinator picks the value that was voted for in the highest-numbered round.

An acceptor will vote for a value *c-val* with corresponding round *c-rnd* in Phase 2 if the acceptor has not received any Phase 1 message for a higher-numbered round (Task 4). Voting for a value means setting the acceptor’s variables *v-rnd* and *v-val* to the values sent by the coordinator. If the acceptor votes for the value received, it replies to the coordinator. When the coordinator receives replies from a quorum (Task 5), it knows that a value has been decided and notifies the learners.

Algorithm 1 can be optimized in a number of ways [17]. For instance, the coordinator can execute Phase 1 before values are received from proposers. This way, decisions can be faster: when a proposer sends a value to the coordinator, the coordinator can immediately start Phase 2, as Phase 1 has already been executed. In the next section, we describe Ridge in detail, including its optimization to Paxos Phase 2.

4 Ridge

Ridge is an atomic multicast protocol that can deliver messages to groups of processes. In this section, we define the properties ensured by Ridge (Section 4.1), describe its general idea (Section 4.2), detail its operation (Section 4.3), explain how it tolerates failures (Section 4.4), and how it implements optimistic delivery (Section 4.5). Correctness is discussed in Appendix A.

4.1 Problem definition

To multicast a message m to a set of groups of processes γ, primitive multicast(γ, m) is used. Ridge performs up to two deliveries for each multicast message m: deliver(m) (atomic delivery) and opt-deliver(m) (optimistic delivery).

The **atomic delivery** ensures the following properties:

(i) If a correct process multicasts m, then every correct process in γ delivers m *(validity)*.

(ii) For any message m, every process p in γ delivers m at most once, and only if some process has multicast m previously *(integrity)*.

(iii) If a process delivers m, then every correct process in γ delivers m *(uniform agreement)*.

(iv) No two processes p and q deliver m and m' in different orders, and the delivery order is acyclic *(atomic order)*.

The **optimistic delivery** guarantees validity, integrity and the following properties:

(v) If a correct process opt-delivers m, then every correct process in γ opt-delivers m *(agreement)*.

(iv) If the given optimistic assumptions hold, no two processes p and q deliver m and m' in different orders, and the delivery order is acyclic *(optimistic order)*.

4.2 Overview of the protocol

Ridge makes use of Paxos, executing Phase 1 similarly to Algorithm 1, while optimizing Phase 2 for high throughput and low latency. Ridge utilizes a collection of *ensembles*, where an ensemble is a set of processes capable of executing a sequence of Paxos instances. Each ensemble contains $2f+1$ acceptors, where f is the maximum number of failures tolerated by the ensemble. For each run of Paxos, Ridge defines a majority quorum Q with $f+1$ acceptors. To decide on message m in an ensemble, the ensemble's coordinator (which has previously run Phase 1 of Paxos for multiple consensus instances) starts Phase 2 by sending m to an acceptor, which forwards it to another acceptor, and so on, until $f+1$ acceptors have received the message. Unless something abnormal occurs (e.g., a failure or multiple coordinators proposing messages in the same Paxos instance), the last acceptor (i.e., the $f+1$-th acceptor to receive m) will know that a majority of acceptors have received m, configuring a quorum. Then, the last acceptor sends m to a learner. Such a learner, called the *distributing learner* (or distributor) for m, then sends m to all other learners directly, completing Phase 2 of Paxos. Learners take turns as the distributing learners for different messages. By doing this, Ridge achieves high throughput, assuming that each learner distributes the same amount of data. To ensure that data is equally divided among learners, a load balancing procedure is used: the last acceptor keeps track of how much data each learner has distributed so far and chooses the distributing learner for the next message accordingly.

To implement atomic multicast, as described in Section 4.1, Ridge allows learners to receive messages from different ensembles; hereafter, we denote messages decided in an ensemble a *message stream*. If a learner subscribes to multiple ensembles, a deterministic function is used to merge the corresponding message streams, as we explain next.

To merge message streams, Ridge employs Paxos instance ids to ensure a gap-free sequence of decisions from each ensemble. It also employs timestamps to determine the position of each message in the resulting merged sequence. For this to be possible, each ensemble must ensure that the timestamp order of decided messages follow the messages' decision order (for messages of the same ensemble). More formally, let m be a message with timestamp $m.ts$ decided in consensus instance k by ensemble e, and let m' be a message with timestamp $m'.ts$ decided in instance k' also by ensemble e. We need that, if $k < k'$, then $m.ts < m'.ts$. When a message m is created, an *initial timestamp* is assigned to it by its sender. When m is decided and received by a learner, it is only delivered when the learner knows that there will be no message m' with a lower timestamp than that of m to be received from any ensemble the learner subscribes to. To ensure that the timestamp order and the instance order agree for each ensemble, each learner follows a deterministic procedure: upon delivery, if messages m and m' are decided in the same ensemble, $k < k'$, but $m.ts > m'.ts$, then the timestamp of m' is adjusted to be higher than that of m, being its *final timestamp*. Once the learner has determined the final timestamp of a message, it proceeds with deterministic merge. Every timestamp is assumed to be unique across the system and the final message sequence follows the timestamp order of the merged message streams.

The way Ridge merges multiple message sequences raises a liveness concern: what if a learner never receives a message from one of the ensembles it subscribes to (i.e., because no process multicasts a message to this ensemble)? To ensure liveness, Ridge uses *null messages*: if no message is multicast to an ensemble for a predefined time Δ, the coordinator of the ensemble generates a null message (i.e., a message with no payload), with the sole purpose of preventing learners from blocking while waiting for ensembles with low or no traffic.

Finally, in Ridge processes can propagate a message to a single ensemble only, which is at odds with the definition of atomic multicast, where a message can be multicast to multiple groups. To implement the abstraction of groups, Ridge maps groups to ensembles, as we describe next. Suppose a system in which messages can be multicast to any combination of groups $g_1, ..., g_n$. This abstraction can be implemented in Ridge with $n+1$ ensembles, $e_1, ..., e_n, e_{all}$. For every group g_i that contains process p, p becomes a learner of ensembles e_i and e_{all}. If m is multicast to a single group g_i, it is propagated to e_i; if m is multicast to multiple groups, it is sent to e_{all}, with non-addressee processes simply discarding m—that is, the ensemble's learners that have not subscribed to any of the groups that m was multicast to will just disregard m.

4.3 Detailed algorithms

We detail here the algorithms executed during normal operation, i.e., when there are no failures or failure suspicions; we discuss abnormal cases in Section 4.4. Algorithm 2 shows how Ridge optimizes Paxos Phase 2 for both throughput and latency (unchanged code from original Paxos is grayed out). For Algorithm 3, which details Ridge's deterministic merge procedure, we assume that each ensemble uses consensus as a black box, with primitives $propose_e(k, v)$ and $decide_e(k, v)$ to respectively propose and decide a value v for consensus instance k of ensemble e. Finally, Algorithm 4 shows a simple way of mapping Paxos ensembles to multicast groups, allowing messages to be multicast to multiple process groups.

To solve consensus in Ridge, we assume that each ensemble eventually has a single and correct coordinator [17]. To execute Phase 1 (Task 1), the coordinator defines a quorum Q for the proposed round of Paxos containing a majority of the acceptors in A, arranged in a sequence $a_1; a_2; \ldots; a_{(f+1)}$. When starting Phase 2 of Paxos (Task 2), the coordinator sends message $\langle$PHASE 2, *c-rnd*, *c-val*, Q, 0$\rangle$ to acceptor a_1. This tuple means that it is a message concerning Phase 2 of Paxos, for round *c-rnd*, proposing value *c-val*, the acceptor sequence in Q is the quorum, and no votes have been cast by the acceptors yet.

Upon receiving a message $\langle$PHASE 2, *c-rnd*, *c-val*, Q, *count*$\rangle$ (Task 4), acceptor a_i knows that *count* votes have been cast so far to decide value *c-val* in consensus. If the number of votes, plus a_i's own vote, is still not enough to reach a quorum, a_i increments the vote count and forwards the message to the next acceptor in Q, $a_{(i+1)}$. When the $|Q|$ votes necessary to decide *c-val* have been cast, the acceptor that completed the quorum will choose a distributing learner and ask for it to distribute the value decided to the other learners (Task 5). We want to divide equally the amount of data distributed by each of the learners. For simplicity, Algorithm 2 assumes that all learners are chosen uniformly by the last acceptor to be the distributors and that all messages have the same size. Ridge's actual implementation uses a load balancer at each acceptor to help choose distributing learners.

Algorithm 2 Ridge: executing Paxos in one ensemble

```
1:  Initialization:
2:      $c\text{-}rnd \leftarrow 0$; $rnd \leftarrow 0$; $v\text{-}rnd \leftarrow 0$
3:      $c\text{-}val \leftarrow null$; $v\text{-}val \leftarrow null$
4:  Task 1 (coordinator)
5:  when receiving value $v$ from proposer
6:      $c\text{-}rnd \leftarrow$ unique value higher than current $c\text{-}rnd$
7:      // define a quorum $Q$ as an acceptor sequence for round $c\text{-}rnd$
8:      $Q \leftarrow \{a_1, a_2, \ldots, a_{(f+1)}\}$ // majority of the $2f+1$ acceptors
9:      for all $acc \in A$ do send ($acc$, ⟨PHASE 1A, $c\text{-}rnd$⟩)
10: Task 2 (acceptor)
11: when receiving ⟨PHASE 1A, $c\text{-}rnd$⟩ from coordinator
12:     if $c\text{-}rnd > rnd$ then
13:         $rnd \leftarrow c\text{-}rnd$
14:         send (coordinator, ⟨PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$⟩)
15: Task 3 (coordinator)
16: when receiving ⟨PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$⟩ from $Q$, such that $rnd = c\text{-}rnd$
17:     $h \leftarrow$ highest $v\text{-}rnd$ value received
18:     $V \leftarrow$ set of ⟨$v\text{-}rnd$, $v\text{-}val$⟩ received with $v\text{-}rnd = h$
19:     if $h = 0$ then $c\text{-}val \leftarrow v$
20:     else $c\text{-}val \leftarrow$ the only $v\text{-}val$ in $V$
21:     send (acceptor $a_1$, ⟨PHASE 2, $c\text{-}rnd$, $c\text{-}val$, $Q$, 0⟩)
22: Task 4 (acceptor $a_i$)
23: when receiving ⟨PHASE 2, $c\text{-}rnd$, $c\text{-}val$, $Q$, $count$⟩
24:     if $c\text{-}rnd \geq rnd$ then
25:         $v\text{-}rnd \leftarrow c\text{-}rnd$
26:         $v\text{-}val \leftarrow c\text{-}val$
27:         if $count + 1 < |Q|$ then
28:             send ($a_{(i+1)}$, ⟨PHASE 2, $v\text{-}rnd$, $v\text{-}val$, $Q$, $count+1$⟩)
29:         else
30:             for all $l \in L$ do send ($l$, ⟨DECISION, $id(v\text{-}val)$⟩)
31:             $l_d \leftarrow$ a learner in $L \setminus Q$
32:             send ($l_d$, ⟨DISTRIBUTE, $v\text{-}val$⟩)
33: Task 5 (learner)
34: when receiving ⟨DISTRIBUTE, $v\text{-}val$⟩
35:     for all $l \in L \setminus Q$ do send ($l$, ⟨VALUE, $v\text{-}val$⟩)
```

It is possible that some of the learners are also acceptors in the majority quorum Q. An acceptor from Q should not be a distributing learner, as such an acceptor already receives and forwards proposed values (in Task 4). Adding the burden of distributing decided values would likely overload the acceptor and defeat the purpose of Ridge, which is to maximize throughput. For this reason, the distributing learner is chosen from $L \setminus Q$. For learners in Q, which in normal cases have already received the value decided, to be notified about decisions, two separate messages are sent: VALUE, which contains the decided value *v-val*, and DECISION, which contains only *id(v-val)*, a unique identifier of *v-val*. The last acceptor of Q sends DECISION to all learners, while the distributing learner sends VALUE only to learners that haven't received *v-val* yet. By doing this, and assuming that the size of *id(v-val)* is negligible, Ridge's throughput efficiency is maintained even when processes are both learners and acceptors.

Ridge implements atomic multicast with the help of an intermediate layer, called *ensemble-multicast*, which allows processes do deterministically merge messages decided in multiple ensembles. If we consider the set of learners of each ensemble as an *ensemble-group*, the properties of ensemble-multicast (i.e., *ensemble-delivery*) are the ones of atomic multicast, described in Section 4.1, except that each message can be ensemble-multicast to only one single ensemble-group.

Algorithm 3 shows how ensemble-multicast operates. To multicast a message m to an ensemble-group of learners of ensemble e, the proposer p first assigns a unique timestamp $m.ts$ to m, which is based on p's system clock. Then, p sends m to e's coordinator (lines 2–4). Upon receiving m, the coordinator proceeds with proposing m in the next consensus instance k (lines 10 and 11). When a learner becomes aware that m

is the next decided message in ensemble e (line 30), it checks if m has a timestamp higher than $last\_ts_e$ (the timestamp of the last message from e); if not, $m.ts$ is adjusted to $last\_ts_e + 1$, ensuring that the timestamp and instance id orders agree in every ensemble. After checking this, the learner sets $last\_ts_e$ to $m.ts$ and puts $\langle e, m\rangle$ in a queue to be delivered when possible (lines 31–35). This queue contains a gap-free sequence of decisions from each ensemble; k_e is incremented each time a decision is received from an ensemble e, ensuring that the learner will be notified about decisions from e in the correct order (line 36).

When the *gap_free* queue contains at least one decision from every ensemble the learner subscribes to, the learner removes the message with lowest timestamp from *queue* and ensemble-delivers it (lines 37–41). This can be done because the learner processes consensus decisions from each ensemble in order; since final timestamps are adjusted to agree with the consensus decision order, any new decisions will contain a message with higher timestamp.

Algorithm 3 also shows how Ridge ensures liveness. At the initialization, and whenever a proposal is made, a timer is set to expire in Δ time units (lines 8, 13 and 19). This makes sure that, if a learner has a message m enqueued for delivery, it will eventually be able to deliver it, because it will receive messages with increasing timestamps from every ensemble at least every Δ (line 37).

Finally, Algorithm 4 shows how Ridge implements atomic multicast on top of ensemble-multicast. The ensemble-multicast protocol does not allow a message to be multicast to multiple ensembles. To allow each message to be multicast to multiple groups, we can have an ensemble e_γ for each combination γ of destination groups. If a message is multicast to γ, then it is implemented as an ensemble-multicast to e_γ. To avoid requiring 2^n ensembles to accommodate all possible combinations of up to n groups, we decided to have n+1 ensembles: if a message m is multicast only to g_i, then it is ensemble-multicast to e_i; if m is multicast to multiple groups, it is ensemble-multicast to e_{all} (lines 4–7). Using this simple mapping, m will be ensemble-delivered by all processes if it has been multicast to more than one group; for this reason, m receives a tag $m.dests$ that contains m's actual destination groups (line 3). Upon ensemble-delivery of m, each process p checks if m is actually addressed to p, by comparing the set of m's destination groups with the set σ of groups that p subscribed to; if that is the case, m is delivered to p.

4.4 Tolerating failures

Each Ridge ensemble is an implementation of Paxos, so leader election, failure detection and fault-tolerance is handled in the same way as the original protocol. What changes is the way messages are routed. Because of that, Ridge is sensitive not only to the failure of the coordinator, but also to the failure of the acceptors in Q and to the failure of distributing learners.

4.4.1 Coordinator failure

If the coordinator is suspected of failing, another process will take its place, by (eventually) running Phase 1 of Paxos with a round number higher than that used by the suspected coordinator [17].

4.4.2 Acceptor failure

In case the coordinator suspects that an acceptor of Q has failed during the execution of a consensus instance k, the coordinator falls back to original Paxos (Algorithm 1). It re-executes the consensus instance k, including Phase 1 and Phase 2 of original Paxos, with a higher round number.

4.4.3 Learner failure

If the last acceptor of Q suspects of a failure of the distributing learner, the acceptor will forward the message to a different learner. In any case, learner failures may cause other learners not to receive some consensus decisions. When suspecting that decisions were not received, learners can check with the acceptors what has been decided so far and fill any possible gaps.

4.5 Optimistic deliveries

Ridge allows messages to be delivered optimistically, with the optimistic order likely matching the atomic order. For this purpose, Ridge uses a scheme similar to that of [6], making the following assumptions: (a) there are no message losses, (b) messages from each ensemble are decided in the order of their initial

Algorithm 3 Ensemble-multicast with deterministic merging

```
Task 1 (proposer)
To ensemble-multicast message $m$ to ensemble $e$
    $m.ts \leftarrow$ unique timestamp based on local clock
    send $m$ to the coordinator of $e$
Task 2 (coordinator of ensemble e)
Initialization:
    $k \leftarrow 0$
    set timer to expire in $\Delta$
when receiving $m$ from a proposer
    propose$_e(k, m)$
    $k \leftarrow k+1$
    stop timer
    set timer to expire in $\Delta$
when timer expires
    create empty null message
    $null.ts \leftarrow$ unique timestamp based on local clock
    propose$_e(k, null)$
    $k \leftarrow k+1$
    set timer to expire in $\Delta$
Task 3 (acceptor)
execute consensus (follow Algorithm 2)
Task 4 (learner)
Initialization
    $\forall e : k_e \leftarrow 0$
    $\forall e : last\_ts_e \leftarrow 0$
    $all\_decisions \leftarrow \emptyset$
    $gap\_free \leftarrow \emptyset$
when decide$_e(k, m)$
    $all\_decisions \leftarrow all\_decisions \cup \{\langle e, k, m\rangle\}$
when $\exists e : \langle e, k_e, m\rangle \in all\_decisions$
    if $m.ts < last\_ts_e$ then
        $m.ts \leftarrow last\_ts_e + 1$
    $last\_ts_e \leftarrow m.ts$
    $all\_decisions \leftarrow all\_decisions \setminus \{\langle e, k_e, m\rangle\}$
    $gap\_free \leftarrow gap\_free \cup \{\langle e, m\rangle\}$
    $k_e \leftarrow k_e + 1$
when $\forall e : l$ receives from $e, \exists m : \langle e, m\rangle \in gap\_free$
    take $\langle e_d, m_d\rangle$, where $m_d.ts$ is the lowest in $gap\_free$
    $gap\_free \leftarrow gap\_free \setminus \{\langle e_d, m_d\rangle\}$
    if $m_d \neq null$ then
        ensemble-deliver$(m_d)$
```

timestamps (thus their final timestamps will be the same as their initial ones) and (c) when a learner delivers a message m optimistically, it has already received all messages that could possibly have a timestamp lower than that of m.

Condition (a) is approximated with a reliable communication protocol (e.g., TCP). We satisfy conditions (b) and (c) by having each process wait "long enough" before proposing a message or delivering a message optimistically. In the case of (b), the coordinator waits for a certain amount of time before proposing messages, allowing messages received out of order to be proposed in the correct (initial timestamp) order; as for (c), the wait time allows the learner to deliver optimistically in the correct order.

To perform optimistic deliveries, when a process p ensemble-multicasts a message m to an ensemble e, p first sends[1] m directly to all learners of e. After receiving m directly from p, each learner delivers m optimistically after waiting a certain time, which is based on the estimated latency and clock skews of

[1]To ensure that the value was received, the process multicasting m uses reliable multicast [18] to send the value to the learners and to the coordinator.

Algorithm 4 Ridge - multicasting messages to multiple groups

```
1:  Task 1 (multicaster)
2:  To multicast $m$ to set of groups $\gamma$
3:      $m.dests \leftarrow \gamma$
4:      if $|\gamma| = 1 \wedge \gamma = \{g_i\}$ then
5:          ensemble-multicast $m$ to $e_i$
6:      else
7:          ensemble-multicast $m$ to $e_{all}$
8:  Task 2 (destination)
9:  Initialization
10:     destination $p$ subscribes to set of groups $\sigma$
11:     for each $g_i \in \sigma$ do
12:         become a learner of ensemble $e_i$
13:     if $|\sigma| > 1$ then
14:         become a learner of ensemble $e_{all}$
15: when ensemble-deliver $m$
16:     if $m.dests \cap \sigma \neq \emptyset$ then
17:         deliver($m$)
```

the system. Besides sending m to the learners, p sends m also to the coordinator of e, which proceeds with Phase 2 of Paxos for m. Once the $(f+1)$-th acceptor is reached, it only notifies the learners about the decision, by sending *id(m)* to them. At this point, learners are expected to have already received m directly from p; if that is the case, they can deliver m atomically. If a learner still hasn't received m when receiving *id(m)*, the learner asks the acceptors for m. Note that the optimistic delivery does not guarantee uniformity: if both p and a learner l_f are faulty, it is possible that l_f delivers m optimistically, when no other process delivers m. However, this does not affect the correctness of the atomic delivery.

5 Theoretical analysis

In this section, we show that Ridge achieves high throughput in a point-to-point network. We first describe the performance model used here, then show that having a distributor for each different message can lead to optimal throughput, and then explain why it achieves low latency.

5.1 Performance model

Our performance model is based on the one used in [7] and [8]. In that model, each process can send one message per round and receive one message per round: if a single process p sends one message to a destination q, it will take a single round for q to receive the message. Likewise, if q has n messages to receive, it will finish receiving after n rounds. Although this is a very good model for throughput, it predicts latency as a function of throughput, which is not always realistic. For instance, in a 1 Gbps network with 50 μs average latency, a process can send up to 125 megabytes per second, or approximately 125 bytes per microsecond. If we consider 125-byte messages, the round length (time necessary to send one message) would be 1 μs. This model predicts that the message would take only 1 round (1 μs) to arrive at the destination. However, in reality it would take around 50 μs (or 50 rounds) in such a network.

We extend this performance model to decouple latency from throughput, defining the following. At each round k, every process can send only one message (we assume that the underlying network provides only one-to-one communication) and can also receive only one message. We define δ as the latency, in number of rounds, for a message to arrive from a process p at a process q, where $\delta > 0$. In more detail, at each round k, every process p_i can execute all or a subset of the following steps:

(1) compute a message for round k, $m(i,k)$,

(2) send $m(i,k)$ to one process,

(3) receive at most one message sent at round $k-\delta$.

5.2 Throughput

We consider a system with n processes: $p_0, p_1, ..., p_{(n-1)}$. To simplify the explanation, let $p_{(i \bmod n)}$ be p_i, so p_n represents the same process as p_0, $p_{(n+1)}$ refers to p_1, and so on. Say all processes are broadcasting a message at the same time. Each process p_i is broadcasting m_i to $p_0, ..., p_{(n-1)}$. At the beginning, process p_i already has its own message m_i, so at round 1, each p_i sends m_i to $p_{(i+1)}$; at round 2, p_i sends m_i to $p_{(i+2)}$, and so on. At the $(n-1)$-th round, p_i will send m_i to the last destination $p_{(i+n-1)}$. So, we have that, after $n-1$ rounds, the n processes will have each finished sending their messages to all n destinations, resulting in a throughput of $n/(n-1)$ messages per round. This shows that each process sending its own message directly to all other processes achieves the optimal throughput found in [8].

In the case of Ridge, we assume that Phase 1 of Paxos was pre-executed, so we focus on Phase 2. We consider that all n processes in the system are learners, so $L = \{p_0, p_1, ..., p_{(n-1)}\}$. For this analysis, we also assume that no failures happen nor are suspected to happen, so the majority quorum $Q = \{a_1, a_2, ..., a_{(f+1)}\}$ never changes, where a_1 is the coordinator. We show that, if the coordinator a_1 proposes one message per round, the throughput of the system is one message decided per round, which is the maximum rate of messages any process can receive from the network. With Ridge, Paxos Phase 2 is executed by having each acceptor a_i, where $1 \leq i \leq f$, receive one message m at each round and forward it to $a_{(i+1)}$, along with $id(m)$. Acceptor $a_{(f+1)}$ then sends m to l_d, which is a learner picked from $L \setminus Q$ at random (following a uniform distribution). Process l_d sends the message $|L \setminus Q|$ times, one for each of the remaining learners that have not received the message yet.[2] Each learner in $L \setminus Q$ is chosen to be a distributor $1/|L \setminus Q|$ times, so each learner sends one message per round on average. As every process receives one message and sends one message per round, we conclude that the throughput of the system is one message decided per round.

5.3 Latency

Note that, following our modified performance model, the theoretical analysis of maximum system throughput using a ring and using a different process as distributor for each message remain both as $n/(n-1)$ messages per round. The theoretical latency, though, is different. In the case of a ring, the first process after the sender p_i in the ring will receive m_i after δ rounds, while the last process in the ring will receive m_i after $\delta(n-1)$, which is the time necessary to complete the broadcast. When using distributors, each process p_i sends its message m_i to all others, one destination after the other, in $n-1$ rounds. The first destination, $p_{(i+1)}$ will receive m_i in δ rounds. Process $p_{(i+2)}$, in $\delta + 1$, and so on. The last destination, $p_{(i+n-1)}$, will receive m_i in $\delta + n - 2$ rounds, which is the time needed to finish the broadcast. (This means a single communication step if we consider only the network latency between processes, i.e., if we assume that the time necessary to send a message is negligible.)

To give an example, suppose a wide-area system with 100 processes, where each has a 1 megabyte/s connection, both for sending and receiving, in full-duplex. The average latency of the system is 100 ms and every message has 1 kilobyte length. Following our model, n is 100, the round length is 1 ms (since each process can send roughly one thousand messages per second), and δ is 100 rounds. Using distributors, we have latency $\delta + n - 2 = 100 + 100 - 2 = 198$ ms. Using a ring, we would have $\delta(n-1) = 100 \times (100-1) = 9900$ ms.

With Ridge, it takes $|Q|\delta$ rounds for a message to pass through all the acceptors in the quorum Q and arrive at the distributing learner. From that point on, the learners in $L \setminus Q$ (i.e., the learners that are not acceptors in Q) take turns being the distributor, so $(\delta + |L \setminus Q| - 2)$ extra rounds are necessary for all remaining processes to receive the message. Therefore, the latency of Ridge is: $(|Q|+1)\delta + |L \setminus Q| - 2$.[3]

6 Performance evaluation

We present here the results of experiments with Ridge, (Multi-)Ring Paxos [10,12], LibPaxos[4] and Spread [14]. In Section 6.1, we detail the environment used and the parameters given to the different protocols. In Section 6.2, we show the results for broadcast (i.e., multicast with a single group). In Section 6.3, we show how

[2]Since l_d already has the message, it only sends it $|L \setminus Q| - 1$ times, but it is simpler to approximate to $|L \setminus Q|$. On the other hand, $a_{(f+1)}$ sends $id(m)$ to all $|L|$ learners of the system, letting them know that m was decided, but we consider that the size of $id(m)$ is negligible.

[3]If learners are delivering messages from multiple ensembles, it may be necessary to wait for Δ (maximum time between two consecutive null messages) more rounds.

[4]https://bitbucket.org/sciascid/libpaxos

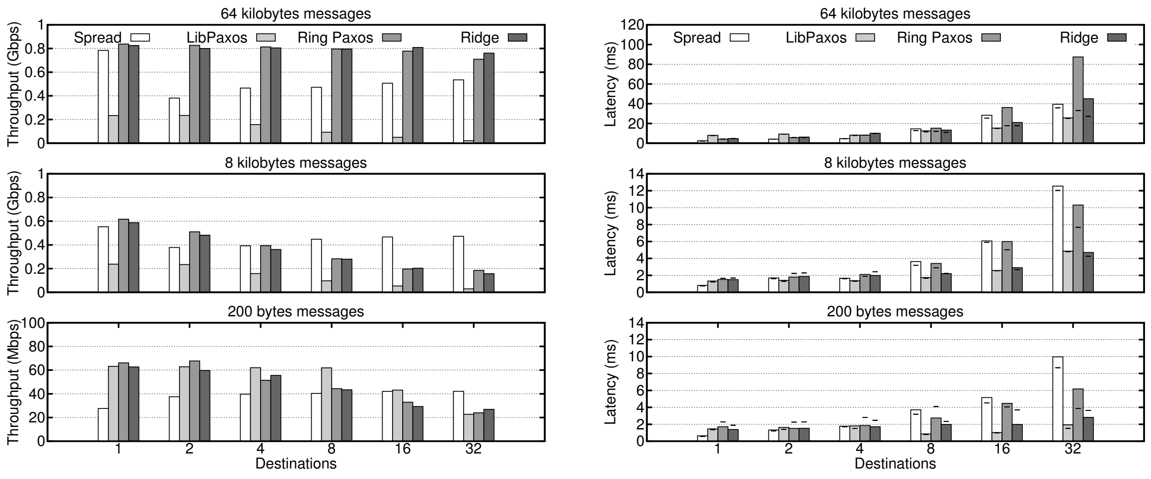

Figure 1: Throughput (maximum) and latency (with the system at maximum power) as the number of destinations increases.

throughput scales with the number of multicast groups, for each protocol. Finally, in Section 6.4, we show results for optimistic deliveries.

6.1 Environment setup and configuration parameters

We conducted all experiments on a cluster that had two types of nodes: (a) HP SE1102 nodes, equipped with two Intel Xeon L5420 processors running at 2.5 GHz and with 8 GB of main memory, and (b) Dell SC1435 nodes, equipped with two AMD Opteron 2212 processors running at 2.0 GHz and with 4 GB of main memory. The HP nodes were connected to an HP ProCurve 2920-48G gigabit network switch, and the Dell nodes were connected to another, identical switch. Those switches were interconnected by a 20 Gbps link. The average round-trip latency measured with ping was 133 μs. All nodes ran CentOS Linux 6.5 with kernel 2.6.32 and had the Oracle Java SE Runtime Environment 8. Clocks were kept synchronized with NTP for better results with Ridge and Multi-Ring Paxos with multiple groups and to collect consistent measurements from different processes in the system.

In all our experiments, there are clients and servers: each client multicasts a message to a group of servers, receives a reply from one of the servers, then sends another message, and so on. In all experiments with Paxos-based protocols, each Paxos group had 3 acceptors, with in-memory storage. One of such acceptors was also a proposer acting as coordinator and the servers were pure learners (i.e., being neither acceptors nor proposers). Each client message was sent to a coordinator, which proposed the message in the next consensus instance. In the case of Multi-Ring Paxos, we used a scheme similar to that of Ridge: there was a ring r_i for each multicast group g_i and a ring r_{all} for messages for multiple groups. Each process subscribing to messages from g_i would learn decision from r_i and r_{all}. Null and skip messages were sent every $\Delta = 1$ ms. LibPaxos implements Paxos in C. In LibPaxos, the coordinator sends each proposal directly to all acceptors, which then send their Phase 2B messages directly to the the learners. Upon receiving a Phase 2B message from a majority of acceptors, the learners declare the value as decided. This is done to minimize latency, although it may be detrimental to the maximum throughput. All Paxos-based protocols used TCP for communication.

As a reference, we included experiments with Spread [14], version 4.4.0. In our Spread deployments, each server had a local Spread daemon, and all daemons belonged to the same Spread segment (within a segment, daemons are arranged in a ring and send message payloads using ip-multicast). Each server joined one multicast group and had clients connected to it. The message service type used was “safe”.

6.2 Broadcast results: performance vs. number of destinations

We show here the results of experiments with a single multicast group (i.e., broadcast). The throughput we report is the maximum, which we find by increasing the system load as long as the throughput increases. To report latency, we look for the load that leads the system to its maximum *power*, which we define as the ratio between throughput and latency. As load increases, power tends to increase as well, until latency increases faster than throughput, indicating that the system is overloaded and that latency values are no longer reliable.

Ridge has roughly the same throughput achieved by Ring Paxos. Figure 1 (left) shows that, for 64 kB messages, both algorithms reach 0.8 Gbps in a 1 Gbps network, with very little variation as the number of destinations increases. This comes from the fact that both algorithms use throughput-optimal dissemination techniques: a ring (Ring Paxos) and alternating distributors (Ridge). For 8 kB and 200 B messages, there is a throughput drop as the number of destinations increases. LibPaxos, which is implemented in C and is optimized for latency in detriment of throughput, has the lowest throughput of all algorithms, except for very small messages, where CPU is more likely to be the bottleneck than network. Spread uses ip-multicast for disseminating messages, having a somewhat constant throughput of 0.4–0.5 Gbps for 64 kB and 8 kB messages, and around 40 Mbps for 200 B messages. With 64 kB messages and a single destination, Spread had the same 0.8 Gbps throughput of Ridge. This is a special case for Spread, as there is a single server, i.e., a ring with a single Spread daemon.

Figure 1 (right) shows latency results. Bars and dashes represent the 95th percentile and average latencies, respectively. Among the protocols that use Paxos, LibPaxos had the lowest latency in most cases. However, Ridge's latency was comparable to that of LibPaxos. The latency of Spread was not as sensitive to message size as the latency of the other protocols tested, since Spread uses ip-multicast to disseminate payloads. Still, even not making use of ip-multicast, Ridge has lower latency than Spread in many cases. This happens because Spread uses a ring topology: although it uses ip-multicast to disseminate the payload of each message, such message can only be safely delivered after relevant data has traversed the whole ring.

6.3 Multicast results: performance vs. number of groups

In this section, we show how throughput scales with the number of groups in the system. In our experiments, each group had four servers, and each message was sent to a single group. Ideally, the aggregate throughput would scale linearly with the number of groups. We do not show multicast results for LibPaxos because it only offers a broadcast API (by means of consensus with Paxos).

Figure 2 (left) shows the scalability of maximum throughput for each protocol, while latency is reported on Figure 2 (right). The maximum throughput of each protocol is normalized by its absolute value with a single group, which is shown in the graph. We can see that both Ridge and Multi-Ring Paxos have very similar throughput scalability in all cases. This is expected, since both algorithms use one separate group of Paxos acceptors for each multicast group. By doing this, these protocols are able to scale the aggregate throughput by adding independent sets of Paxos acceptors. This scheme allows Ridge and Multi-Ring Paxos to have nearly ideal scalability for 64 kB messages. Ridge's throughput scaled 8 times, proportionally to the number of groups. These algorithms had good scalability for 8 kB and 200 B messages, with throughput scaling 4 times with 8 groups. Spread uses a single ring for all servers in the system, even if they subscribe to different groups. Because of that, network was a bottleneck and Spread's throughput did not scale much beyond 0.5 Gbps with 64 kB and 8 kB messages. This result is similar to what Spread achieved for broadcast with multiple destinations (here, each group has four destinations). Spread scaled better with small messages, as network was likely not the bottleneck, and having more groups allowed Spread's throughput to increase.

6.4 Results for optimistic deliveries

We also benchmarked Ridge with optimistic deliveries enabled. We had a number of clients sending 64 kB messages to a multicast group, where a server delivered messages both atomically and optimistically, comparing the sequences. A *mistake* happened when the i-th atomic delivery in the server did not match the i-th optimistic delivery. The results we found are displayed in Figure 3, where we show mistakes as a percentage of the optimistic deliveries. We also highlight the load with which the system had maximum power, i.e., the maximum ratio between throughput and latency (of the atomic delivery).

Figure 3 (top) shows that the optimistic delivery has significantly lower latency than the conservative delivery, and this difference tends to increase with the load on the system. This happens because each message is delivered optimistically after being received directly from its origin, while the atomic delivery of a message only happens after the message has been decided in consensus. In Figure 3 (bottom), we can see

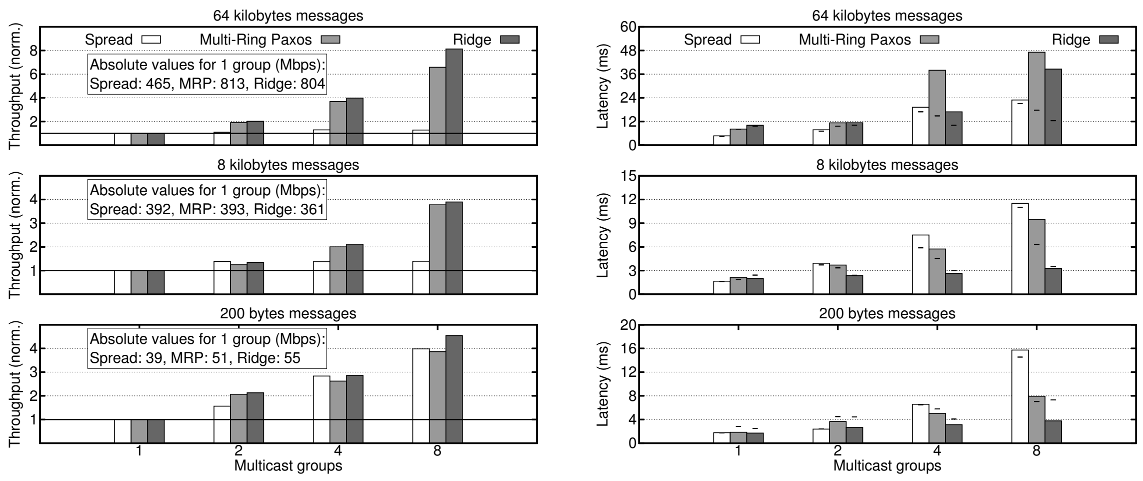

Figure 2: Throughput (maximum) and latency (with the system at maximum power) for each multicast protocol as the number of groups increases. The throughput of each protocol was normalized by the protocol's throughput with a single multicast group.

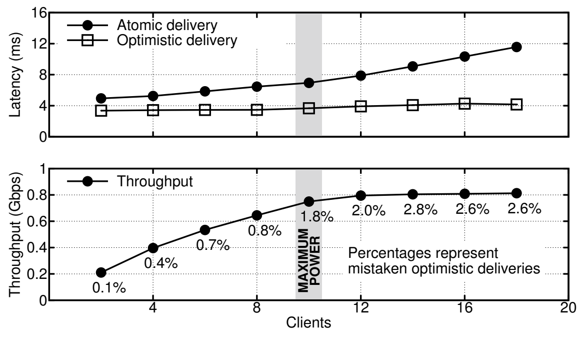

Figure 3: Latency, throughput and mistakes with the optimistic delivery enabled.

that the rate of mistakes tends to increase with the load, which is expected: more messages lead to higher chance of out-of-order optimistic deliveries. However, even with the system heavily loaded, the percentage of mistakes never reached 3%.

We can see that delivering every message twice did not hurt throughput. Ridge's maximum throughput of 0.8 Gbps was the same found previously (Section 6.2) without optimistic deliveries. With optimistic deliveries enabled, Ridge relies on the process multicasting a message to send (with reliable multicast [18]) the actual message payload to each Paxos learner directly. This is done to create a route that bypasses consensus, allowing optimistic deliveries to happen. When this happens, the acceptors do not send the message payload to the learners, but only its id, so that the learners do not waste downstream bandwidth to receive the same message twice. Although each individual client sends messages multiple times (to the learners directly, then to the Paxos coordinator), the throughput of the system remains unaltered.

7 Related work

In [8], the authors proved that ring topologies allow systems to achieve optimal throughput. Some protocols that benefit from such topologies are LCR [8], Totem [19], Spread [14] and Ring Paxos [10]. LCR arranges processes in a ring and uses vector clocks to ensure total order. One disadvantage of LCR in comparison to Ridge is that it requires *perfect failure detection*: suspecting that a correct process failed is not tolerated. Totem is a ring-based protocol based on Transis [20] that provides total order for messages. Spread, which is based on Totem, relies on daemons interconnected as a ring to order messages, while message payloads are disseminated using ip-multicast. Finally, Ring-Paxos deploys Paxos [17] processes in a ring to maximize throughput. A problem of all such ring-based protocols is that their latency is proportional to the size of the system times the network point-to-point latency. S-Paxos [11] is an atomic broadcast protocol that achieves high throughput by offloading the coordinator. Unlike Ridge, S-Paxos does not implement atomic multicast and does not optimize for latency.

Skeen's algorithm (presented in [21]) is possibly the first atomic multicast algorithm. Even though it is not fault-tolerant, it is *genuine*: processes communicate only if they actually have application messages to exchange [22]. In Skeen's algorithm, the destination processes of a message m exchange timestamps and eventually decide on m's final timestamp. Messages are then delivered based on their final timestamps. Several extensions have been proposed to Skeen's protocol, aiming to provide fault-tolerance [22–25]. The basic idea of such extensions is to replace each process with a fault-tolerant group of processes that act as one single entity by means of consensus, as in state-machine replication [26].

Multi-Ring Paxos [12, 13] achieves very high throughput by increasing the number of multicast groups. Each group uses Ring Paxos to solve consensus and message streams from different groups are merged as proposed in [27]. Ridge also uses Paxos and deterministic merge, but it does not use a ring overlay. Instead, Ridge employs alternating distributors, being also capable of reaching very high throughput. Multi-Ring Paxos merges messages from multiple rings in a round-robin fashion, assuming that all rings produce the same number λ of decisions per time unit; if a ring does not have enough application messages to reach λ, *skip-messages* are created. If a ring produces more than λ decisions per time unit, performance likely deteriorates. Ridge's merge function is different, using timestamps taken from the processes' system clocks; these timestamps are also used to deliver messages optimistically. The merge functions of both Ridge and Multi-Ring Paxos are sensitive to clock synchronization: the better the clocks are synchronized, the lower is the average latency. Moreover, both the theoretical analysis and our experiments show that Ridge has throughput similar to that of Multi-Ring Paxos, with significantly lower latency for the atomic delivery, and even lower latency for the optimistic delivery.

Regarding optimistic deliveries, optimistic atomic broadcast [5] relies on messages being spontaneously delivered through the network in the same order at all destinations. If this doesn't happen, the algorithm runs consensus to ensure that a final, totally-ordered delivery is also done. In [28], the authors propose a technique that approximates spontaneous ordering in a wide-area setting. The idea is for each process to insert artificial delays in incoming messages, so that the resulting artificial latency between each process and a given sequencer is the same. Fast Paxos [29] allows messages to be sent directly to the acceptors (bypassing the coordinator), saving time. However, if the acceptors receive those messages in different orders, classic Paxos is run to ensure total order. In [6], the authors propose an optimistic atomic multicast algorithm. Such algorithm is *quasi-genuine*, in the sense that processes only communicate if they “are able to" multicast messages to each other. The system's configuration determines which processes can communicate with one another, and such configuration can be changed dynamically. Such optimistic atomic multicast does not optimize throughput, unlike Ridge.

8 Conclusion

This work introduces Ridge, an atomic multicast protocol that combines high throughput, low latency, scalability and optimistic deliveries. Unlike previous works, it achieves such high throughput without resorting to ring topologies nor to ip-multicast. Instead, Ridge makes use of alternating distributors, which is a technique capable in theory of achieving optimal throughput, while having lower latency than a ring. We conducted a number of experiments that prove that this also holds in practice, with different numbers of destinations and multicast groups. Finally, we demonstrate that enabling optimistic deliveries allows messages to be delivered faster with even lower latency than that of the atomic delivery. Despite the fact that each message is sent through different routes and delivered twice, enabling optimistic deliveries does not reduce the maximum throughput of the system.

Acknowledgements

This work was supported in part by CAPES PVE (Special Visiting Researcher) Grant No. 88881.062190/2014-01 and by FAPESP BEPE Grant No. 2013/21651-0.

References

[1] X. Défago, A. Schiper, and P. Urbán, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *CSUR*, vol. 36, pp. 372–421, Dec. 2004.

[2] L. Lamport, “Generalized consensus and paxos,” Tech. Rep. MSR-TR-2005-33, Microsoft Research (MSR), Mar. 2005.

[3] Y. Mao, F. P. Junqueira, and K. Marzullo, “Mencius: building efficient replicated state machines for wans,” OSDI, 2008.

[4] F. Pedone and A. Schiper, “Generic broadcast,” DISC, 1999.

[5] F. Pedone and A. Schiper, “Optimistic atomic broadcast,” DISC, 1998.

[6] C. E. Bezerra, F. Pedone, B. Garbinato, and C. Geyer, “Optimistic atomic multicast,” ICDCS, 2013.

[7] R. Guerraoui, D. Kostic, R. R. Levy, and V. Quema, “A high throughput atomic storage algorithm,” ICDCS, 2007.

[8] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, “Throughput optimal total order broadcast for cluster environments,” *TOCS*, vol. 28, pp. 5:1–5:32, July 2010.

[9] M. Kapritsos and F. P. Junqueira, “Scalable agreement: Toward ordering as a service,” HotDep, 2010.

[10] P. Marandi, M. Primi, N. Schiper, and F. Pedone, “Ring Paxos: A high-throughput atomic broadcast protocol,” DSN, 2010.

[11] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, “S-paxos: Offloading the leader for high throughput state machine replication,” SRDS, 2012.

[12] P. Marandi, M. Primi, and F. Pedone, “Multi-Ring Paxos,” DSN, 2012.

[13] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, “Building global and scalable systems with atomic multicast,” Middleware, 2014.

[14] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, “The spread toolkit: Architecture and performance,” *Center for Networking and Distributed Systems (CNDS) Technical report CNDS-2004-1*, 2004.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty processor,” *JACM*, vol. 32, no. 2, pp. 374–382, 1985.

[16] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *JACM*, vol. 35, no. 2, pp. 288–323, 1988.

[17] L. Lamport, “The part-time parliament,” *TOCS*, vol. 16, pp. 133–169, May 1998.

[18] H. Garcia-Molina and A. Spauster, “Ordered and reliable multicast communication,” *TOCS*, vol. 9, pp. 242–271, Aug. 1991.

[19] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella, “The Totem single-ring membership protocol,” *TOCS*, vol. 13, no. 4, pp. 311–342, 1995.

[20] Y. Amir, D. Dolev, S. Kramer, and D. Malki, “Transis: a communication subsystem for high availability,” FTCS, 1992.

[21] K. P. Birman and T. A. Joseph, “Reliable communication in the presence of failures,” *TOCS*, vol. 5, pp. 47–76, Feb. 1987.

[22] R. Guerraoui and A. Schiper, “Genuine atomic multicast in asynchronous distributed systems,” *TCS*, vol. 254, no. 1-2, pp. 297–316, 2001.

[23] N. Schiper and F. Pedone, “On the inherent cost of atomic broadcast and multicast in wide area networks,” ICDCN, 2008.

[24] U. Fritzke and P. Ingels, “Transactions on partially replicated data based on reliable and atomic multicasts,” ICDCS, 2001.

[25] L. Rodrigues, R. Guerraoui, and A. Schiper, “Scalable atomic multicast,” ICCCN, 1998.

[26] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *CACM*, vol. 21, no. 7, pp. 558–565, 1978.

[27] M. K. Aguilera and R. E. Strom, “Efficient atomic broadcast using deterministic merge,” PODC, 2000.

[28] A. Sousa, J. Pereira, F. Moura, and R. Oliveira, “Optimistic total order in wide area networks,” SRDS, 2002.

[29] L. Lamport, “Fast paxos,” *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[30] T. D. Chandra, V. Hadzilacos, and S. Toueg, “The weakest failure detector for solving consensus,” *JACM*, vol. 43, no. 4, pp. 685–722, 1996.

Appendix

A Correctness Proof (Sketch)

As described in Section 2, we assume a partially synchronous system: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called GST (Global Stabilization Time). Before GST, there are no bounds on message delay or relative process speed. After GST, such bounds exist but are unknown. We assume that after GST, all remaining processes are correct.

A.1 Consensus

We provide here a proof sketch for the correctness of Ridge. We prove that the optimized version of Paxos proposed here (Algorithm 2) ensures properties (ii) and (iii) of consensus. Property (i) holds trivially from the algorithm.

Proposition 1. *(ii) No two processes decide different values.*

Proof sketch. Let v and v' be two decided values, with unique identifiers *id(v)* and *id(v′)*, and proposed by coordinators c and c', respectively. We prove that *id(v)* = *id(v′)*.

Let r be the round, with quorum Q, in which some acceptor a sent a DECISION message with *id(v)* to all learners, and let r' be the round, with quorum Q', in which some acceptor a' sent a DECISION message with *id(v′)* to all learners.

In Ridge, a sends the DECISION message with *id(v)* after: (a) c receives $f+1$ messages of the form $\langle$PHASE 1B, r, *, *$\rangle$; (b) c selects value *v-val* $= v$ with the highest round number *v-rnd* among the set M_{1B} of PHASE 1B messages received, or picking a value *v-val* if *v-rnd* $=0$; (c) $f+1$ different acceptors (a majority of the acceptors) have cast a vote for value v in round r.

When acceptor a receives a message of the form $\langle$PHASE $2, r, v, Q, f\rangle$, it is equivalent to a receiving a message $\langle$PHASE 2B, $r, v\rangle$ (as in Algorithm 1) from f other acceptors. That's because each acceptor forwards a message of form $\langle$PHASE $2, r, v, Q, count\rangle$ to the next acceptor in Q only if r is higher than or equal to the previous *v-rnd* held by that acceptor. If a also casts a vote for $\langle r, v\rangle$, this is equivalent to a sending a message $\langle$PHASE $2, r, v, Q, f+1\rangle$ to itself, so a knows that v was decided in round r, equivalently to coordinator c receiving $f+1$ $\langle$PHASE 2B, $r, v\rangle$ messages in the original Paxos algorithm. That's why a can safely send a $\langle$DECISION, *id(v)*$\rangle$ message to all learners. Let M_{2B} be the set of such equivalent $f+1$ Phase 2B messages. Now consider a system running original Paxos where the coordinator c received the same sets of messages M_{1B} and M_{2B}. In this case, c would send a DECIDE message with *id(v)* as well. Since the same reasoning can be applied to coordinator c', and Paxos implements consensus, we have that *id(v)* = *id(v′)*.

□

Proposition 2. *(iii) If one (or more) correct process proposes a value then eventually some value is decided by all correct processes.*

Proof sketch. After GST, processes eventually select a correct coordinator c. Coordinator c considers a quorum Q of acceptors, and c sends a Phase 1A message to all acceptors in the ensemble. If all acceptors in the majority quorum Q are correct and c does not suspect that any of them has failed, then all messages exchanged between coordinator and acceptors in Q are received and all correct processes eventually decide some value. If c suspects that any acceptor in Q has failed, the protocol falls back to the original Paxos algorithm, which guarantees property (iii) of consensus. If any acceptor a_f in Q has in fact failed, c eventually suspects that a_f has failed. The weakest failure detector $\diamond W$ [30] is enough to provide this guarantee: There is a time after which every process p_f that crashes is always suspected by some correct process p_s. Process p_s then sends to c a notification n_f that p_f is suspected of failing. Since both p_s and c are correct after GST, n_f is received by c.

□

A.2 Atomic multicast

The following applies to Algorithm 3 and can be trivially extended to Algorithm 4.

Lemma 1. *Let t be a timestamp, and ε a set of ensembles. Each correct process p that is a learner in every ensemble $e \in \varepsilon$ eventually decides a message $m : m.ts > t$, where m was decided by e, for every $e \in \varepsilon$.*

Proof sketch. Each ensemble $e \in \varepsilon$ eventually elects a correct coordinator c after GST and proposes messages (possibly null) at least every Δ time units. Decided messages from an ensemble are assigned monotonically increasing final timestamps, so p eventually decides a message m such that $m.ts > t$, for every ensemble e in ε. □

Proposition 3. (VALIDITY) *If a correct process multicasts m, then every correct process in γ delivers m.*

Proof sketch. To multicast m, a correct process p sends m to the coordinator c of an ensemble e, and p keeps sending m until it reaches reaches the correct coordinator. After GST, a correct process c is eventually elected as coordinator of e. Since both p and c are correct, c delivers m and proposes it. From the properties of consensus, all correct learners of e decide m. Therefore, and from Lemma 1, l eventually delivers m.

□

Proposition 4. (INTEGRITY) *For any message m, every process p in γ delivers m at most once, and only if some process has multicast m previously.*

Proof sketch. For m to be delivered, it must have been proposed by the coordinator c of an ensemble e. The proof that m was multicast by some process is immediate from Algorithm 3: c only proposes m after receiving m from a process q that multicast m. After GST, e will eventually have a correct coordinator c, so proving that m is delivered only once is also obvious from the algorithm: c proposes m only once in some consensus instance k. After deciding m, learner p of e eventually delivers m (from Lemma 1).

□

Proposition 5. (UNIFORM AGREEMENT) *If a process delivers m, then every correct process in γ delivers m.*

Proof sketch. For a process p, which subscribes to a set of ensembles ε, to deliver a message m, p must be a learner that decided m after m was accepted by a majority of acceptors in some ensemble $e \in \varepsilon$. Since e has a majority of correct acceptors, at least one correct acceptor a knows that m was decided at an instance k. After GST, eventually a correct coordinator c is elected for e, proposing messages every Δ time units at least. From the properties of consensus, every correct learner of e will decide such messages, knowing if there is any gap in the message stream from e. In other words, there will eventually be a message decided in instance $k' : k' > k$ in e. Since at least the correct acceptor a knows about m, m can be recovered by any correct learner l that missed m. This way, l is sure to eventually know about m, being able to deliver it once a message $m' : m'.ts > m.ts$ has been decided in each ensemble ε, which is guaranteed to eventually happen after GST (Lemma 1).

□

Proposition 6. (ATOMIC ORDER) *No two processes p and q deliver m and m' in different orders, and the delivery order is acyclic.*

Proof sketch. Message m is decided in an ensemble e, and the learners of e will decide the same final timestamp for m, adjusting it if necessary according to a deterministic procedure. The same reasoning happens to m', which is decided in an ensemble e'. Delivery is based on timestamp order, that is, if $m.ts < m'.ts$ (these are final timestamps), m' can only be delivered after m. Suppose that p delivers m before m', and that q delivers m' before m. One way of this happening could be if p decided that $m.ts < m'.ts$, while q decided that $m.ts > m'.ts$, which contradicts the properties of consensus and the deterministic adjustment of timestamps by learners. Another possibility would be that a message m was not decided by a process p to which both m and m' were multicast. This means that there is an instance k of e that p did not decide. From the algorithm, a learner p does not deliver m before a gap-free sequence of decisions of e has been received by p. Also, assume that messages are delivered in cyclic order: m_1 is delivered before m_2 by p, m_2 is delivered before m_3 by q, and m_3 is delivered before m_1 by r. This means that $m_1.ts < m_2.ts \wedge m_2.ts < m_3.ts \wedge m_3.ts < m_1.ts$, which is a contradiction. □

B Other results

Here, we show some other results that further demonstrate Ridge's performance. Figure 4 shows the latency of each protocol with a single client and one group. The goal of running benchmarks with a single client was to reduce queueing effects as much as possible, so that the latency reported would be mostly due to the way they are propagated by each protocol. Figure 5 shows the CDF (cumulative distribution function) of latency for each protocol, with a single client, one group and 32 destinations.

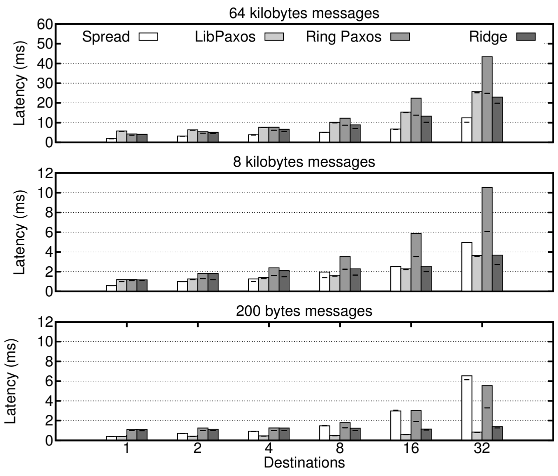

Figure 4: Latency of each protocol for one group, with one single client.

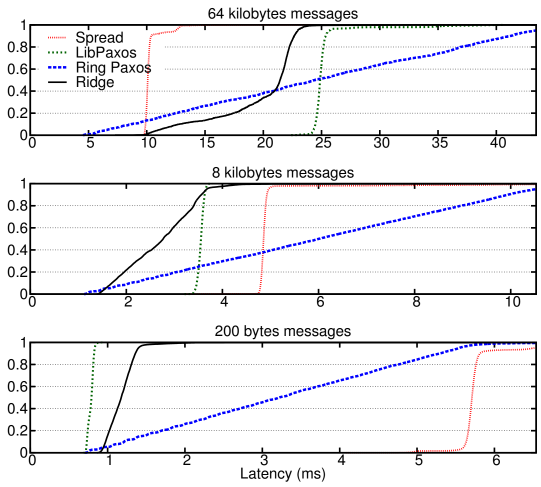

Figure 5: Latency CDFs of each protocol for one group, with 32 destinations and one client.

Figure 6 shows the throughput of each protocol with one group and the system at its maximum power. The throughput at maximum power is usually lower than the maximum throughput because, at the maximum throughput, the system is typically overloaded and latency tends to increase sharply with the number of clients. We used the point of maximum power to report latency (back in Figure 1), maximizing the ratio between throughput and latency. In Figure 7, we can see the latency CDFs for the system at maximum power, with one group and 32 destinations.

As for multicast, Figure 8 shows the absolute values for maximum throughput for each protocol and increasing number of multicast groups. Figure 9 shows what those numbers mean in terms of *scalability efficiency*, that is, how close each protocol is to the ideal throughput for each number of groups. To find the ideal throughput of a protocol with n groups, we multiply the throughput $t(1)$ measured with a single group by n, and then compare the result with the throughput $t(n)$ measured with n groups. The scalability efficiency for n group is, then, $t(n)/(n \times t(1))$. Thus, the ideal scalability efficiency for any number of groups is 1.

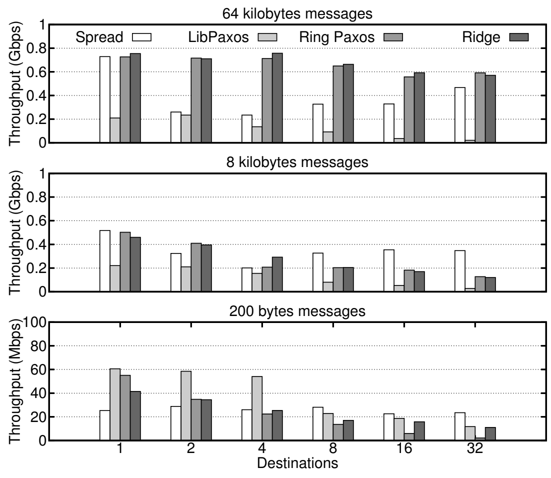

Figure 6: Throughput of each protocol for one group, with the system at maximum power.

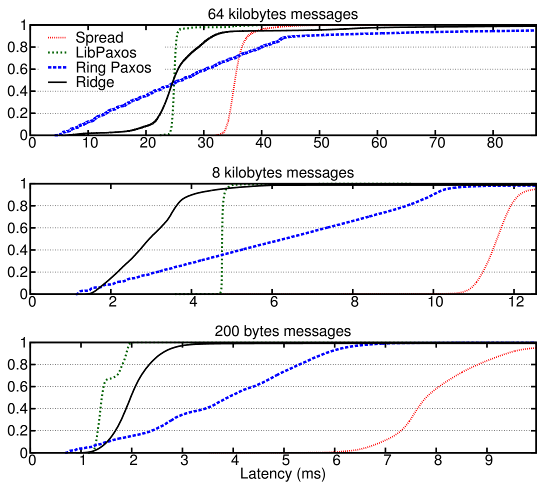

Figure 7: Latency CDFs of each protocol, with 32 destinations, with the system at maximum power.

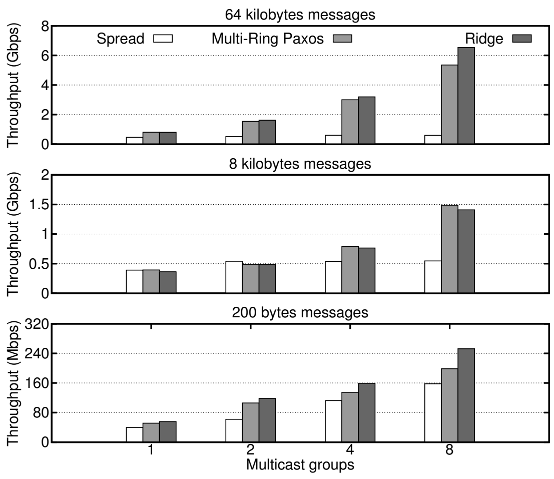

Figure 8: Maximum throughput of each protocol for different numbers of multicast groups.

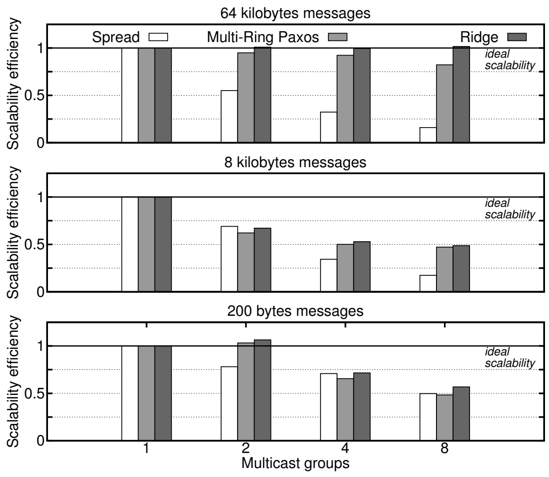

Figure 9: Scalability efficiency of each protocol. A value of 1 for n groups means that the throughput measured with n groups was n times bigger than the throughput measured with a single group.