RamCast: RDMA-based Atomic Multicast

Long Hoang Le
Università della Svizzera italiana (USI)
Switzerland

Mojtaba Eslahi-Kelorazi
Università della Svizzera italiana (USI)
Switzerland

Paulo Coelho
Federal University of Uberlândia
Brazil

Fernando Pedone
Università della Svizzera italiana (USI)
Switzerland

ABSTRACT

Atomic multicast is a group communication abstraction useful in the design of highly available and scalable systems. It allows messages to be addressed to a subset of the processes in the system reliably and consistently. Many atomic multicast algorithms have been designed for the message-passing system model. The paper presents RamCast, the first atomic multicast protocol for the shared-memory system model. We design RamCast by leveraging Remote Direct Memory Access (RDMA) technology and by carefully combining techniques from message-passing and shared-memory systems. We show experimentally that RamCast outperforms current state-of-the-art atomic multicast protocols, increasing throughput by up to 3.7× and reducing latency by up to 28×.

CCS CONCEPTS

• **Computer systems organization** → *Distributed architectures*; • **Computing methodologies** → *Distributed algorithms.*

KEYWORDS

group communication, atomic multicast, RDMA

ACM Reference Format:
Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo Coelho, and Fernando Pedone. 2018. RamCast: RDMA-based Atomic Multicast. In *Middleware '21: ACM/IFIP Middleware conference, December 06–10, 2021, Quebec, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Today's online services are expected to operate uninterruptedly despite server failures. Many such services must also handle ever-increasing demand without performance hiccups. Services that match these expectations are deemed highly available and scalable. Many years of research in dependable distributed systems have deepened the understanding of how to design systems that can tolerate failures. A golden rule is that abstractions can significantly reduce complexity, and avoid design and programming errors. For example, state machine replication, a *de facto* standard for fault tolerance, requires replicas to order requests. But ordering requests in a distributed system prone to failures is difficult [22]. By relying on atomic broadcast, an abstraction equivalent to consensus [11, 25], system designers can decompose ordering from execution in state machine replication, as atomic broadcast provides reliable and ordered delivery of requests.

In order to both scale performance and tolerate failures, service state is typically sharded and each shard is replicated (e.g., [4, 13, 26]). Atomic multicast is a group communication abstraction that generalizes atomic broadcast by allowing requests to be propagated to groups of processes (in this case shards) with reliability and order guarantees. Intuitively, all non-faulty processes addressed by a request must deliver the request and processes must agree on the order of delivered requests. Atomic multicast offers strong communication guarantees and should not be confused with network-level communication primitives (e.g., IP-multicast), which offer "best-effort" guarantees.

Since messages can be multicast to different sets of destinations and interleave in non-obvious ways, implementing message order in a distributed setting is challenging. Some atomic multicast protocols address this challenge by ordering all messages using a fixed group of processes, regardless of the destination of the messages. To be efficient, however, an atomic multicast algorithm must be *genuine*: only the message sender and destination processes should communicate to propagate and order a multicast message [24]. A genuine atomic multicast is the foundation of scalable and fault-tolerant systems, since it does not depend on a fixed group of processes, and ensures reliable communication [12].

Many atomic multicast algorithms have been designed for the message-passing system model (e.g., [7, 9, 12, 17, 23, 39]). This paper presents RamCast, the first atomic multicast protocol tailor-made for the shared-memory model. Our motivation is practical: recent years have seen widespread development and adoption of Remote Direct Memory Access (RDMA) technology. RDMA extends the traditional send and receive communication primitives with read and write operations on shared memory. Essentially, RDMA provides a node with the capability to read and write the memory of another node, without involving its processor. In addition, RDMA offers the possibility for a process to safeguard its memory by specifying which processes can read or write which regions of its memory. This guarantee is quite powerful. In particular, if every process revokes the write permission of other processes before writing to shared memory, then a process that writes successfully knows that it executed in isolation, without having to take additional steps (e.g., reading the memory). Protocols can leverage this property to optimize performance [1].

Middleware '21, December 06–10, 2021, Quebec, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

Designing an efficient RDMA-based atomic multicast protocol is not trivial, since RDMA's communication primitives vary substantially in performance [32, 40]. Figure 1 compares the latency of TCP/IP to RDMA's send/receive and read/write operations (setup details are presented in §5.2). RDMA primitives largely outperform TCP/IP communication because they bypass the network stack. RDMA shared-memory primitives deliver performance superior to message-passing primitives, although the advantage depends on the message size. In our environment, RDMA read and write operations have similar performance, unless the write operation can "inline" data (i.e., with 64-byte messages in our case) [40]. This happens because the interface adapter has the data available and does not need to retrieve data from the memory. RamCast uses remote writes only and avoids remote reads. There are two reasons for this design. First, most writes are small (e.g., acknowledgments) and can be inlined. Second, a process detects a write in shared memory with busy-polling reads, and reading from a process's own memory is faster than reading from the memory of another process. Consequently, process p can read process q's write more efficiently when q issues a (remote) write on p's memory and p issues busy-polling reads on its own memory.

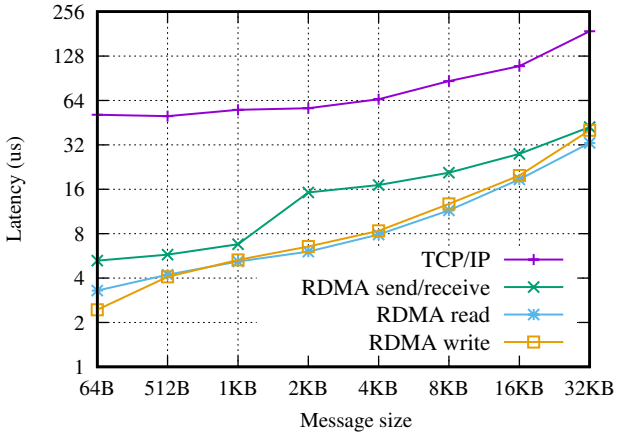

Figure 1: Performance comparison between communication primitives.

In order to deliver performance that largely outperforms the most efficient message-passing atomic multicast protocols, RamCast combines ideas from Skeen's genuine atomic multicast algorithm [10], a blocking algorithm that has been used as the basis for fault-tolerant atomic multicast protocols (e.g., [12, 23]), the leader-follower replication model, explored by atomic multicast and broadcast protocols (e.g., [2, 5, 23, 31]), and RDMA technology, recently used to boost the performance of distributed systems (e.g., [1, 32, 33, 40]). RamCast is the first shared-memory genuine atomic multicast algorithm that fully exploits RDMA capabilities. RamCast can order messages multicast to a single group after 2 RDMA write delays, at the leader process, and order messages multicast to multiple groups after 3 RDMA write delays, at both the leader and followers. As a reference, the most delay-efficient message-passing atomic multicast algorithm orders messages after 3 communication delays at the leader process and after 4 communication delays at the followers [23]. We have implemented and evaluated RamCast under various conditions. We show that RamCast outperforms current state-of-the-art atomic multicast protocols, increasing throughput by up to 3.7× and reducing latency by up to 28×.

The remainder of the paper is structured as follows. Section 2 presents the system model, a formal definition of the atomic multicast problem, and the basics of Remote Direct Memory Access. Section 3 details the RamCast protocol. We start with a short description of the ideas that have inspired RamCast, then describe its normal behavior, in the absence of failures, and how it handles failures. Section 4 presents our prototype, and Section 5 details its performance. Section 6 surveys related work and Section 7 concludes the paper.

2 BACKGROUND

In this section, we present the system model (§2.1), define the atomic multicast communication problem (§2.2), and overview RDMA, the key technology used by RamCast (§2.3).

2.1 System model

We assume a hybrid distributed system model in which processes can use both message-passing and shared-memory [1]. The system is composed of a set of client and server processes. Processes communicate by exchanging messages or accessing portions of each other's memory. Processes are *correct*, if they do not fail, or *faulty*, otherwise. In either case, processes do not experience arbitrary behavior (i.e., no Byzantine failures). Our protocols ensure safety under both asynchronous and synchronous execution periods. To ensure liveness, we assume the system is *partially synchronous* [20], that is, it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)*, and it is unknown to the processes. Before GST, there are no bounds on communication and processing delays; after GST, such bounds exist but are unknown.

A process can share memory regions with other processes and define permissions for those shared memory regions. Process q can read and write a register v in p's memory region mr with operations read(p, v) and write$(p, v, value)$, respectively. A permission associated with memory region mr defines disjoint sets of processes, R_{mr}, W_{mr}, and RW_{mr}, that can read, write, and read-write the registers in region mr. Process q has permission to read (respectively, write and read-write) v in p's mr if $q \in R_{mr}$ (resp., W_{mr}, RW_{mr}). A process can initially assign permissions for its shared memory regions and later change these permissions.

Processes can also communicate by exchanging messages over a set of directed links using primitives send(p, m) and receive(m), where p is the addressee of message m. We assume messages are unique. Communication links are reliable in that every message sent by a process p to another process q is guaranteed to be eventually delivered by q if both p and q are correct. Moreover, a message is received at most once, and only if it was previously sent.

2.2 Problem statement

Let Π be the set of server processes in the system and $\Gamma \in 2^{\Pi}$ the set of process groups in the system, where $|\Gamma| = k$. Groups are disjoint

and each group contains $n = 2f + 1$ processes, where f is the maximum number of faulty processes per group. The assumption about disjoint groups has little practical implication since it does not prevent collocating processes that are members of different groups on the same machine. Yet, it is important since atomic multicast requires stronger assumptions when groups intersect [24]. A set of $f + 1$ processes in group g is a *quorum* in g.

A process atomically multicasts a message m to groups in $m.dst$ by invoking primitive multicast(m), where $m.dst$ is a special field in m with m's destinations; a process delivers m with primitive deliver(m). We define the relation $<$ on the set of messages processes deliver as follows: $m < m'$ iff there exists a process that delivers m before m'.

Atomic multicast ensures the following properties:

- *Validity*: if a correct process p multicasts a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, deliver m.
- *Integrity*: for any process p and any message m, p delivers m at most once, and only if $p \in g$, $g \in m.dst$, and m was previously multicast.
- *Uniform agreement*: if a process p delivers a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, deliver m.
- *Uniform prefix order*: for any two messages m and m' and any two processes p and q such that $p \in g, q \in h$ and $\{g, h\} \subseteq m.dst \cap m'.dst$, if p delivers m and q delivers m', then either p delivers m' before m or q delivers m before m'.
- *Uniform acyclic order*: the relation $<$ is acyclic.

Atomic broadcast is a special case of atomic multicast in which there is a single group (i.e., Γ is a singleton).

We require atomic multicast protocols to be *genuine* [24]: an atomic multicast algorithm is genuine if in any run in which a message m is multicast, then for every process p that participates in ordering m, p is the process that multicasts m or $p \in g$ and $g \in m.dst$.

2.3 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a protocol that enables direct data access to the memory of a remote machine without involving the operating system and processor of the remote machine. RDMA implements the network stack in hardware, and provides both low latency and high bandwidth by bypassing the kernel and supporting zero-copy communication. RDMA provides two-sided operations (e.g., send, receive), one-sided operations (e.g., read, write), and atomic operations (e.g., compare-and-swap, fetch-and-increment). The two-sided operations involve the CPU of the remote host and rely on user-space memory copies. Thus, they introduce overhead when compared to one-sided RDMA verbs [18]. Besides, previous studies have established that remote write operations provide performance superior to remote reads, and send and receive operations, and much better performance than atomic operations [32, 33, 40]. In RamCast's normal operation, processes communicate using remote write operations only. We refrain from using remote read operations, and resort to send and receive operations when handling failures, since they lead to simpler logic.

RDMA provides three transport modes: Reliable Connection (RC), Unreliable Connection (UC) and Unreliable Datagram (UD). While RC and UC are connection-oriented and support only one-to-one data transmission, UD supports both one-to-one and one-to-many transmission without establishing connections. RC ensures data transmission is reliable and correct in the network layer, while UC does not have such a guarantee. In this work, we use RC to provide in-order reliable delivery. To establish a connection between two remote hosts, the RDMA-enabled network card (RNIC) on each host creates a logical RDMA endpoint known as a Queue Pair (QP), including a send queue and a receive queue for storing data transfer requests. Operations are posted to QPs as Work Requests (WRs) to be consumed and executed by the RNIC. When an RDMA operation is completed, a completion event is pushed to a Completion Queue (CQ). Operations can be made *unsignaled* by setting a flag in the WR; these verbs do not generate a completion event, and the application detects completion using application-specific methods. Each host makes local memory regions (MR) available for remote access by asking its OS to pin the memory pages that would be used by the RNIC. Both QPs and MRs can have different access modes (i.e., read-only or read-write). The hosts specify the access mode when initializing the QP or registering the MR, but the access mode can be dynamically updated later. The host can register the same memory for different MRs. Each MR then has its own access mode. In this way, different remote machines can have different access rights to the same memory region.

3 RAMCAST: RDMA-BASED ATOMIC MULTICAST

In this section, we recall the building blocks that inspired RamCast (§3.1) and present its design and algorithms. We start with an overview of RamCast (§3.2), then detail its data structures (§3.3) and algorithms in the absence of failures (§3.4) and in the presence of failures (§3.5). We argue for the correctness of RamCast (§3.6) and conclude with a few extensions to the protocol (§3.7).

3.1 Building blocks

RamCast leverages two ideas, Skeen's atomic multicast algorithm [10] and Protected Memory Paxos [1]. Skeen's algorithm orders messages multicast to multiple processes consistently but it does not tolerate failures. Protected Memory Paxos takes advantage of RDMA permissions to improve the efficiency of Paxos [35]. Like Paxos, it implements atomic broadcast (i.e., it assumes a single group of processes).

3.1.1 Skeen's atomic multicast. In Skeen's algorithm, each process assigns unique timestamps to multicast messages based on a logical clock [34]. The correctness of the algorithm stems from two basic properties: (i) processes in the destination of a multicast message first assign tentative timestamps to the message and eventually agree on the message's final timestamp; and (ii) processes deliver messages according to their final timestamp. These properties are implemented as follows.

(i) To multicast a message m to a set of processes, p sends m to the destinations. Upon receiving m, each destination updates its logical clock, assigns a tentative timestamp to m, stores

m and its timestamp in a buffer, and sends m's timestamp to all destinations. Upon receiving timestamps from all destinations in $m.dst$, a process computes m's final timestamp as the maximum among all received tentative timestamps for m.

(ii) Messages are delivered respecting the order of their final timestamp. A process p delivers m when it can ascertain that m's final timestamp is smaller than the final timestamp of any messages p will deliver after m (intuitively, this holds because logical clocks are monotonically increasing).

3.1.2 Protected Memory Paxos. In Paxos [35], to order a message m, the leader proposes m in a consensus instance. In the normal case, where there is a single leader, the followers accept the proposed message and reply to the leader. In Protected Memory Paxos, the followers grant exclusive write permission to their memory to the leader. If a new leader takes over, then it revokes the permission of the previous leader. To order m, the leader writes m in the memory of the followers. If the leader succeeds in writing the message in the memory of a quorum of followers, then no other leader took over, and the message is ordered.

Just like Paxos, to ensure that the new leader makes decisions that are consistent with the decisions of the previous leader, each leader associates a *round* to its proposed message. Rounds are unique across the system. When process q becomes leader, upon suspecting the failure of the current leader p, q must pick a round bigger than p's round. Process q then proceeds in two steps. First, q needs to acquire permission from a quorum of processes, which it does by contacting all processes and providing its chosen round. Processes grant write permission to q if the provided round is bigger than the round of the process that currently holds the write permission. Second, q must check whether other processes have already accepted any values. If so, q must propose the value that has been accepted in the largest round; otherwise, q can propose a new value.

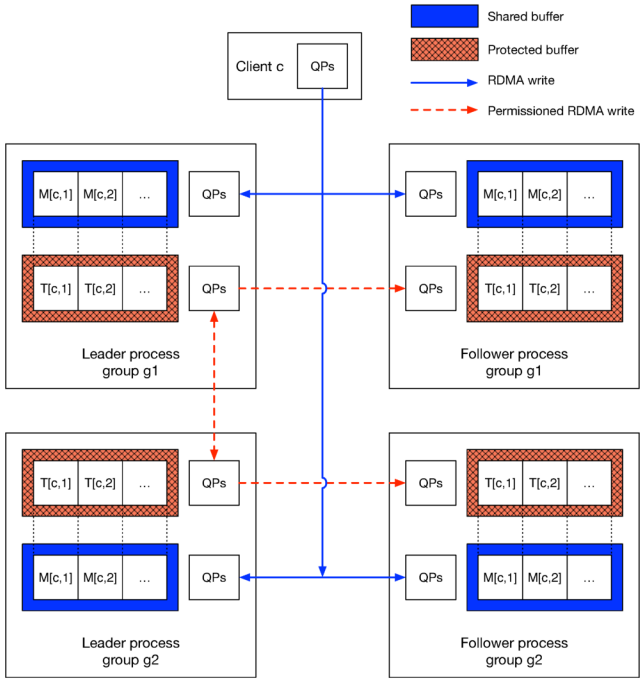

Figure 2: RamCast's memory layout. The normal execution proceeds in three steps: in step 1, the client writes a multicast message in the memory of all destination processes; in step 2 the leader of each destination group proposes and writes a timestamp for the message in the memory of its followers and other leaders; in step 3 the leaders propagate the timestamp written by other leaders, and the followers acknowledge the proposed timestamps. The message is delivered after step 3.

3.2 RamCast design and architecture

Figure 2 depicts the various components and memory layout of RamCast. Processes within each group coordinate using the leader-follower model [2, 5, 23, 31]. Each server process has a fixed-size buffer per client, analogously to other RDMA-based systems [2, 18, 42, 46]. A buffer in RamCast is divided into two parts, a message buffer M, and a timestamp buffer T. Message buffer M is a shared memory region that can be read and written by any processes, including the client process the buffer is associated with. Timestamp buffer T is protected and can only be written by the leader of each group; the buffer can be read by any processes. Each slot in M, with a multicast message m, has a corresponding slot in T, with m's timestamp.

Clients keep a copy of the remote head and tail pointer of their buffer at each server. A client increases the remote tail after writing to the shared memory. The server process updates the head pointer on the client buffer after handling the message. Each process p periodically polls the memory cell at the head position of each connected QPs to detect new messages.

RamCast consists of the following main components:

- *Memory management.* This component handles the shared buffer and the protected buffer (detailed in §3.3). While all processes have read and write access to the shared buffer of a process, only the leader of each group has write permission to the protected buffer of a process.
- *RDMA communication.* This component provides functions to read and write remote memory, used by the normal execution component, and to send and receive messages, used by the failure handling module.
- *Normal execution.* The normal protocol execution (detailed in §3.4) is invoked when there is a sole leader per group with support of at least a majority of processes in its group. A leader is responsible for proposing the group's timestamp for a new multicast message, and propagating timestamps from other groups to followers of its own group.
- *Failure handling.* Upon detecting the failure of a leader, processes in a group elect a new leader. The new leader must ensure that its execution is consistent with the execution of the previous leader (detailed in §3.5).
- *Leader election.* RamCast requires processes to detect a slow or crashed leader, and elect a new leader. Leader election is not assumed to be perfect: the protocol ensures safety despite multiple leaders in a group. To ensure progress, though, eventually every group should elect a single operational

and stable leader [1, 35]. Stable leader election can be implemented in the partially synchronous model [3].

3.3 Data structures

Algorithm 1 presents the data structures used by processes in RamCast. Every server process has a shared buffer M per client c, where slot $M[c, i]$ contains the i-th message msg multicast by client c, the groups dst the message is addressed to, and an address vector ptr, where $ptr[g, p] = j$ means that at process p in group g message msg is stored in slot $M[c, j]$. The address vector is used by a process to know where to write in the memory of another process addressed by the message. Servers compute the message's timestamp tmp, based on the timestamps proposed by the leader of each destination group, and the acknowledgements from the members of the leader's group, stored in vector ack. A multicast message state $stat$ can be null ($\perp$), pending a final timestamp (MCAST), assigned a final timestamp (ORDERED) or delivered (DONE).

To compute a message's timestamp, each server process has a protected buffer T, where the i-th slot $T[c, i]$ matches the corresponding slot $M[c, i]$ in the shared buffer M associated with c. The slot contains a timestamp vector tmp and a round vector rnd, each one with an entry per group g: $tmp[g]$ contains the timestamp proposed by the current leader in g in round $rnd[g]$. Timestamps and rounds are tuples $\langle cnt, pid\rangle$, where cnt is a scalar and pid is a process identifier, unique across the system. It follows that $\langle cnt, pid\rangle > \langle cnt', pid'\rangle$ iff $cnt > cnt'$, or $cnt = cnt'$ and $pid > pid'$. We further assume that $time(\langle cnt, pid\rangle) = cnt$.

To multicast a message, a client writes the message in the shared buffer of each process in the groups addressed by the message. To know in which entry the multicast message must be written, the client keeps an address vector $cptr$ with an entry per group and per process in the group.

Process p's local state includes a $clock$, used to compute logical timestamps, p's current round, used when p is the leader of the group, vector $Leader$ with p's view on the current leader of each group, and vector $Round$, where entry $Round[g]$ contains the largest round p has accepted from $Leader[g]$.

3.4 Normal execution

RamCast is optimized for the normal case, when a message is addressed to groups whose leaders are operational and stable. Algorithm 2 presents RamCast's normal execution, and Figure 3 illustrates one normal execution. We explain next the behavior of each one of the tasks in Algorithm 2.

- *Task 1.* To multicast message m, client c first calculates the next available slot in the buffer of every process addressed by m. Then, c invokes the $Relay$ procedure, which copies the message, its destination, and the address vector for m in the message buffer of every process addressed by m.
- *Task 2.* Once leader process L in group g reads m from its message buffer, it computes a group-wise timestamp for m and writes the proposed timestamp in the protected timestamp buffer of all follower processes in the leader's group g, and all leader processes of other groups in the destination of m. If a remote write is denied, then L ends the task since another process in g became leader.

Algorithm 1 Data structures

1: Each server has a shared buffer M and a protected buffer T per client c; each slot in M stores a multicast message; the corresponding slot in T stores the message's timestamps
2: Each slot $M[c, i]$ contains the following information:
3: msg: the message m multicast by client c
4: dst: destination groups m is addressed to
5: $ptr[1..k, 1..n]$: for each g in destination, slot with msg at processes in g; $null$ if g is not in the message's destination
6: tmp: the timestamp of m, initially $\langle 0, 0\rangle$
7: $ack[1..k, 1..n]$: for each g in destination, acknowledgment of timestamp in $T[c, i].tmp[g]$ from processes in g
8: $stat$: state of m: $\perp$ (initially), MCAST, ORDERED or DONE
9: Each entry $T[c, i]$ contains the following information:
10: $tmp[1..k]$: timestamp proposed by leader of group g
11: $rnd[1..k]$: the round of g's leader, initially $\langle 0, 0\rangle$
12: Each client c has vector $cptr[1..k, 1..n]$, with the next available slot in buffers M and T per group g and process p
13: Each server p at group g also has:
14: $clock$: logical timestamp counter at p, initially $\langle 0, p\rangle$
15: $round$: the round of p, when leader, initially $\langle 0, p\rangle$
16: $Leader[1..k]$: the leader at each group
17: $Round[1..k]$: last accepted round at each group

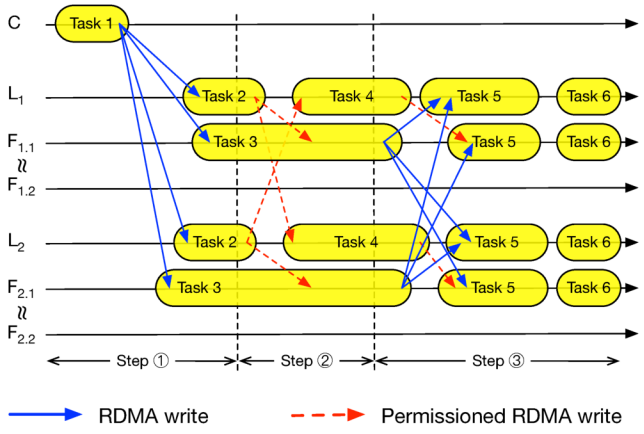

Figure 3: Normal execution of RamCast (i.e., group leaders are operational and stable). We show the steps at one follower per group only to avoid cluttering.

- *Task 3.* When a follower detects that a multicast message has been assigned a timestamp by the group leader, it updates its clock. This is done to ensure that timestamps from a group are monotonically increasing (necessary in case the process becomes leader). Then, the follower acknowledges that it has read this timestamp by writing the round used by the leader in its entry in the ack vector of every process in the destination of the message. The follower detects the leader proposed timestamp by checking whether the round of the timestamp matches the round associated with the leader. This assumes that both the timestamp and the round have been updated by the leader. We discuss in Section 4 how we ensure this with RDMA.
- *Task 4.* When the leader L in g reads a timestamp written by a leader in another group, L updates its local clock with the timestamp, to ensure that any future proposed timestmaps

Algorithm 2 Normal case (stable leader)

```
1: Client c multicasts message m to groups in m.dst as follows:
2: for each h in m.dst, for each q in h                                   {Task 1}
3:     cptr[q] ← cptr[q] + 1
4: Relay(c, m, m.dst, cptr)
5: Server p in group g executes as follows:
6: when ∃c, i : M[c, i].stat = MCAST and p = Leader[g]                    {Task 2}
7:     clock ← clock + 1
8:     for each follower q in g and each leader q in M[c, i].dst
9:         j ← M[c, i].ptr[q]
10:        write(q, T[c, j].tmp[g], ⟨clock, p⟩)
11:        write(q, T[c, j].rnd[g], round)
12:        if write denied then end task
13: when ∃c, i : M[c, i].stat = MCAST and
                T[c, i].rnd[g] = Round[g]                                 {Task 3}
14:    clock ← max(clock, time(T[c, i].tmp[g]))
15:    for each h in M[c, i].dst, for each q in h
16:        j ← M[c, i].ptr[q]
17:        write(q, M[c, j].ack[p], Round[g])
18: when ∃c, i, h : M[c, i].stat = MCAST and
                T[c, i].rnd[h] = Round[h] and h ≠ g                       {Task 4}
19:    clock ← max(clock, time(T[c, i].tmp[g]))
20:    for each follower q in g
21:        j ← M[c, i].ptr[q]
22:        write(q, T[c, j].tmp[h], T[c, i].tmp[h])
23:        if write denied then end task
24: when ∃c, i, h : M[c, i].stat = MCAST and ∃quorum Q in h:
        for each q in Q: M[c, i].ack[q] = Round[h]                        {Task 5}
25:    M[c, i].tmp ← max(M[c, i].tmp, T[c, i].tmp[h])
26:    if for each group h in M[c, i].dst: ∃quorum Q in h:
                    for each q in Q: M[c, i].ack[q] = Round[h] then
27:        M[c, i].stat ← ORDERED
28: when ∃c, i : M[c, i].stat = ORDERED and
            ∄d, j : M[d, j].stat ∈ {ORDERED, MCAST} and
            M[d, j].tmp < M[c, i].tmp                                     {Task 6}
29:    deliver m
30:    M[c, i].stat ← DONE
31: procedure Relay(c, msg, dst, ptr)
32:    for each h in dst: for each q in h
33:        write(q, M[c, ptr[q]].msg, msg)
34:        write(q, M[c, ptr[q]].dst, dst)
35:        write(q, M[c, ptr[q]].ptr, ptr)
36:        write(q, M[c, ptr[q]].stat, MCAST)
```

will be bigger, and writes the read timestamp in the memory of each one of its followers. This task is executed by the leader of a group only, since only the leader is updated with timestamps from the leader of another group (see Task 2). The reason why only the leader of a group is updated is to ensure that any timestamps assigned by the leader are consistent with any other timestamps assigned by the group.

- *Task 5.* When a message has a timestamp proposed by a leader from each destination group of the message, and a quorum of processes in each destination group agrees with the proposed timestamp, the message becomes ordered.
- *Task 6.* A process delivers an ordered message when it can assert that no other messages can be assigned a smaller timestamp.

In the normal case, a message multicast by a client to multiple groups is delivered by the leaders and the followers of the addressed groups after three RDMA write delays (see Figure 3). In Section 3.7, we discuss how messages addressed to a single group can be delivered by the group's leader after two RDMA write delays.

3.5 Handling failures

RamCast handles the failure of a leader using a mechanism similar to Paxos. As a consequence, it can tolerate multiple processes that believe to be leader in a group without violating safety. In order to ensure progress, however, eventually there should be only one operational leader process per group. When a process becomes leader, it needs to catch up with the previous leader. In the following, we describe how the newly elected leader does this. The procedure uses both shared memory and message passing for communication. In RDMA, message passing is less efficient than shared memory, but it reduces complexity, as we do not have to handle concurrent accesses to shared memory. Since failures are hopefully rare, we consider that trading performance for simplicity is acceptable.

- *Task 7.* When a process that will become the next leader of the group suspects the current leader, it determines its *first undecided slot (FUS)* per client in its shared buffers. A slot is undecided if its state is equal to MCAST. Then, the new leader chooses a round and sends a catch-up message to every server process in the system. Since slot i in the new leader's buffer may correspond to a different slot at another process, the new leader must convert its FUS into one that is meaningful for the contacted process (line 7).
- *Task 8.* A process p will consider a catch-up message from new leader q in group h if q has picked a round bigger than the current round for h at p. This is a requirement from Paxos, to ensure that a new leader will not decide on a value different than a previously decided value. If the catch-up message can be considered, then p revokes permissions to the previous leader, grants permission to the shared buffer T to q, collects all information requested by q, and sends it to q. Finally, p updates h's round and leader.
- *Task 9.* When the new leader receives responses for a catch-up request from a quorum of processes in a group, it handles each entry i for every client c received as follows. First, the process selects the response with the largest round. From Paxos, this ensures that if a timestamp has been chosen, it can only be the one with the largest round. The next steps depend on whether the process received the responses from its own group or not. If the process received the responses from its own group, then it picks the timestamp in the selected response, if any, or picks a timestamp using its own clock. In either case, the process proposes the picked timestamp to all other members of its group and the leaders of the other involved groups. If the process received the responses from another group, then it forwards the timestamp in the selected response to the followers in its group.

We also consider the case of faulty clients, who may fail to update all destinations of a multicast message.

- *Task 10.* When a process detects the failure of a client, it relays all the messages multicast by the faulty client that have not been ordered yet. This means that only messages in the MCAST state need to be relayed.

Algorithm 3 Handling failures and suspicions

```
1: when suspect Leader[g] and p is g's next leader                    {Task 7}
2:     for each c do
3:         FUS[c] ← i, where M[c, i] is the first undecided entry
4:     round ← ⟨time(round) + 1, p⟩
5:     for each h in Γ
6:         for each q in h
7:             for each c: xFUS[c] ← M[c, FUS[c]].ptr[h, q]
8:             send (CATCH_UP, xFUS, round) to q

9: when receive (CATCH_UP, FUS, round) from q in h and
                                        round > Round[h]              {Task 8}
10:    revoke previous permissions and grant permission to q
11:    pend ← ∅
12:    for each c do
13:        let j be the last entry in M such that M[c, j] ≠⊥
14:        for i in FUS[c]..j do
15:            if h ∈ M[c, i].dst then
16:                pend ← pend ∪ (c, i, M[c, i].msg, M[c, i].dst,
                                  M[c, i].ptr, T[c, i].tmp[g], T[c, i].rnd[g])
17:    send (MY_STATE, pend) to q
18:    Round[h] ← round
19:    Leader[h] ← q

20: when receive (MY_STATE, pend) from quorum Q in h,
        including p's response if g = h                               {Task 9}
21:    bag ← union of all received pend from h
22:    let maxts be the largest timestamp tmp in bag
23:    clock ← max(clock, time(maxts))
24:    for each (c, i, –, –, –, –) in bag
25:        let (c, i, msg, dst, ptr, tmp, rnd) in bag be such that
                                  ∄(c, i, –, –, –, rnd′) in bag and rnd′ > rnd
26:        if g = h then
27:            if rnd > 0 then
28:                t ← tmp
29:            else
30:                clock ← clock + 1
31:                t ← ⟨clock, g⟩
32:            for each q in g and each leader q in dst
33:                write(q, T[c, ptr[q]].tmp[g], t)
34:                write(q, T[c, ptr[q]].rnd[g], round)
35:                if write denied then end task
36:        else
37:            for each q in g
38:                write(q, T[c, ptr[q]].tmp[h], tmp)
39:                if write denied then end task

40: when suspect client c                                             {Task 10}
41:    for each i such that M[c, i].stat = MCAST
42:        Relay(c, M[c, i].msg, M[c, i].dst, M[c, i].ptr)
```

3.6 Correctness

In this section, we argue that RamCast implements atomic multicast, as defined in §2.2.

Proposition 1. *(Uniform integrity) For any message m, every process p delivers m at most once, and only if p is a destination of m and m was previously multicast.*

Proof: Process p delivers m at Task 6 if m's state is ORDERED. After delivering m, p sets m's state to DONE, and thus m cannot be delivered more than once.

Let c be the client that multicasts m to groups in dst, and let p be in group g. From Task 6, p only delivers m if it is in p's M buffer and m's state is ORDERED. Message m's state is set to ORDERED in Task 5 if its current state is MCAST. A message's state is set to MCAST in procedure *Relay*, which is invoked in two cases: (a) by client c upon multicasting m (Task 1) to groups in dst, in which case $g \in dst$; or (b) by some process q that suspects c (Task 10), has m in its buffer in state MCAST, and g is a destination of m. In case (b), m was written in q's buffer either (b.1) directly by c or (b.2) indirectly by some other process. In any case, there is some process r such that m is included in r's buffer by c. It follows from Task 1 that p is a destination of m and m was multicast by client c. □

Lemma 1. *If all correct processes in the destination of an atomically multicast message m have m in their M buffer in the MCAST state, then they eventually set m to the ORDERED state.*

Proof: Let m be addressed to groups in dst and q be a correct process addressed by m. We claim that for each $h \in dst$, q will have a timestamp for h that is acknowledged by a quorum of processes in h. By the leader election oracle and the fact that each group has a majority of correct processes, group h eventually has a stable correct leader l. Either (a) l executes Task 2 and proposes its clock value as h's timestamp or (b) l executes Task 7 to replace a suspected leader. In (b), l sends a CATCH_UP message to all processes and will receive for each group $g \in dst$ the timestamp proposed in g, if any, and the corresponding acknowledgements from processes in g (Task 8). For the case where $h = g$, l will pick the timestamp decided by a previous leader or choose one if no timestamp has been decided (Task 9). Thus, in both cases (a) and (b), the leader writes the chosen timestamp in the M buffer of each process in h and in the leaders of other groups in dst. From Task 3, every follower in h will acknowledge this timestamp in the buffer of each process in the destination of m. From Task 4, when l has a timestamp from $g \neq h$, l writes the timestamp in the buffer of its followers, which concludes the claim. Therefore, eventually q has a timestamp for every group in dst, can compute m's final timestamp, and set m's state as ORDERED.

Lemma 2. *If a correct process p has an atomically multicast message m in its M buffer in the ORDERED state, p eventually delivers m.*

Proof: Assume for a contradiction that q does not deliver m. Thus, there is some message m' in the buffer such that $m \neq m'$, m''s timestamp is smaller than m's timestamp, and m''s state is not DONE.

We first show that any message added in the buffer after m becomes ORDERED has a timestamp bigger than m's timestamp. Message m only becomes ordered after it has timestamps from all groups in m's destinations dst. When q reads a timestamp x for m from some group in dst, q updates its clock such that it contains the maximum between its current value and x. Since the next event that q handles for a message m'' will increment its clock, it follows that m'' will have a timestamp bigger than x.

We now show that every message that contains a timestamp smaller than m's final timestamp ts is eventually delivered and its state set to DONE. To see why, let m' be the message with the smallest timestamp in the buffer. Thus, such a message is eventually delivered and its state set to ORDERED. Eventually, m will be the message in the buffer with smallest timestamp and therefore delivered, a contradiction. We conclude then that q eventually delivers m. □

Proposition 2. *(Validity) If a correct client c multicasts a message m, then eventually every correct process p in m's destination dst delivers m.*

Proof: Upon multicasting m, c relays m to groups in dst (see Task 1). The Relay procedure then copies m to the M buffer of every correct process p in groups in dst and sets its state to MCAST. From Lemma 1, it follows that every correct process p set m's state to ORDERED. From Lemma 2, p eventually delivers m. □

Proposition 3. *(Uniform agreement) If a process p delivers a message m, then eventually all correct processes q in m's destination dst deliver m.*

Proof: For process p to deliver m, from Task 6, p has a timestamp for every group h in dst in the M buffer such that ts is the largest among these timestamps. Moreover, there is no message m' in the buffer such that $m \neq m'$, $ts < y$, where y is a timestamp assigned to m', and m' is not ordered.

We first show by contradiction that q eventually has m in its M buffer. Let c be the client that multicasts m. If c is correct then, c writes m in q's buffer, so consider that c fails before it can write m in q's buffer. Since p delivers m, it has a quorum of acknowledges from each group in dst. Any quorum includes at least one correct process, which from Task 10, eventually suspects c and relays m to all processes in dst, including q, a contradiction.

It follows from Lemma 1 that q eventually sets the state of m to ORDERED in its buffer, and from Lemma 2 that q eventually delivers m. □

Proposition 4. *(Uniform prefix order) For any two messages m and m′ and any two processes p and q such that {p, q} ⊆ dst ∩ dst′, where dst and dst′ are the groups addressed by m and m′, respectively, if p delivers m and q delivers m′, then either p delivers m′ before m or q delivers m before m′.*

Proof: The proposition trivially holds if p and q are in the same group, so assume p is in group g and q is in group h and suppose, by way of contradiction, that p does not deliver m' before m nor does q deliver m before m'. Without loss of generality, suppose that m's timestamp ts is smaller than m''s timestamp ts'.

We claim that q inserts m into the M buffer before delivering m'. In order for m (respectively, m') to be delivered by p (resp., q), p's (resp., q's) M buffer must contain a timestamp ts_g from group g and ts_h from group h (resp., ts'_g from group g and ts'_h from group h).

From Task 2 (or Task 9 if some process has suspected the leader), the leader l in group g must have included the timestamp ts_g for message m and ts'_g for message m' in p's M buffer and both timestamps have been acknowledged by a quorum of processes in group g. Assume that the leader l has written ts_g before ts'_g to the M buffer of every follower in group g and the leader l_h in group h. From Task 2, we have $ts_g < ts'_g$. Therefore, from Task 4, l_h will write to the M buffer of every follower in group h, including q, both ts_g for message m and ts'_g for message m'.

Consequently, from the claim, q delivers m before m' since $m.ts < m'.ts$, a contradiction that concludes the proof. □

Proposition 5. *(Uniform acyclic order) Let relation < be defined such that m < m′ iff there exists a process that delivers m before m′. The relation < is acyclic.*

Proof: Suppose, by way of contradiction, that there exist messages $m_1, ..., m_k$ such that $m_1 < m_2 < ... < m_k < m_1$. From Task 6, processes deliver messages following the order of their final timestamps. Thus, there must be processes p and q such that the final timestamps they assign to m_1, ts_p and ts_q, satisfy $ts_p < ts_q$, a contradiction since both p and q have the same timestamps for each group in dst in Task 6. □

Theorem 1. *RamCast implements atomic multicast.*

Proof: This follows directly from Propositions 1 through 5. □

3.7 Extensions

We now discuss how to speed up the execution of messages multicast to a single group of processes and how to reuse entries in the client buffers (i.e., essentially, how to turn the data structures into circular buffers).

Since only one process at a time can hold permission to write in the timestamp buffer of processes, if a leader manages to write its proposed timestamp for a multicast message (Task 2) in a quorum of processes, it knows that the timestamp proposed has been accepted by the followers and can change the message's state to ORDERED. Thus, at the leader the message is ready to be delivered without the acknowledgements from the followers. We use this optimization to speed up the delivery of single-group messages at the leader.

A client can recycle a buffer slot when the slot will not be needed by any processes. This is the case when all message destinations have delivered the message (i.e., message state is DONE). Therefore, periodically, all message destinations inform the client about the slot with their *Last Delivered Message (LDM)*. The client then computes the *Last Stable Group Message (LSGM)* as the lowest LDM received in the group. The client can safely update the pointer to the tail of its buffer to the LSGM. This procedure, although simple, requires feedback from all processes in a group. To tolerate failures, processes must checkpoint their state. When $f + 1$ processes in a group have checkpointed a state that includes the i-th slot, then the group's LSGM can be updated to i.

4 IMPLEMENTATION

We implemented a prototype of RamCast in Java using jVerbs (DiSNI library) version 2.1,[1] an open-source user-level networking library developed by IBM that supports RDMA communication [45]. jVerbs offers low latencies to applications running inside a Java Virtual Machine by exposing RDMA network hardware resources directly to the JVM. The source code of RamCast is publicly available.[2]

In RamCast, we applied a number of optimizations to further decrease latency and improve performance. When establishing the connections between hosts, we use two-sided operations (e.g., send and receive) to exchange memory addresses, and use the one-sided

[1] https://github.com/zrlio/disni
[2] https://github.com/longle255/libRamcastV3

writes for data transfer. As the two-sided operation is only used for control information at the start up and in the case of failures, this procedure does not affect performance of normal execution. The one-sided operation for the actual transfer makes the overall data transfer efficient. In RDMA, writes and sends with payloads below a limit specified by devices may be written to the work request (WR) as inlined data, thus the RNIC does not need to fetch that payload via a DMA read. In RamCast, we inline all writes whose payload is lower than the inline limit (i.e., 64 bytes) [40].

Normally, the RNICs actively poll a completion event (CE) from the CQ to ensure a write resides in remote memory. Polling CE is time consuming as it involves synchronization between the RNICs on both sides of a CQ [46]. Thus, for multi-group messages, we employ *selective signaling* [32] to reduce this overhead by only checking for a CE after pushing a number of writes. When using selectively signaled writes with requests of size n, up to $n - 1$ consecutive operations can be unsignaled, i.e., a CE will not be pushed for these operations. Note that if an operation ended with an error (e.g., a leader's write permission is revoked), it will generate a CE even if it was supposed to use unsignaled completion.

In a shared memory context, when a process reads entries that are updated by another process, it is important that the reader process does not read incomplete data that has not been fully updated by the writer process, (e.g., processes in RamCast continually monitor their shared buffer for new messages and may be reading an incomplete entry). We resolve this issue by adding an extra canary value at the end of each entry, as used in previous works [2, 18, 28, 32, 46]. Before writing a message to a remote host, a process in RamCast adds the checksum of the entry to the end of the entry. A remote process always first checks the checksum value and waits for the checksum to match the entry.

5 EXPERIMENTAL EVALUATION

In this section, we discuss the evaluation rationale (§5.1), describe the experimental environment (§5.2), and present the three sets of experiments we conducted (§5.3–5.5).

5.1 Evaluation rationale

We conducted three sets of experiments. In the first set (§5.3), we seek to understand the effects of message size on RamCast's performance. In the second set (§5.4), we compare RamCast's performance to WBCast's, an efficient message-passing atomic multicast protocol. As we will see, RamCast largely outperforms WBCast in both throughput and latency. Even though both protocols are assessed in the same environment, one may wonder whether RamCast's advantage is a result of RDMA's efficient write operations (used by RamCast) when compared to message-passing operations (used by WBCast). In the third set of experiments (§5.5), we compare RamCast's "inherent performance" (i.e., in the absence of contention and queueing effects) to high-performance atomic broadcast protocols that rely on RDMA technology (APUS and Mu) or bypass the network stack (Kernel Paxos).

In the following, we briefly comment on these protocols and their configuration in the experimental study. We provide more details about each protocol in Section 6.

White-Box Atomic Multicast (WBCast) [23] is a genuine atomic multicast protocol that delivers exceptional performance, thanks to some algorithmic optimizations. WBCast provides a C-language implementation that uses libevent for communication.[3] We extended the code to include additional statistics information. We include WBCast in our evaluation because it is currently the best-performing message-passing atomic multicast protocol.

APUS is a general-purpose atomic broadcast protocol that implements Paxos. As part of the execution, nodes store ordered messages on stable storage (e.g., SSD). In order to ensure a fair comparison among the various protocols, which store messages in main memory only, we configured APUS with a RAM disk storage instead.

Mu [1] implements Protected Memory Paxos. It was designed to replicate micro services and optimizes atomic broadcast in one important aspect: by co-locating clients and the Paxos's leader on the same host. As a consequence, a broadcast message can be ordered after one RDMA write delay (i.e., done by the leader to place the message in the memory of the followers). As described in Section 3.1.2, this is enough to ensure that the message is ordered. Unfortunately, co-locating clients and leaders on the same host is not possible in atomic multicast: the motivation and scalability of atomic multicast stem from the fact that one can create multiple groups, each one operating independently. We consider Mu in our evaluation since it is the best-performing RDMA-based atomic broadcast protocol.

Kernel Paxos [21] is a Multi-Paxos implementation that improves the performance of the original libpaxos library.[4] The main idea is to reduce system calls by running Paxos logic in the Linux kernel, bypassing the network stack, and avoiding the TCP/IP stack. We used the original code[5] and deployed a single group with three replicas. We compare RamCast to Kernel Paxos because both systems avoid the overhead of the communication stack.

5.2 Environment and configuration

We conducted all experiments in CloudLab [19] with two sets of nodes: (a) R320 nodes, equipped with one eight-core Xeon E5-2450 processor running at 2.1GHz, 16 GB of main memory, and a Mellanox FDR CX3 NIC; and (b) XL170 nodes, equipped with one ten-core Intel E5-2640v4 processor running at 2.4GHz, 64 GB of main memory, and a Mellanox ConnectX-4 NIC. A 10 Gbps network link with around 0.1ms round-trip time connects all nodes running Ubuntu Linux 18.04 with kernel 4.15 an Oracle Java SE Runtime Environment 11. In all experiments, clients and servers are independent processes. Clients submit requests in a closed-loop, that is, a client multicasts a message to servers and waits for a response before multicasting the next message. In all RamCast experiments, clients measure latency as the interval between the multicast of a message and the response received from the first server in each group addressed by the message. In all protocols, each group has 3 processes with in-memory storage.

[3] https://github.com/imdea-software/atomic-multicast
[4] https://bitbucket.org/sciasciad/libpaxos
[5] https://github.com/esposem/Kernel_Paxos

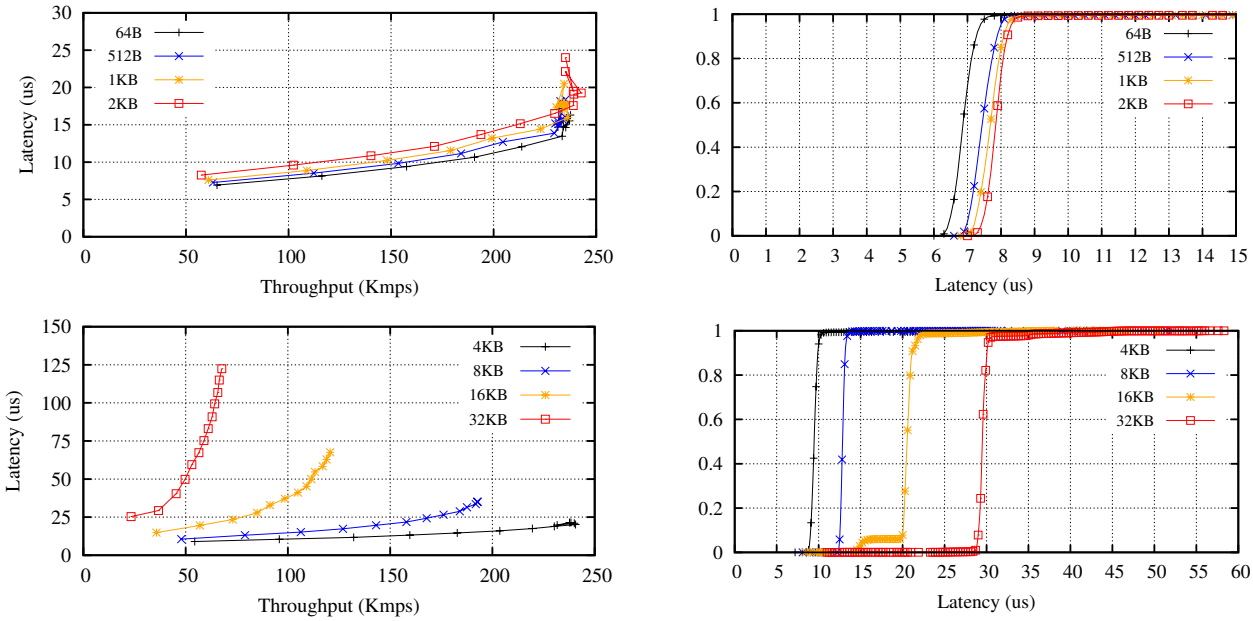

Figure 4: RamCast performance with different message sizes: 64B to 2 KB (top) and 4 KB to 32 KB (bottom), throughput versus latency (left) and latency cumulative distribution function for a single client (right).

5.3 The impact of message size

In this experiment, conducted on XL170 nodes, we measure RamCast throughput and latency for different message sizes. For each message size, we increase the number of clients until the system is saturated (i.e., throughput increases minimally with the number of clients). Figure 4 (left) shows that up to 4KB messages, the impact of message size on the system throughput is negligible, with nearly 250 thousand messages delivered per second. As the message size increases past 4KB, the maximum throughput decreases with 70 thousand messages per second for 32KB messages. The latency cumulative distribution function (CDF) in Figure 4 (right) exhibits minimum latency variation for messages with up to 2KB, around 8 microseconds at 95^{th} percentile. At 4KB messages, the latency slightly goes up to around 10 microseconds.

5.4 The performance of atomic multicast

The next set of experiments assess RamCast behavior in scenarios with up to 8 groups of 3 replicas each, deployed on XL170 nodes. The first experiment comprises executions in which clients multicast single-group 64-byte messages in setups with 1, 2, 4, and 8 groups. Figure 5 (top left) shows the aggregated throughput results when the system is saturated. The results show that the throughput of both RamCast and WBCast grow linearly with the number of groups for single-group messages. RamCast outperforms WBCast, however, by a factor of 3.6× in all configurations. Since groups do not exchange any information when dealing with single-group messages, the latency CDF is similar for all configurations, no matter the number of groups in the system, as depicted in Figure 5 (middle and bottom left). RamCast's efficient single-group multicast (see §3.7) together with RDMA's high-performance writes grant RamCast a 28× median latency advantage to WBCast (i.e., ~7 us against ~200 us).

The next experiment evaluates the protocols with multi-group messages of 64 bytes addressed to all the groups. This is the most stressful case for a genuine atomic multicast protocol, since to order a multicast message, all groups addressed by the message must interact. Therefore, the more groups addressed by a message, the lower the expected performance. RamCast's maximum throughput is greater than WBCast's in every configuration with 233, 145, 80, and 40 thousand messages per second for 1, 2, 4, and 8 destination groups against 63, 50, 35, and 27 thousand for WBCast, as shown in Figure 5 (top right). The values correspond to improvements of 3.7×, 2.9×, 2.3× and 1.5×, respectively.

The difference is more expressive when we consider the latency for a single client, i.e., when both protocols are contention-free. Figure 5 (middle right) shows that the latency CDF for RamCast with values of 8, 46, 78 and 150 microseconds for 1, 2, 4, and 8 destination groups if we consider the 95^{th} percentile. The equivalent values for WBCast, as depicted in Figure 5 (bottom right), are 214, 445, 673, and 1055 microseconds, representing 20× to 7× slower delivery times when compared to RamCast's.

5.5 RamCast's inherent performance

We now compare RamCast to atomic broadcast protocols using a single group of three replicas, and 64-byte and 1-kilobyte messages, on R320 nodes. Figure 6 shows a similar trend for both message sizes. Mu's co-location of clients and leader on the same host (with the resulting single RDMA write delay) significantly pays off: 4.8× and 3.1× reduction in the median latency with respect to RamCast

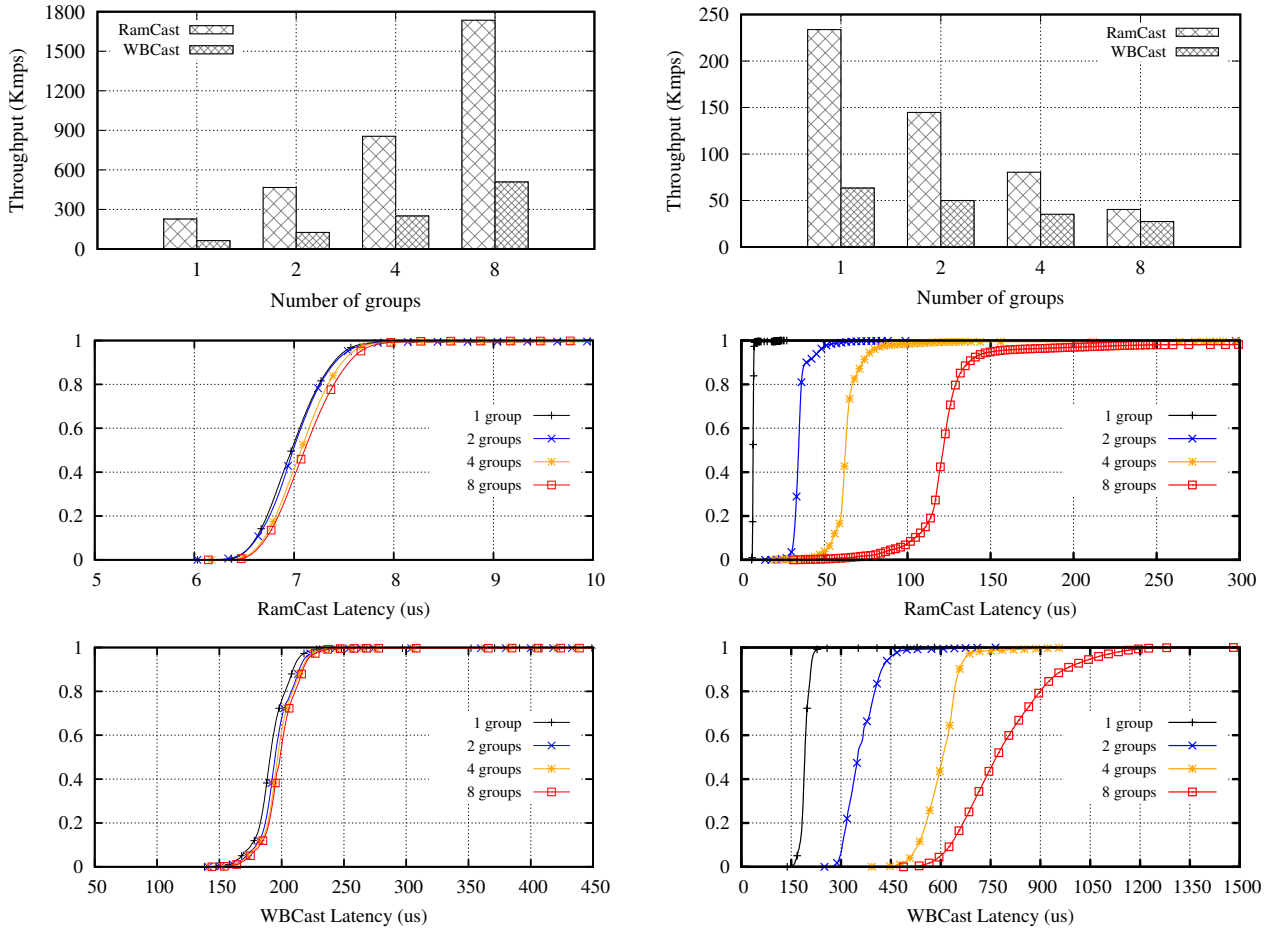

Figure 5: Performance of atomic multicast when messages are multicast to a single group (graphs on the left) and to all groups (graphs on the right). In each case, we show: throughput (top) and latency cumulative distribution function with one client for RamCast (middle) and WBCast (bottom).

for 64-byte and 1-kilobyte messages, respectively. However, co-locating the clients and the leaders on the same host hampers atomic multicast scalability (see §5.1). When compared to APUS, RamCast reduces the median latency by 4.7× and 5.6×, with messages with 64-byte and 1-kilobyte messages, respectively. Compared to to Kernel Paxos, the improvements are in the range of 4.4× and 4.7×.

6 RELATED WORK

RamCast is at the intersection of atomic multicast protocols (§6.1), RDMA-based systems (§6.2), and RDMA-based consensus protocols (§6.3).

6.1 Atomic multicast

Atomic multicast is a well-studied problem. Skeen's algorithm (described in §3.1) is possibly the first atomic multicast algorithm. Even though it is not fault-tolerant, it is genuine: processes only communicate if they are in the destinations of the messages. Later timestamp-based genuine atomic multicast algorithms implemented fault-tolerant versions of Skeen's protocol. FastCast [12] speeds up the delivery of messages by overleaping some parts of the protocol (i.e., the order proposed by the leader and the consensus needed to decide on the proposed order). In good runs, FastCast delivers multi-group messages in 4 communication steps. White-Box Atomic Multicast [23] further improves latency with a protocol that combines Paxos and a fault-tolerant version of Skeen's protocol. White-Box Atomic Multicast delivers multi-group messages in 3 communication steps at the leaders of the involved groups and 4 communication steps at the followers. RamCast improves on White-Box Atomic Multicast in that both leaders and followers can deliver a multi-group message in 3 communication steps.

Ring-based protocols [7, 17, 39] proposed a different approach to high throughput by propagating messages along a predefined ring overlay and ensuring atomic multicast properties by relying on this topology. However, ring-based algorithms are non-genuine: involved processes communicate with processes outside the destination groups to deliver messages. The time complexity of these algorithms is proportional to the number of destination groups.

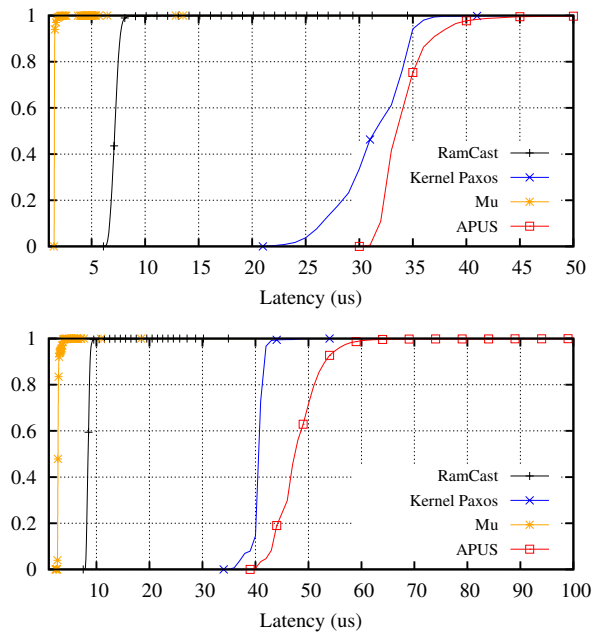

Figure 6: Latency cumulative distribution function for RamCast and atomic broadcast protocols with a single client: 64-byte messages (top) and 1K-byte messages (bottom).

6.2 RDMA systems

Remote Direct Memory Access (RDMA) [33] is an interface that allows servers to read and write the memory of a remote server directly. Over the years, RDMA has become an active area of research for its high throughput, low latency, and low CPU overhead. RDMA techniques have been implemented in various architectures, including Infiniband [41], RoCE [6], and iWRAP [44]. RDMA has already been explored and applied in a variety of applications, from key/value stores [18, 32, 40, 47], to databases [8, 27], and distributed file systems [28, 36, 48]. Pilaf [40] is a distributed in-memory key-value store that implements client-lookup operations with one-sided RDMA reads. In contrast, in HERD [32], clients use one-sided RDMA writes to send requests to servers which poll their receive RDMA buffers to process requests. FaRM [18] proposes a distributed computing platform, which provides the transactional interface for applications to access the shared memory. NVFS [28] provides a novel design of HDFS with byte-addressable NVM and RDMA network. Octopus [38] is a distributed, shared persistent memory file system that combines RDMA and NVM's new features by redesigning the software. Besides, many optimization guidelines were proposed by Kalia et al. [33] to enhance performance of RDMA system. We have applied many of the mentioned best practices in the implementation of RamCast.

6.3 RDMA-based consensus

RDMA has received limited attention in the context of consensus protocols, and only a few crash-tolerant replication protocols based on RDMA have been proposed. DARE [42] aims to optimize for low latency in replica communication. The consensus leader in DARE replicates requests to its follower with RDMA one-sided read/write operations, and makes use of permissions when changing leaders. APUS [46] improves upon DARE. APUS combines RDMA with Paxos and focuses on scalability with the number of connections and replicas. APUS is based on intercepting inbound socket calls, so it does not require modifying applications for integration. Derecho is a library for structuring applications into subgroups and shards with support for SMR within them. Updates occur with a variation of Paxos, while queries exploit a new form o snapshot isolation. The dynamic membership tracking uses virtual synchrony. Even though Derecho organizes processes into subgroups and shards, there is no abstraction that provides total order for update operations involving multiple subgroups and shards. Mu [2] exploits Protected Memory Paxos and colocates the client with the leader of Paxos to reach low latency. Similarly to Mu, RamCast also relies on RDMA's protected memory to order single-group messages efficiently. However, RamCast cannot colocate clients and leaders since it implements atomic multicast. This happens because clients may multicast a message to different groups and colocating all leaders in the same host defeats the purpose of atomic multicast.

6.4 Hardware-accelerated consensus

In addition to RDMA, consensus and atomic broadcast have been also accelerated using other hardware artifacts, such as FPGAs and programmable switches (e.g., [14–16, 29, 30, 37, 43]). RamCast implements atomic multicast, a more general abstraction than atomic broadcast (§2.2). An interesting open question is whether atomic multicast could also benefit from FPGAs and programmable switches.

7 CONCLUSION

Atomic multicast is a fundamental communication abstraction in the design of scalable and highly available strongly consistent distributed systems. This paper presents RamCast, the first genuine atomic multicast protocol tailor-made for the shared-memory model. In addition to introducing a novel algorithm that leverages the permission mechanism of RDMA's write operations to reduce the number of communication steps, we also have implemented and evaluated the protocol under a large range of parameters. The results show that RamCast outperforms a state-of-the-art message-passing genuine atomic multicast protocol and atomic broadcast protocols that optimize communication and rely on comparable assumptions.

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for the constructive feedback. This work was partially supported by the Swiss National Science Foundation (project number 175717).

REFERENCES

[1] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. Marathe, and I. Zablotchi. 2019. The Impact of RDMA on Agreement. In *PODC*.
[2] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygkis, and I. Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *OSDI*.
[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. 2001. Stable Leader Election. In *DISC*.
[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. 2007. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *SOSP*.

[5] P. A. Barret, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Verissimo, L. Rodrigues, and N. A. Speirs. 1990. The Delta-4 extra performance architecture (XPA). In *FTCS*.

[6] M. Beck and M. Kagan. 2011. Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure. In *DC-CaVES*.

[7] C. E. Bezerra, D. Cason, and F. Pedone. 2015. Ridge: high-throughput, low-latency atomic multicast. In *SRDS*. IEEE, 256–265.

[8] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. 2015. The end of slow networks: It's time for a redesign. *arXiv preprint arXiv:1504.01048* (2015).

[9] K. P. Birman and T. A. Joseph. 1987. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)* 5, 1 (1987), 47–76.

[10] K. P. Birman and T. A. Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems* 5, 1 (Feb. 1987), 47–76.

[11] T. D. Chandra and S. Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (1996), 225–267.

[12] P. R. Coelho, N. Schiper, and F. Pedone. 2017. Fast Atomic Multicast. In *DSN*.

[13] J. C. Corbett, J. Dean, and M. et al Epstein. 2012. Spanner: Google's globally distributed database. In *OSDI*.

[14] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Trans. Netw.* 28, 4 (Aug. 2020), 1726âĂŞ1738.

[15] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. 2016. Paxos Made Switch-y. 46, 2 (May 2016), 18–24.

[16] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. 2015. NetPaxos: Consensus at Network Speed. 1–7.

[17] C. Delporte-Gallet and H. Fauconnier. 2000. Fault-Tolerant Genuine Atomic Multicast to Multiple Groups.. In *OPODIS*. Citeseer.

[18] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*.

[19] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. 2019. The Design and Operation of CloudLab. In *USENIX-ATC*.

[20] C. Dwork, N. Lynch, and L. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323.

[21] E. Giuseppe Esposito, P. R. Coelho, and F. Pedone. 2018. Kernel paxos. In *SRDS*. IEEE.

[22] M. J. Fischer, N. A. Lynch, and M. S. Patterson. 1985. Impossibility of Distributed Consensus with one Faulty Process. *J. ACM* 32, 2 (1985), 374–382.

[23] A. Gotsman, A. Lefort, and G. Chockler. 2019. White-Box Atomic Multicast. In *DSN*.

[24] R. Guerraoui and A. Schiper. 2001. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.* 254, 1-2 (2001), 297–316.

[25] V. Hadzilacos and S. Toueg. 1993. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, Sape J. Mullender (Ed.). Addison-Wesley, Chapter 5, 97–145.

[26] L. Hoang Le, E. Fynn, M. Eslahi-Kelorazi, R. Soulé, and F. Pedone. 2019. DynaStar: Optimized Dynamic Partitioning for Scalable State Machine Replication. In *ICDCS*.

[27] B. Huang, L. Jin, Z. Lu, M. Yan, J. Wu, P. CK Hung, and Q. Tang. 2019. RDMA-driven MongoDB: An approach of RDMA enhanced NoSQL paradigm for large-Scale data processing. *Information Sciences* 502 (2019), 376–393.

[28] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. 2012. High performance RDMA-based design of HDFS over InfiniBand. In *SC*. IEEE.

[29] Z. István, D. Sidler, G. Alonso, and M. Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. 425–438.

[30] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. 35–49.

[31] F. P. Junqueira, B. C. Reed, and M. Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *DSN*.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen. 2014. Using RDMA efficiently for key-value services. In *SIGCOMM*.

[33] A. Kalia, M. Kaminsky, and D. G. Andersen. 2016. Design Guidelines for High Performance {RDMA} Systems. In *USENIX-ATC*.

[34] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[35] L. Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[36] B. Li, P. Zhang, Z. Huo, and D. Meng. 2009. Early experiences with write-write design of NFS over RDMA. In *NAS*. IEEE.

[37] J. Li, E. Michael, N. Kr. Sharma, A. Szekeres, and D. R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. 467–483.

[38] Y. Lu, J. Shu, Y. Chen, and T. Li. 2017. Octopus: an rdma-enabled distributed persistent memory file system. In *USENIX-ATC*.

[39] P. J. Marandi, M. Primi, and F. Pedone. 2012. Multi-ring paxos. In *DSN*. IEEE.

[40] C. Mitchell, Y. Geng, and J. Li. 2013. Using One-Sided {RDMA} Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX-ATC*.

[41] G. F. Pfister. 2001. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O* 42, 617-632 (2001), 102.

[42] M. Poke and T. Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *HPDC*.

[43] D. R. K. Ports, J. Li, V. Liu, N. Kr. Sharma, and A. Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. 43–57.

[44] M. J. Rashti and A. Afsahi. 2007. 10-Gigabit iWARP Ethernet: comparative performance analysis with InfiniBand and Myrinet-10G. In *IPDPS*. IEEE, 1–8.

[45] P. Stuedi, B. Metzler, and A. Trivedi. 2013. jVerbs: ultra-low latency for data center applications. In *SoCC*.

[46] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. 2017. APUS: Fast and Scalable Paxos on RDMA. In *SoCC*.

[47] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*.

[48] J. Wu, P. Wyckoff, and D. Panda. 2003. PVFS over InfiniBand: Design and performance evaluation. In *ICPP*. IEEE.