Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Practical atomic multicast: a trade-off between genuineness and performance

Tesis de Licenciatura en Ciencias de la Computación

Patricio Ezequiel Inzaghi Pronesti

Advisor: Professor Fernando Pedone (Università della Svizzera italiana)

Co-advisor: Professor Paulo R. Coelho (Universidade Federal de Uberlândia)

Buenos Aires, 2019

ABSTRACT (ESPAÑOL)

Los servicios de Internet actuales tienen requerimientos de disponibilidad y escalabilidad elevados. Para cumplir estándares de alta disponibilidad estos deben permanecer funcionales a pesar de producirse fallas en sus nodos o enfrentar caídas completas de datacenters. La escalabilidad permite incrementar la performance agregando al sistema más componentes y con eso lograr soportar un incremento de la carga. El protocolo *atomic multicast* es una pieza fundamental en este tipo de servicios. Vamos a considerar sistemas donde los procesos pueden organizarse en grupos y los clientes envían mensajes destinados a un subconjunto de ellos. Algunos protocolos consiguen ordenar los mensajes usando un grupo fijo de procesos o involucrando a todos ellos, sin importar a quién está destinado el mensaje. Durante mucho tiempo se creyó que, para ser eficiente, un algoritmo de *atomic multicast* debe ser genuino: solo el emisor y los procesos involucrados en su destino deben comunicarse para propagar y ordenar un mensaje. Esta tesis vuelve a evaluar esta propiedad de los protocolos de *atomic multicast* y experimenta con la hipótesis de que, relajandola, se puede obtener un mejor throughput o latencia en distintas topologías de red. Se presentan dos enfoques: BaseCast, un algoritmo de *atomic multicast* genuino, y TreeCast, uno parcialmente genuino que escala con respecto al número de grupos para mensajes enviados a un solo grupo.

Palabras claves: Sistemas Distribuidos, *Atomic multicast*, Máquinas de estados replicadas, Escalabilidad, Autenticidad

ABSTRACT

Modern online services have strict availability and scalability requirements. Highly available services remain operational despite node crashes and datacenter disasters. Scalable services can increase performance by adding system components and thereby accommodate increased load. Atomic multicast is a communication building block for this type of services. We are interested in systems where processes are organized in different groups and clients can send messages addressed to a subset of them. Some atomic multicast protocols address this challenge by ordering all messages using a fixed group of processes or involving all groups, regardless of the destination of the messages. It has long been believed that to be efficient, an atomic multicast algorithm must be genuine: only the message sender and destination processes should communicate to propagate and order a multicast message. The thesis revisits the genuineness of atomic multicast protocols and tests whether relaxing genuineness can lead to better throughput or latency in different network topologies. The thesis presents two approaches: BaseCast, a genuine atomic multicast algorithm, and TreeCast, a partially genuine atomic multicast that scales with the number of groups for messages addressed to a single group.

Keywords: Distributed Systems, Atomic multicast, State Machine Replication, Scaling, Genuineness

AGRADECIMIENTOS

Debo agradecer a la Università della Svizzera italiana por aceptarme en su *Master's Research Scholarship* y permitirme realizar la tesis de Licenciatura allí. Agradezco a mis directores de tesis Fernando Pedone y Paulo R. Coelho por guiarme y dar su tiempo a este trabajo.

Al Departamento de Computación por ser un lugar excepcional dentro de la Universidad de Buenos Aires donde pude desarrollarme como alumno, docente y divulgador. Al Laboratorio de Robótica y Sistemas Embebidos por darme un lugar donde poder aprender y comenzar a entender lo que es una carrera científica.

A Victoria, Ariel, Elisa, Julian, Marcelino, Augusto, Matias, mis compañeros de la facultad, con los que recorrí este (largo) camino. Las miles de horas de estudio y frustraciones hubiesen sido insoportables sin su compañía.

A mis amigos Gonzalo, Cristian y Ricardo, que me ayudaron a distenderme y olvidarme un poco de la facultad.

A mi mamá que me supo inculcar la perseverancia y el trabajo, a mi papá la curiosidad y la locura, todas características importantes para una carrera científica. A mi hermana por alegrarme con su sentido del humor tan Inzaghi.

Quiero agradecerles profundamente a Delia, Hugo, Jessica, Carlos, Kaethe, Nelson, Roxana, Gonzalo, Naiara, Ayelen, Erik, Alejandro, Silvia y por supuesto a la pequeña Anastasia, por incluirme desde siempre como parte de su familia. Son parte mi corazón desde hace tiempo.

A mi familia universitaria: Augusto, Matias y Solange. Poco a poco nuestras reuniones fueron siendo cada vez menos para estudiar y más para ayudarnos mutuamente.

Finalmente agradecerle a Solange, el amor de mi vida, gracias por darme todo tu cariño y soportarme con mi falta de paciencia y complicado humor. Sigamos teniendo muchas más aventuras juntos.

Para Solange, mi compañera de vida.

CONTENTS

1. Introduction 1

2. Background 3
   2.1 Consensus 3
   2.2 Model and definitions 3
   2.3 Atomic multicast 4
   2.4 Related work 5

3. Algorithms solving atomic multicast 7
   3.1 BaseCast 7
   3.2 TreeCast 10
   3.3 URingPaxos 11
      3.3.1 Paxos consensus 11
      3.3.2 Algorithm 12

4. Implementation 15
   4.1 AmcastNode 15
      4.1.1 BaseCast 16
      4.1.2 TreeCast 16
   4.2 Communication 17
   4.3 Experiments 17
   4.4 Configuration 17

5. Performance evaluation 19
   5.1 Infrastructure 19
   5.2 Experiments 19
   5.3 Results 22
      5.3.1 LAN environment 22
      5.3.2 WAN_1 environment 24
      5.3.3 WAN_2 environment 26

6. Conclusions 31
   6.1 Discussion 31
   6.2 Future work 31

1. INTRODUCTION

Many modern applications need to handle a large amount of requests from clients, so they need to be scalable and highly available. Such applications usually solve this by replicating servers near clients, looking for a decrease of latency and increasing tolerance to failures if one or more of the servers is faulty. In some contexts a weak consistency system could be successful, on this kind of models the system does not guarantee that subsecuent accesses will return the updated value of an object. This solution is not adequate for every application because clients may observe a non-intuitive application behavior. Strong consistency (e.g., linearizability [17]) systems are more intuitive to the users because they always show a single image of the application's state. However, such systems are harder to maintain since we need to order all the user requests across the complete system to achieve this.

Atomic multicast [16] is a building block that allows messages to be propagated to groups of processes with reliability and order guarantees. The goal of this important primitive is to provide a way to send a message to all non-faulty processes in the destination of a message, and make them agree on the relative order of delivery. To obtain scalability in this kind of systems we need to provide an atomic multicast that is **genuine** [16], this means that only the participants involved in a message's destination colaborate to agree on its delivery.

State machine replication is a well-established approach to fault tolerance [19, 26]. The idea is that by executing service requests deterministically in the same order, correct replicas will transition through the same sequence of state changes and produce the same output for every request. Atomic broadcast can be used to guarantee that replicas deliver requests in the same order.

State machine replication provides linearizability, a consistency criteria. A system is linearizable if it satisfies the following requirements [5]: (i) It respects the real-time ordering of requests across all clients. There exists a real-time order among any two requests if one request finishes at a client before the other request starts at a client. (ii) It respects the semantics of the requests as defined in their sequential specification.

With state machine replication, every server has a full copy of the service state. Several approaches have proposed to shard the service state and handle each shard as a replicated state machine (e.g., [10, 4, 21]). Atomic multicast is a natural abstraction to order requests in a sharded replicated system (e.g., [10, 18]). Requests that can be entirely executed within a shard are multicast to the required shard; requests that involve data in multiple shards must be consistently multicast to all target shards.

This master thesis is about the implementation and evaluation of two different atomic multicast protocols in different contexts. It contributes deriving **TreeCast**, a crash-failure version of ByzCast [9] (a Byzantine Fault-Tolerant atomic multicast protocol) that is partially genuine and scales with the number of groups for messages addressed to a single group. TreeCast is a hierarchical protocol, it uses an overlay tree where a node in the

tree is a group of processes. Two different structures of this tree will be compared to have an insight about the performance of this protocol. We say that this protocol is partially genuine in that messages atomically multicast to a single group of processes only require coordination between the message sender and the destination group; messages addressed to multiple groups of processes, however, may involve processes that are not part of the destination (i.e., these processes help order the messages though).

Another contribution of this work is the implementation from scratch of **BaseCast** [5, 15, 25] a genuine atomic multicast algorithm that scale with the number of groups. The code base, which is written in Java, is simple to understand and extensible. Both protocols made use of URingPaxos[1], a high throughput atomic multicast protocol also implemented in Java. The code support other ways of solving consensus, a key part of the algorithm, so it is prepared for future optimizations and experiments if needed.

Both protocols were compared against each other in two main environments: a *local-area network configuration* with very low delays between nodes and an *emulated wide-area network (emulated WAN)* that emulates a real WAN with different regions and high delays between them. With our experiments we will test whether relaxing genuineness can lead to better throughput or latency in different network topologies. We will revisit genuineness comparing this two different approaches.

The remainder of this thesis is structured as follows. Chapter 2 provides some theoretical background about the subject. Chapter 3 introduces all the algorithms used on this work. Chapter 4 explains how the protocols are implemented in detail. Chapter 5 evaluates and compares them with experimental results and finally Chapter 6 comments on our conclusions about this work and points to future work.

[1] https://github.com/sambenz/URingPaxos

2. BACKGROUND

This chapter is intended to present the fundamental concepts of the thesis and introduce the problem we are facing.

2.1 Consensus

Consensus is a fundamental problem in distributed systems, when a number of processes requires agreement for a single data value. Some of these agents may fail so protocols solving consensus need to be fault tolerant. Usually these are designed to support a limited number of faulty processes.
A process can be either correct or faulty, and two types of failures are commonly considered: benign (i.e., crash failures) or arbitrary behavior (i.e., Byzantine failure). In this thesis we will consider systems where the first type of failure can occur.

The consensus service allows processes to propose values and ensures that eventually one of the proposed values is decided. In its basic form consensus processes only decide once, in real scenarios we will need to consensuate values continuously, we will call every one of this executions a propose instance. A process in group g proposes a value (or a set of values) x in instance i by invoking $propose_g[i](x)$, and decides on y in instance i with $decide_g[i](y)$.

In a crash-stop failure model, consensus is defined as follows [7]:

- **Termination:** Every correct process eventually decides some value.
- **Agreement:** No two correct processes decide differently.
- **Uniform integrity:** Every process decides at most once.
- **Uniform validity:** If a process decides v, then v was proposed by some process.

To make consensus solvable in each group [14], we further assume that all processes at each group must have access to a weak leader election oracle [7]. The oracle outputs a single process denoted $leader_{g,p}$ such that there is (a) a correct process l_g in g and (b) a time after which, for every p in g $leader_{g,p} = l_g$.

2.2 Model and definitions

We consider a system $\Pi = \{p_1, ..., p_n\}$ of processes. Processes communicate by exchanging messages and do not have access to a shared memory or a global clock. The system is asynchronous: messages may experience arbitrarily large (but finite) delays and there is no bound on relative process speeds. We assume that processes may fail by crashing (i.e., no malicious behavior). A process that never crashes is *correct*; otherwise it is $faulty$.

Communication links are fair-lossy, i.e., links do not create, corrupt, or duplicate messages, and guarantee that for any two correct processes p and q, and any message m, if p

sends m to q infinitely many times, then q receives m an infinite number of times.

We define $\Gamma = \{g_1, ..., g_m\}$ as the set of process groups in the system. Groups are disjoint, non-empty, and satisfy $\bigcup_{g \in \Gamma} g = \Pi$. In general, a consensus algorithm can make progress using $n = 2f+1$ participants, despite the simultaneous failure of any f of them, so each group contains $2f + 1$ processes, where f is the maximum number of faulty processes per group. Finally we will also assume the existence of a uniform consensus service in each group g.

2.3 Atomic multicast

A message m in our model contains two attributes, the *value* to be sent ($m.val$) and a set of *destination* groups $m.dst$ (e.g., a message $m = \{hello, \{2, 3\}\}$ will send the value *hello* to groups 2 and 3). A message is called *local* if $|m.dst| = 1$ or *global* if $|m.dst| > 1$. A process reliably multicasts a message m by invoking a non-uniform FIFO primitive r-multicast(m) and receives the message m with primitive r-deliver(m).

To atomic multicast a message m in the system, a process invokes the primitive a-multicast(m) and delivers m with a-deliver(m). The a-deliver primitive must not be confused with r-deliver, the first one determines that the message m is ready to be processed satisfying atomic multicast guarantees and the second one is simply the act of receiving a message from a reliable channel. We define the relation $<$ on the set of messages processes a-deliver as follows: $m < m'$ iff there exists a process that a-delivers m before m'.

Atomic multicast satisfies the following properties:

- **Validity:** If a correct process p a-multicasts a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, a-deliver m.
- **Agreement:** If a correct process p a-delivers a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, a-deliver m.
- **Integrity:** For any correct process p and any message m, p a-delivers m at most once, and only if $p \in g$, $g \in m.dst$, and m was previously sent using a-multicast.
- **Prefix order:** Let $\{g, h\} \subseteq \Gamma$. For any two messages m and m' and any two correct processes p and q such that $p \in g$, $q \in h$ and $\{g, h\} \subseteq m.dst \cap m'.dst$, if p a-delivers m and q a-delivers m', then either p a-delivers m' before m or q a-delivers m before m'.
- **Acyclic order:** The relation $<$ is acyclic.

An atomic multicast algorithm $\mathcal{A}$ is genuine if and only if for any admissible run R of $\mathcal{A}$ and for any correct process p in R, if p sends or receives a message, then some message m is a-multicast, and either (a) p is the process that a-multicasts m or (b) $p \in g$ and $g \in m.dst$. [16]

For example, lets consider a system with groups $\Gamma = \{g_1, g_2\}$. If the client *Alice* sends the messages $[\{foo_1, \{g_1\}\}, \{foo_2, \{g_1, g_2\}\}]$ and *Bob* sends $[\{var_1, \{g_2\}\}, \{var_2, \{g_1, g_2\}\}]$

we expect that g_1 and g_2 deliver the messages foo_2 and var_2 in the same relative order.

The following is a valid delivery state because messages foo_2 and var_2 respect a relative order:

g_1 receives:	$\{foo_1, \{g_1\}\}$	$\{foo_2, \{g_1, g_2\}\}$	$\{var_2, \{g_1, g_2\}\}$
g_2 receives:	$\{foo_2, \{g_1, g_2\}\}$	$\{var_1, \{g_2\}\}$	$\{var_2, \{g_1, g_2\}\}$

This is also a valid state:

g_1 receives:	$\{foo_1, \{g_1\}\}$	$\{foo_2, \{g_1, g_2\}\}$	$\{var_2, \{g_1, g_2\}\}$
g_2 receives:	$\{foo_2, \{g_1, g_2\}\}$	$\{var_2, \{g_1, g_2\}\}$	$\{var_1, \{g_2\}\}$

But this is not a valid execution, our global messages are in different orders in both groups:

g_1 receives:	$\{foo_1, \{g_1\}\}$	$\{var_2, \{g_1, g_2\}\}$	$\{foo_2, \{g_1, g_2\}\}$
g_2 receives:	$\{foo_2, \{g_1, g_2\}\}$	$\{var_2, \{g_1, g_2\}\}$	$\{var_1, \{g_2\}\}$

2.4 Related work

Several multicast and broadcast algorithms have been proposed in the literature [12]. Moreover, many systems ensure strong consistency with "ad hoc" ordering protocols that do not implement all the properties of atomic multicast (e.g., [11, 28, 10]). We focus next on atomic multicast algorithms that tolerate benign failures.

Existing atomic multicast algorithms fall into one of three categories: timestamp-based, round-based, and ring-based. Algorithms based on timestamps (i.e., [15, 24, 27]) are all genuine and can be considered variations of an early atomic multicast algorithm [5], designed for failure-free systems. In these algorithms, processes assign timestamps to messages, ensure that destinations agree on the final timestamp assigned to each message, and deliver messages following this timestamp order. The precise way in which these properties are ensured varies from one algorithm to another. The algorithms in [15, 27] have a best-case time complexity of 6δ (being δ the communication delay) for the delivery of global messages. The algorithm in [24] can deliver global messages in 5δ and it ensures another property besides genuineness called message-minimality. This property states that the messages of the algorithm have a size proportional to the number of destination groups of the multicast message, and not to the total number of processes. BaseCast verify this property, and although TreeCast is not genuine with respect to global messages, it satisfies this property for local messages which can be delivered as fast as the underlying atomic broadcast algorithm.

In round-based algorithms, processes execute an unbounded sequence of rounds and agree on messages delivered at the end of each round. A round-based non genuine atomic multicast algorithm that can deliver messages in 4δ is presented in [27].

Ring-based algorithms propagate messages along a predefined ring overlay and ensure atomic multicast properties by relying on this topology. An atomic multicast algorithm in

this category is proposed in [13], where consensus is run among the members of each group. The time complexity of this algorithm is proportional to the number of destination groups.

Multi-Ring Paxos [23], Spread [1, 2], and Ridge [3] are ring-based non-genuine atomic multicast protocols. On the one hand, to deliver a message m, they require communication with processes outside of the destination groups of m. On the other hand, these protocols do not require disjoint groups.

3. ALGORITHMS SOLVING ATOMIC MULTICAST

In this work we present two algorithms that solve atomic multicast: BaseCast, a genuine atomic multicast algorithm and TreeCast, a partially genuine atomic multicast that scales with the number of groups for messages addressed to a single group .

BaseCast requires six communication delays to order global messages and is the base of FastCast [8], a novel optimistic algorithm atomic multicast algorithm with reduced number of communication delays for ordering the same type of messages.

TreeCast is inspired in ByzCast [9], a Byzantine Fault-Tolerant atomic multicast protocol. We propose a simplified version of this algorithm supporting only benign failures and using a consensus service for this same model.

3.1 BaseCast

In BaseCast, each process implements a logical clock [19] and assigns timestamps to messages based on the logical clock. The correctness of BaseCast stems from two basic properties: (i) processes in the destination of an a-multicast message first assign tentative timestamps to the message and eventually agree on the message's final timestamp; and (ii) processes a-deliver messages according to their final timestamp. Every atomically multicast global message is delivered in at least 6 δ, where δ is the communication delay of the system [8].

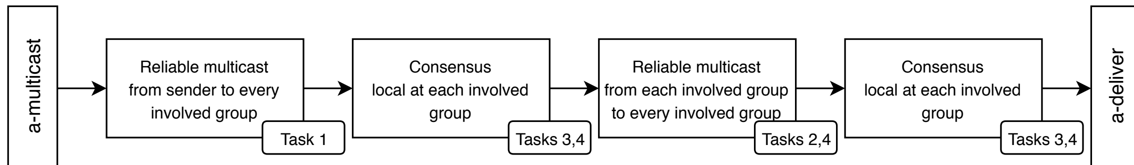

Fig. 3.1: Steps of the BaseCast protocol

BaseCast contains five tasks that execute in isolation (Fig. 3.1) and are described in Algorithm 1. It contains six variables: C_H implements a process's logical clock, used to assign hard tentative timestamps to messages; $\mathcal{B}$ contains timestamps assigned to messages not yet a-delivered; k_p and k_d index consensus instances; and sequences $ToOrder$ and $Ordered$ are used to totally order messages among processes in a group. Let S and R be two sequences. $S \oplus R$ denotes S followed by R and $S \setminus R$ denotes S without the entries that exist in R.

BaseCast defines four main meta-messages types: $START$, $SET\text{-}HARD$, $SEND\text{-}HARD$ and $SYNC\text{-}HARD$. It uses consensus within each group to ensure that processes in the same group evolve through the same sequence of state changes and produce the same outputs. Consensus is needed within a group to order $START$ messages (we call this consensus a $SET\text{-}HARD$ step) and $SEND\text{-}HARD$ messages (we call this consen-

sus a $SYNC\text{-}HARD$ step). Ordering $SET\text{-}HARD$ and $SYNC\text{-}HARD$ events within a group ensures that processes in the group assign the same hard tentative timestamp to an a-multicast message m and update their logical clock in the same deterministic way upon handling hard tentative timestamps from m's destination groups. We refer to the propagation of the $START$ message followed by the $SET\text{-}HARD$ step and the propagation of $SEND\text{-}HARD$ messages as the first phase of the algorithm, and to the $SYNC\text{-}HARD$ step as the second phase of the algorithm. The meta-message syntax is defined as follow: $\langle type, group, timestamp, message\rangle$. We will denote an undefined field of the meta-message with $\bot$.

To a-multicast a message m, process p in group g r-multicasts m to m's destinations using a $\langle \text{START}, \bot, \bot, m\rangle$ message. When p r-delivers message $\langle START, \bot, \bot, m\rangle$ in Task 1 (respectively, $\langle SEND\text{-}HARD, h, x, m\rangle$ in Task 2), p adds $\langle SET\text{-}HARD, g, \bot, m\rangle$ (respectively, $\langle SYNC\text{-}HARD, h, x, m\rangle$) to $ToOrder$ to be ordered by consensus in Tasks 3 and 4. Consensus instances are independent and can execute concurrently. However, decision events are handled sequentially according to the order determined by k_d in Task 4.

Process p handles a $\langle \text{SET-}HARD, \bot, \bot, m\rangle$ tuple in Task 4 by choosing a tentative hard timestamp for m, given by C_H , and propagating the chosen timestamp to m's destinations using a $\langle SEND\text{-}HARD, g, C_H, m\rangle$ message, if m is global; if m is local, p adds the chosen timestamp to $\mathcal{B}$ as a $\langle SYNC\text{-}HARD, g, C_H, m\rangle$ tuple.

Process p handles a $\langle SYNC\text{-}HARD, h, x, m\rangle$ tuple by updating its local clock C_H and including the tuple in $\mathcal{B}$.
In Task 5, if p has received tentative timestamps from all groups in m's destinations (i.e., SYNC-HARD tuples in $\mathcal{B}$), p determines m's final timestamp as the maximum timestamp among the tentative timestamps assigned to m by m's destinations. Process p a-delivers m when it ascertains that no a-multicast message m_0 will have a final timestamp smaller than m's. This happens when no a-multicast message m_0 rdelivered by p has (a) a final timestamp smaller than m's and (b) a tentative timestamp smaller than m's final timestamp, if p has not received tentative timestamps from all of m_0's destinations yet.

1 Initialization
2 | $C_H \leftarrow 0$;
3 | $\mathcal{B} \leftarrow \emptyset$;
4 | $k_p \leftarrow 0$; $k_d \leftarrow 0$;
5 | $ToOrder \leftarrow \epsilon$; $Ordered \leftarrow \epsilon$;

6 To a-multicast message m
7 | r-multicast $\langle START, \bot, \bot, m \rangle$ to $m.dst$;

8 **when** r-deliver $\langle START, \bot, \bot, m \rangle$ `/* Task 1 */`
9 | $ToOrder \leftarrow ToOrder \oplus \langle \text{SET-HARD}, g, \bot, m \rangle$;

10 **when** r-deliver ⟨SEND-HARD,h,x,m⟩ `/* Task 2 */`
11 | **if** $\forall y : \langle \textit{SYNC-HARD}, h, y, m \rangle \notin ToOrder$ **then**
12 | | $ToOrder \leftarrow ToOrder \oplus \langle \text{SYNC-HARD}, h, x, m \rangle$;

13 **when** $ToOrder \setminus Ordered \neq \emptyset$ `/* Task 3 */`
14 | $propose_g[k_p](ToOrder \setminus Ordered)$;
15 | $k_p \leftarrow k_p + 1$;

16 **when** $Decided := decide_g[k_d]$ `/* Task 4 */`
17 | **foreach** $z, h, x, m : \langle z, h, x, m \rangle \in Decided \setminus Ordered$ *in order* **do**
18 | | **if** $z =$*SET-HARD* **then**
19 | | | $C_H \leftarrow C_H + 1$;
20 | | | **if** $|m.dst| > 1$ **then**
21 | | | | $\mathcal{B} \leftarrow \mathcal{B} \cup \{\langle \text{SEND-HARD}, g, C_H, m \rangle\}$;
22 | | | | r-multicast⟨ SEND-HARD,g, C_H, m⟩ to $m.dst$;
23 | | | **else**
24 | | | | $\mathcal{B} \leftarrow \mathcal{B} \cup \{\langle \text{SYNC-HARD}, g, C_H, m \rangle\}$;
25 | | **if** $z =$*SYNC-HARD* **then**
26 | | | $\mathcal{B} \leftarrow \mathcal{B} \setminus \{\langle \text{SEND-HARD}, h, x, m \rangle\}$;
27 | | | $\mathcal{B} \leftarrow \mathcal{B} \cup \{\langle \text{SYNC-HARD}, h, x, m \rangle\}$;
28 | | $Ordered \leftarrow Ordered \oplus \langle z, h, x, m \rangle$;
29 | $k_d \leftarrow k_d + 1$;

30 **when** $\exists m \ \forall h \in m.dst \ \exists x : \langle \text{SYNC-HARD}, h, x, m \rangle \in \mathcal{B}$ `/* Task 5 */`
31 | $ts \leftarrow max(\{x : \langle \text{SYNC-HARD}, h, x, m \rangle \in \mathcal{B}\})$;
32 | **foreach** $z, h, x : \langle z, h, x, m \rangle \in \mathcal{B}$ **do** $\mathcal{B} \leftarrow \mathcal{B} \setminus \{\langle z, h, x, m \rangle\}$;
33 | $\mathcal{B} \leftarrow \mathcal{B} \cup \{\langle FINAL, \bot, ts, m \rangle\}$;
34 | **while** $\exists \langle FINAL, \bot, ts, m \rangle \in \mathcal{B} : \forall \langle z, h, x, m' \rangle \in \mathcal{B}, m \neq m' : ts < x$ **do**
35 | | a-deliver m;
36 | | $\mathcal{B} \leftarrow \mathcal{B} \setminus \{\langle FINAL, \bot, ts, m \rangle\}$;

Algorithm 1: BaseCast algorithm (for process p in group g)

3.2 TreeCast

TreeCast is a hierarchical protocol and compared to BaseCast is simpler (Algorithm 2). It uses an overlay tree where a node in the tree is a group of processes. Each group of processes runs an instance of atomic broadcast that encompasses the processes in the group. Hence, ordering messages multicast to a single group is easy enough: it suffices to use the atomic broadcast instance implemented by the destination group. Ordering messages that address multiple groups is trickier. First, it requires ordering such a message in the lowest common ancestor group of the message's destinations (in the worst case the root). Then, the message is successively ordered by the lower groups in the tree until it reaches the message's destination groups. The main invariant of TreeCast is that the lower groups in the tree preserve the order induced by the higher groups.

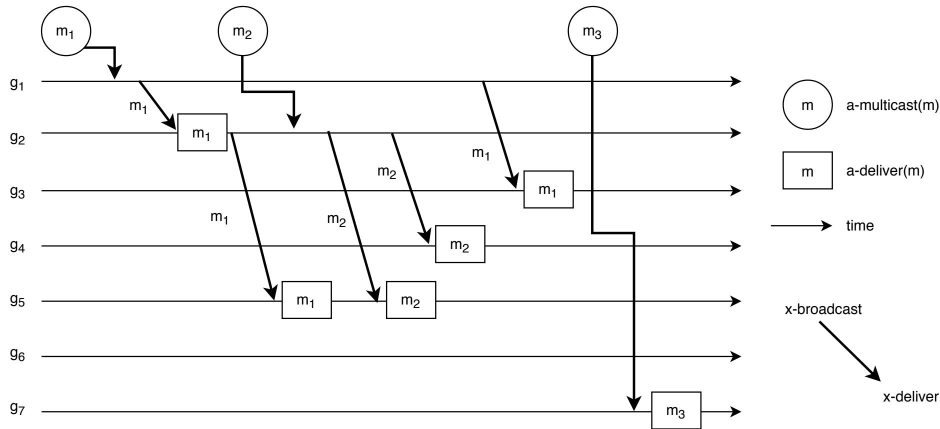

Fig. 3.2: An execution of TreeCast with three messages: m_1 is a-multicast to $\{g_2, g_3, g_5\}$, m_2 to $\{g_4, g_5\}$ and m_3 to g_7

We have the same model described in Section 2.2, with the difference that now groups can't communicate with any group freely, they must respect the hierarchy of the tree. We define the reach of a group x, $reach(x)$, as the set of target groups that can be reached from x by walking down the tree. We denote the children of a group x in the tree as $children(x)$.

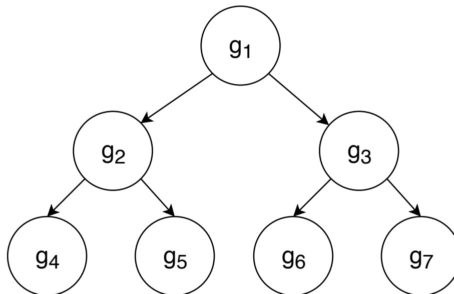

Fig. 3.3: An overlay tree with seven groups in a binary tree topology where g_1 is the root

To a-multicast a message m to a set of target groups in $m.dst$, a process first x_0-

broadcasts m in the *lowest common ancestor* group x_0 of $m.dst$, denoted $lca(m.dst)$. When m is x_k-delivered by process in x_k, each process x_{k+1}-broadcasts m in x_k's child group x_{k+1} if x_{k+1}'s reach intersects $m.dst$. This procedure continues until target groups in $m.dst$ x_k-deliver m, which triggers the a-deliver of m.

In the original algorithm (i.e., ByzCast) every process had to wait for x_k-deliver(m) $f+1$ times, this condition was removed because in our model it can't occur any Byzantine failure, only benign ones.

```
1  Initialization
2  |  T is an overlay tree with groups Γ;
3  |  A-delivered ← ∅;
4  To a-multicast message m
5  |  x0 ← lca(m.dst);
6  |  x0-broadcast(m);
7  Each server process p in group xk executes as follows:
8  |  when xk-deliver m
9  |  |  for each xk+1 ∈ children(xk) such that m.dst ∩ reach(xk+1) ≠ ∅ do
10 |  |  |  xk+1-broadcast(m);
11 |  |  if xk ∈ m.dst and m ∉ A-delivered then
12 |  |  |  a-deliver(m);
13 |  |  |  A-delivered ← A-delivered ∪ {m};
```

Algorithm 2: TreeCast algorithm

Let's consider the execution of TreeCast in Fig. 3.2 with messages m_1, m_2 and m_3 a-multicast to groups $\{g_2, g_3, g_5\}$, $\{g_4, g_5\}$ and g_7, respectively. Assuming an overlay tree $\mathcal{T}$ with seven groups described in Fig. 3.3, m_1 is first x_1-broadcast in group g_1. Upon g_1-delivering m_1, processes in g_1 atomically broadcast m_1 in g_2 and g_3. After that, g_2 x_5-broadcast in group g_5. The order between m_1 and m_2 is determined by their delivery order at g_2 since g_2 is the highest group to deliver both messages. Message m_3 is g_7-broadcast in g_7 directly since it is addressed to a single group.

3.3 URingPaxos

3.3.1 Paxos consensus

BaseCast and TreeCast use consensus to order messages inside each group, so this is a very crucial part of our implementation. Different ways to solve consensus exist, Paxos is a family of protocols for solving consensus in a network of unreliable processors/channels working in an asynchronous model that can tolerate crash failures. The Paxos protocol was first published in 1989 and named after a fictional legislative consensus system used on the Paxos island in Greece.

Paxos is a distributed and failure-tolerant consensus protocol. It was proposed by Lamport [20] and combines many properties which are required in practice. While Paxos operates in an asynchronous model and over unreliable channels, it can tolerate crash

failures. By using stable storage, it can even recover from failures. To guarantee progress, Paxos assumes a leader-election oracle.

The protocol has three roles ilustred in Fig. 3.4: proposers, acceptors and learners. In typical implementations, a single processor may play one or more roles at the same time. This does not affect the correctness of the protocol. The algorithm works as follows: In phase $1a$ a proposer sends a message with a unique number (the ballot) to all acceptors. If the acceptors never saw a higher number for this consensus instance, they confirm the reservation of the ballot by sending back a phase $1b$ message. If the proposer receives at least a quorum $\lceil (n+1)/2 \rceil$ of acceptor answers, it can start with phase $2a$. To get a quorum, a majority of acceptors must be alive. This means that Paxos requires $2f+1$ acceptor nodes to tolerate up to f failures.

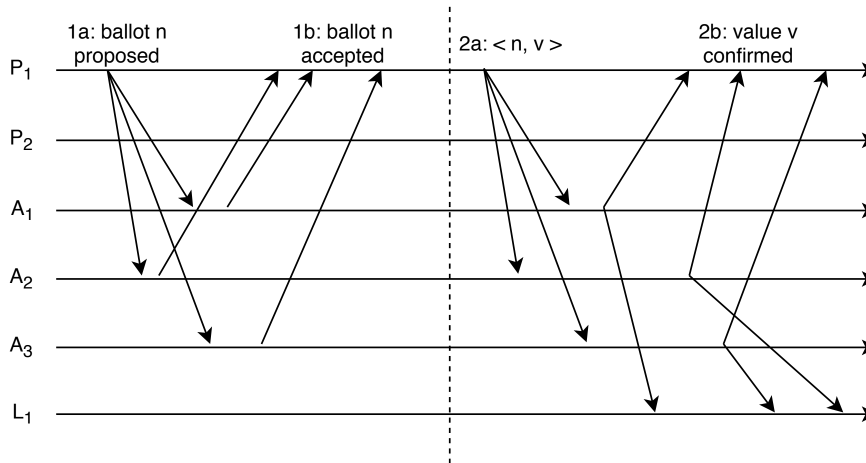

Fig. 3.4: Trivial execution of the Paxos algorithm, in which no acceptor crashes and multiple proposers do not try to reserve the same consensus instance

Phase $2a$ starts with a message, including the value to be proposed and the ballot number, from the proposer to all acceptors. If the ballot in the message corresponds to what the acceptors in phase 1 promised to accept, they will store the value. All acceptors will propagate their decision with a phase $2b$ message.

3.3.2 Algorithm

Several ways to implement Paxos exist. We decided to use URingPaxos[1], a high-throughput atomic multicast based on Ring Paxos [22]. We selected this library because we need a really optimized consensus protocol and this library has a lot of work and effort on it.

URingPaxos is implemented as a collection of coordinated Ring Paxos instances, or rings for short, such that a distinct multicast group is assigned to each ring. Each ring in turn relies on a sequence of consensus instances, implemented as an optimized version of Paxos. Similarly to Paxos, Ring Paxos differentiates processes as proposers, acceptors,

[1] https://github.com/sambenz/URingPaxos

and learners, where one of the acceptors is elected as the coordinator. All processes in Ring Paxos communicate through a unidirectional ring overlay. Using a ring topology for communication enables a balanced use of networking resources and results in high performance.

4. IMPLEMENTATION

The code is implemented under Java OpenJDK 8, a language that is widely used in real implementations, with clear abstractions and a good collection of concurrency libraries. Also, the consensus library used in this project is implemented in Java so developing it in other programming language would have led us out of the focus of this work.

The entry points of our protocol are the `AmcastClient` and the `AmcastNode` interfaces (also called replica). The decision of how to distribute the replicas in the system is a design problem out of the scope of this thesis, but the common scenario is to launch three replicas for every group in the system, which results in a fault tolerance of one node per group. The developed code is meant to provide a service to an application, so the meaning of a group depends of the real scenario. For example in a key-value database we can think of groups as a way to divide the tuples according some hash.

To atomic multicast a message m in the system one should create a new instance of `AmcastClient` and execute its public method `amulticast(byte[] data, Set<Integer> dst)`. This is a blocking method and returns when the protocol has confirmed the correct delivery of the message.

4.1 AmcastNode

The `AmcastNode` (Fig. 4.1) needs `Consensus` and `AmcastServerCommunication` implementations. These interfaces enable different ways of solving consensus within each group and of sending messages between nodes. Our project provides all the implementations required using Java blocking sockets in the `ar.uba.dc.amcast.communication.io.*` package. There is also a proxy class `URingPaxosConsensus` which communicates with the URingPaxos library described in Section 3.3. `TTYNode` is an example of usage of the `AmcastNode` and can be found in the `ar.uba.dc.amcast.lab` package.

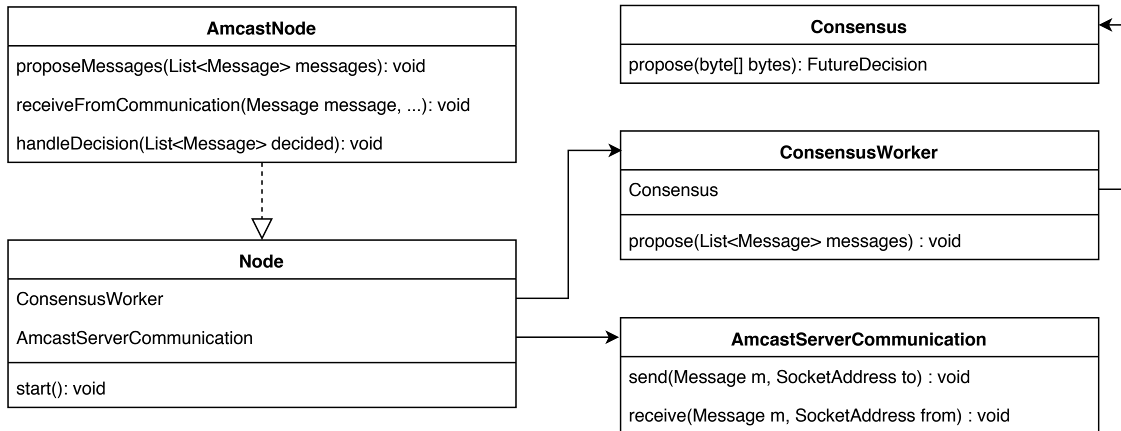

Fig. 4.1: `Node` class diagram, an implementation of `AmcastNode` interface and its related objects.

When the `start()` method is called two main threads are launched: `ConsensusWorker` and `AmcastServerCommunication`.
The first one manages the proposals and decisions needed in the protocol; `AmcastServerCommunication` (Fig. 4.2) allows the communication between replicas.
Two implementations of `AmcastNode` are provided: `BasecastNode` and `TreecastNode` (Chapter 3). Both extend the `Node` class because of their common behavior. We will discuss the details of these protocols in the next sections.

4.1.1 BaseCast

Upon message delivery the `BasecastNode` saves its content in a local cache and when all the replicas know the new value they start using a message ID to identify it. Caching messages prevents the transmission of unnecessary data between replicas and improves performance.

The main component of the `BasecastNode` implementation is the `MessageHandler` object, it manages all the tasks and message status changes described in Algorithm 1. This object uses several `Map` structures to easily obtain the current status of a given message in the protocol. Several optimizations are done: the coordinator of each group is responsible for proposing messages in the group and multicasting messages to others, which prevents unnecessary repeated proposals that would slow down the protocol. In *Task 2* of Algorithm 1 the $SYNC$-$HARD$ message of m is ordered using consensus upon receiving the first $SEND$-$HARD$ message. In the implementation we wait for the arrival of all the $SEND$-$HARD$'s messages of m to propose all the corresponding $SYNC$-$HARD$'s messages together in the group. This results in the coordinator proposing all the $SYNC$-$HARD$ messages together (i.e., only once) and reducing the amount of consensus instances.

4.1.2 TreeCast

The `TreecastNode` starts with the overlay tree configuration of the system; this permits to properly calculate the lowest common ancestor of every received message. There is no need for saving messages in cache like `BasecastNode` because the messages arrive only once to every node.
The resulting implementation of Algorithm 2 is simpler because there are no state changes in the messages. Again, the coordinator of each group is responsible for the main parts of this protocol in order to decrease the amount of messages and proposals in the system.

When a $START$ message arrives in Algorithm 2, the coordinator of the group sends it to all its involved children. This process continues recursively through the tree until all the destinations are reached. Then a bottom-up confirmation process starts: every coordinator node sends a message delivery confirmation to its parent and when the original coordinator of the message m (the lowest common ancestor) receives all the confirmations it can advertise the end of the protocol to the client.

4.2 Communication

We use TCP for message exchange between processes and our application represents messages in a classical length-prefix format: a message starts with a byte representing the

message size followed by the content, so the reader of the message can parse it properly. The implementation uses standard blocking sockets, since in a normal setup the number of concurrent connections is not too high. To improve latency and avoid message buffering, we enable the TCP nodelay option (i.e., Nagle's algorithm[1]).

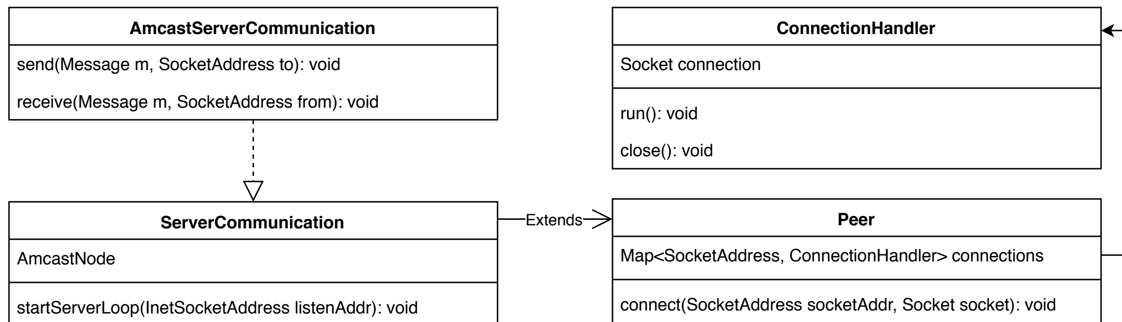

Fig. 4.2: `ServerCommunication` extends a `Peer` class, which handles the connections with other replicas and clients.

The `ServerCommunication` binds a port using the `startServerLoop()` method and waits for connections from other replicas and clients. When a connection request is received a new `ConnectionHandler` thread is created, to manage it. This allows the communication main thread (`ServerCommunication`) to be available for new connections. Every active connection is stored in the `ConcurrentHashMap<SocketAddress, ConnectionHandler>` and all the `ConnectionHandler` objects store two `LinkedBlockingQueue<Message>` for the messages to be sent and the ones received.

4.3 Experiments

The main tools used to evaluate our algorithms are located in the `ar.uba.dc.amcast.lab` package. The `ThroughputExperiment` class simulates several clients sending messages to the system. Every client that a-multicast a message must wait until its correct a-deliver to be able to send a new one, this is simulated with a synchronized message queue with a predefined startup size. If the queue size is very high the system can be saturated, our experiments showed a safe value of 150 concurrent clients constantly sending messages.

4.4 Configuration

Every replica in the system is executed with the following script:

```
JVM_PATH="-Djava.util.logging.config.file=src/main/resources/logging.properties"
JVM_PATH="$JVM_PATH -cp target/amcast-trunk-jar-with-dependencies.jar"
# GC implementation
JVM_OPTS="-XX:+UseParallelGC -Xverify:none -Xms1g"
# start the program
java $JVM_PATH $JVM_OPTS ar.uba.dc.amcast.lab.TTYNode "$@"
```

[1] https://tools.ietf.org/html/rfc896

To distribute the configuration parameters like the IP addresses of the replicas and to inform which replica is the coordinator in each group we used Apache ZooKeeper[2], a centralized service for maintaining configuration information, naming and providing group services for distributed applications.

[2] https://zookeeper.apache.org/

5. PERFORMANCE EVALUATION

5.1 Infrastructure

To perform a benchmark of the different protocols we will execute them in two main environments:
LAN configuration. We ran the experiments in a computer cluster provided by USI, the infrastructure of this cluster consists of interconnected nodes through 1 GBit/s channels. These nodes are HP SE1102 servers, with 8 GB RAM, 2x Intel Xeon L5420 2.5 GHz (Quad-core), 500 GB of hard drive of 7.2K RPM and 16 MB of buffer. Every replica is executed in different nodes and we have an extra node running ZooKeeper. The RTT's between nodes is approximately 0.1ms.
Emulated wide-area network (emulated WAN). For these experiments, we use the LAN environment and divide nodes in three "regions", R1, R2 and R3. The latencies between nodes in different regions were emulated using Linux traffic control tools. We used latency values measured in a real WAN (Fig. 5.1), with average delay of 70ms (R1 $\longleftrightarrow$ R2), 70ms (R2 $\longleftrightarrow$ R3), and 150ms (R1 $\longleftrightarrow$ R3), and standard deviation of 5%. The delay of any client to its local replica (i.e., the one on his same region) is 5ms.

Inside the *scripts* directory of the repository[1] some of the tools used on this work can be found: *ttynode.sh* runs a single replica for a given group, *tmux_launch_group.sh* launches three replicas for a given group (*cluster_tmux_launch_group.sh* does the same job but it first connects via SSH to the USI cluster). There are also two scripts for launching the experiments: *exp_latency.sh* and *exp_throughput.sh*. For simplicity we also wrote an *experiments* text file showing how to execute the experiments in section 5.2. Even though this file wasn't made for executing directly it describes how to set up the environment and reproduce them. The *tmux* scripts have tmux-xpanes[2] as a dependency.

We asume a context where nodes don't change its configuration or status very often (i.e., IP address, leader on each group) so the Apache ZooKeeper service is mainly used on startup. Using this service regularly has a high performance impact, so the configuration is stored to avoid asking for the same information.

5.2 Experiments

To compare both implementations we first need to consider two scenarios: messages with local and messages with global destinations. Both algorithms are genuine when messages are addressed to one destination (i.e., local messages), so we expect to obtain the same performance with them.

The other scenario is more complex: even though BaseCast's network topology doesn't have any restriction about communications links, thus offering a more flexible solution,

[1] https://gitlab.com/pinzaghi/amcast
[2] https://github.com/greymd/tmux-xpanes

this approach increases the amount of messages needed between the involved groups to order the client requests as the number of destinations grows.

On the other hand, when we know how the destinations of messages are distributed we can do something better: take advantage of the hierarchy of TreeCast to organize the groups intelligently to avoid involving all of them in the message ordering. If we lose this knowledge, things change dramatically, we can have messages addressed only to a few groups but ordered by almost all the groups in the system.

Let's consider TreeCast with an overlay tree as described in Fig. 3.3. A message addressed to g_4 and g_5 will involve g_2 because it is their lowest common ancestor, paying an extra overhead of communication delays for ordering and delivering the message. Now with the same tree and a message addressed to g_5 and g_6 our new LCA is g_1. This would result in the worst performance because TreeCast involves five groups for ordering the message where BaseCast only needs two.

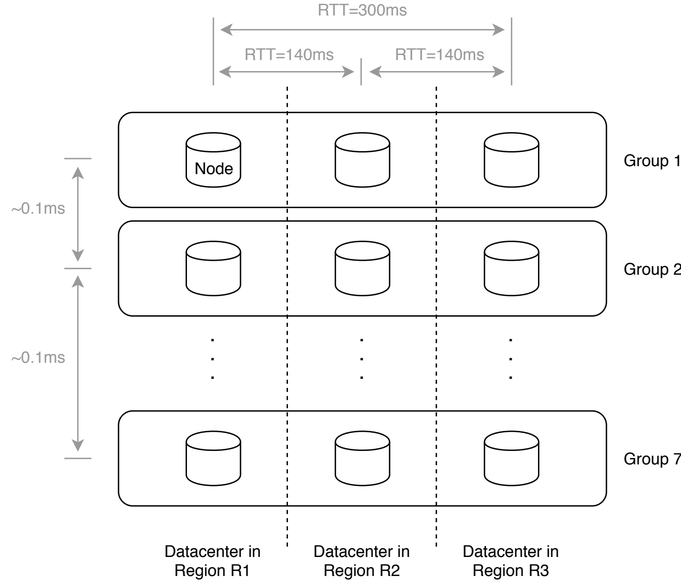

Fig. 5.1: Environment WAN_1 where for every region there is one replica of each group. This configuration supports the failure of a whole region.

Our next step is to formalize this intuition to be able to compare both algorithms in the discussed scenarios. Let $\mathcal{D} \subseteq \mathcal{P}(\{g_1, g_2, ..., g_7\})$ be the set of destinations that are used in a given experiment (i.e., $\mathcal{P}(S)$ is the powerset of set S). In an experiment, when a process multicasts a message, it randomly chooses one element from $\mathcal{D}$ and multicasts a message to this destination.

Let $\mathcal{T}$ be the tree described in Fig. 3.3 and MDH be the maximum destination height of $\mathcal{D}$ with respect to a given tree. For example, if $MDH_{\mathcal{T}} = 1$ then $\mathcal{D}$ contains only singletons (e.g., $\{g_1\}$, $\{g_2\}$, ... $\{g_7\}$). Let's call this $\mathcal{D}_1$. $\mathcal{D}_2$ is the subset of $\mathcal{P}(\{g_1, g_2, ..., g_7\})$ such that it contains destination of height 2. For example, $\mathcal{D}_2 = \{\{g_1\}, ...\{g_2, g_3\}, \{g_2, g_4\}, \{g_4, g_5\}, ...\}$, But $\mathcal{D}_2$ doesn't contain $\{g_5, g_6\}$ and $\{g_4, g_7\}$, for

example. Finally $\mathcal{D}_3 = \mathcal{P}(\{g_1, g_2, ..., g_7\})$.

A very important variable in TreeCast is the initial overlay tree, for which with any amount of groups there is only one posible tree of minimal height (e.g., Fig. 5.2). This tree is interesting because messages are ordered by the only parent and distributed to its children in one communication step. We would expect to have lower delays compared to other trees with greater height. With this configuration any global message will result in the same LCA: the root of the tree. We want to evaluate a configuration with more possible LCA's and see if we can distribute the load between them. We have two extreme cases, a degenerated tree and a balanced one. The first one doesn't take advantage of parallelization because messages need to wait the finish of every consensus instance on each group it crosses. The second one does not suffer from this problem and has the advantage of being smaller in height, a property that we want to keep.

With this taken into account, we will compare three different configurations, all with the same amount of groups running in the system: a) Configuration $\mathcal{C}_1$: TreeCast with g_1 as root and six groups $g_2, ..., g_7$ as children of g_1 (Fig. 5.2). b) Configuration $\mathcal{C}_2$: TreeCast as described in Fig. 3.3. c) $\mathcal{C}_3$: BaseCast running all the seven groups. The goal is to compare throughput and latency of the three configurations $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ running experiments using $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$ as possible message destinations. We will look for scenarios where one configuration overcomes the others depending on the amount and type of destinations.

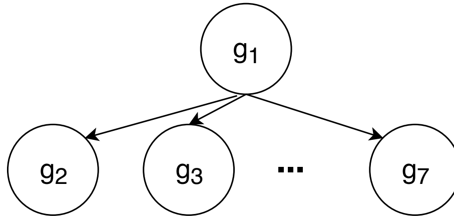

Fig. 5.2: TreeCast in $\mathcal{C}_1$: Seven groups with g_1 as root and the only parent of all other groups

All our experiments were performed in three environments: a LAN, (i.e., a network with small delays between nodes) and in two different configurations of the emulated WAN (i.e., applications where the nodes are geographically distributed and divided into several regions)

The first WAN environment considered contains one participant of each group in the same datacenter with small delays of a LAN (Fig. 5.1). An experimentation in this kind of scenario can give more information about the advantages or disadvantages of the protocols in realistic situations. We will call this environment WAN_1.

Another way to distribute the nodes across the system is assigning every complete group in the same region, we will call it WAN_2. This can make sense when groups contain information related to its own region and we want to decrease the latency between clients and that information. We will asume clients live in region R2, so any message send to groups in that region has a delay of 5ms, but contacting other regions implies delays as indicated in Fig. 5.3.

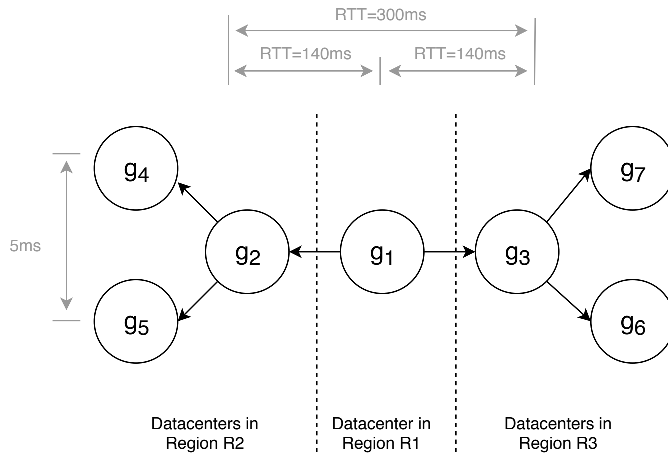

Fig. 5.3: Environment WAN_2 running configuration $\mathcal{C}_2$ where every group is completly located in of the three different regions. We asume clients near regions R2 with a latency of 5ms and every group is located in a LAN.

5.3 Results

This section presents the experimental results for three scenarios: a LAN (Section 5.3.1), a WAN with one participant of each group in the same datacenter (Section 5.3.2) and a WAN where every group is completely contained in the same region (Section 5.3.3). We will discuss the throughput and latency performance of the configurations $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ in these environments when different types of messages are sent to the system.

5.3.1 LAN environment

The first experiment compares how our three configurations $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ respond to messages with an increasing amount of destinations. In the throughput experiment we launched 150 clients concurrently sending messages to random destinations according to each scenario. For example in the 1 Group scenario every message's destination is selected from the set $\{g_1, g_2, ..., g_7\}$ resulting in local messages. In the 4 groups scenario every message's destination is selected from a set of sets of length 4 (e.g., $\{\{g_1, g_2, g_3, g_4\}, \{g_1, g_2, g_3, g_5\}, ...\}$). Fig. 5.4 shows the throughput in messages per second versus the number of groups and Fig. 5.5 shows the latency in milliseconds versus the number of groups.

In the Fig. 5.4 we can observe that in a local message scenario (1 group) the three configurations behave similarly. This responds to the fact that all are genuine protocols with local messages and do a similar procedure of only one consensus instance to finish the protocol. We can also observe the fast degradation of performance in the global message scenarios. $\mathcal{C}_3$ goes from almost 55.000 msg/sec in local messages to a little less than 14.000 msg/sec when $|m.dst| = 2$. This happens because in the first one we only need to execute one consensus instance, but when more groups are involved more communication interchange between nodes and consensus instances occur.

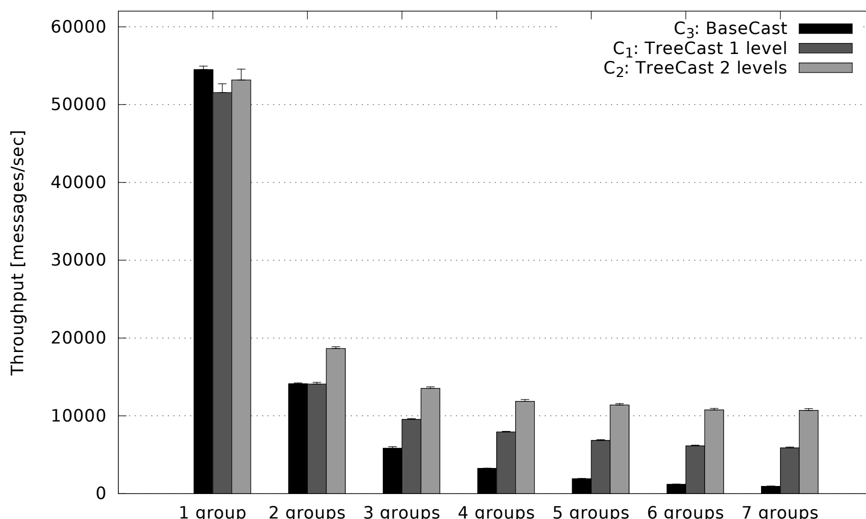

Fig. 5.4: Atomic multicast running in a computer cluster provided by USI (LAN). Throughput when 150 clients send messages to random destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval

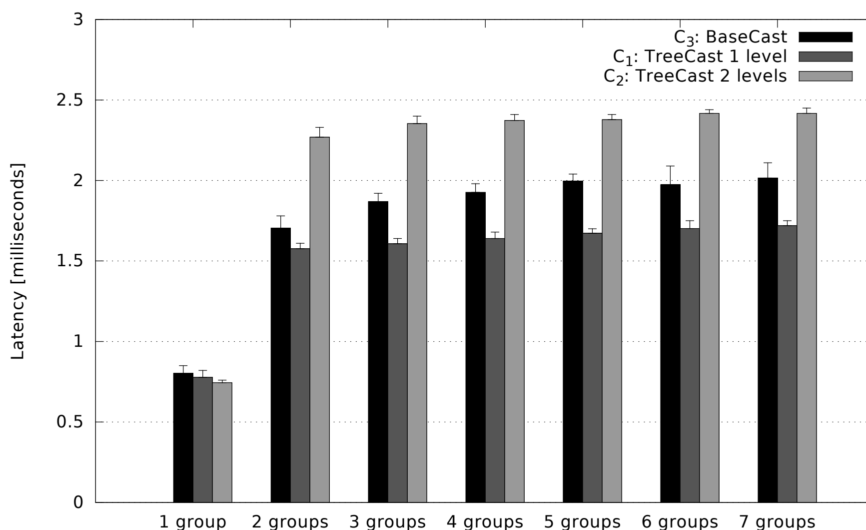

Fig. 5.5: Atomic multicast running in a computer cluster provided by USI (LAN). Latency when one client send messages to random destinations of increasing size. Bars show median latency and whiskers show 90-th percentile

The performance of $\mathcal{C}_3$ decreases notably with the number of destinations, when more groups are involved every one of them needs to synchronize timestamps with the others. This doesn't occur in $\mathcal{C}_1$ and $\mathcal{C}_2$ where only parent nodes are responsible for ordering the messages and then communicate that order to their children. This results in only paying a high price for passing from local to global messages but after that the performance decrease is negligible.

$\mathcal{C}_2$ overcomes $\mathcal{C}_1$ by almost doubling the throughput when we address messages to 7

groups. This happens because in $\mathcal{C}_1$ the group g_1 is the only parent group and it has the responsability for sending all the messages to six groups, lowering the global performance. On the other hand, in $\mathcal{C}_2$ the responsability is divided in three groups g_1, g_2 and g_3.
Fig. 5.5 shows something similar in regards to local and global messages. The latency doubles when going from 1 group to 2 groups but then the increment payed for every new group is negligible. $\mathcal{C}_2$ has a greater latency than the other configurations because the consensus instances required to order the messages are concurrent only in groups that are in the same level, but all the lower groups need to wait until their parents finish. In $\mathcal{C}_1$ we have six over seven groups that are in the same level, resulting in a tree with a small height, therefore the delay impact is low.

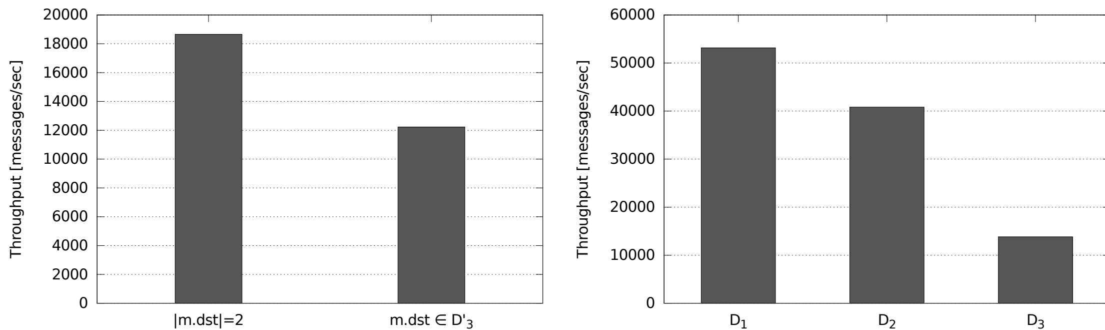

Fig. 5.6: Performance impact of TreeCast in configuration $\mathcal{C}_2$. (Left) Clients sending arbitrary messages with $|m.dst| = 2$ versus messages with $m.dst \in \mathcal{D}'_3$. (Right) Clients sending messages with $m.dst \in \mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$. Bars show average throughput.

In TreeCast, particularly in $\mathcal{C}_2$, the amount of groups addressed in a message is not the only variable when we analise performance. If we define $\mathcal{D}'_3 := \{d : d \in \mathcal{D}_3 \setminus \mathcal{D}_2 \wedge |d| = 2\}$ we can observe in Fig. 5.6 (Left) that sending messages to destinations in $\mathcal{D}'_3$ (e.g., $\{g_5, g_6\}$, $\{g_4, g_7\}$,...) results in maximum overlay trees and a throughput decrease of 30% approximately compared to arbitrary destinations of the same size.

We also measured the throughput of $\mathcal{C}_2$ when we increase the maximum possible height of the overlay tree, in other words, we launched 150 clients that send messages to random destinations in $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$ respectively. Fig. 5.6 (Right) shows that $\mathcal{D}_2$ mantains a good relationship between performance and height of the overlay tree, but in $\mathcal{D}_3$ we obtain similar results to the ones we got in the experiments showed in Fig. 5.4 for global messages.

5.3.2 WAN$_1$ environment

The following section shows the results of executing the same set of experiments seen in Section 5.3.1 but now in our emulated WAN environment. The purpose of this is to evaluate how our protocols respond to more realistic scenarios with different and high/low delays between nodes.

The results show that, compared to the LAN environment, the protocols behave in a similar way. The throughput and delay impact is high because we changed from delays

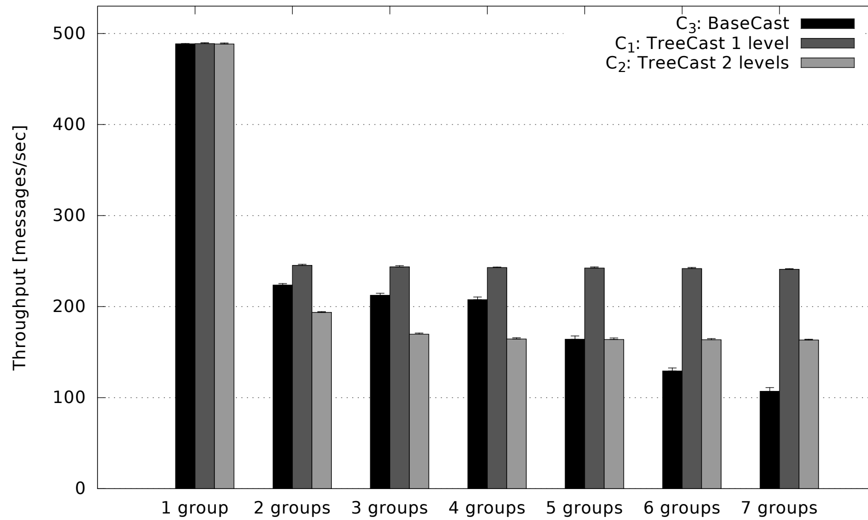

Fig. 5.7: Atomic multicast running in environment WAN_1. Throughput when 150 clients send messages to random destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval

of 0.1ms to ones several orders of magnitude higher. $\mathcal{C}_1$ throughput performance in the global message scenario is a little less than a half compared to the local message scenario. In $\mathcal{C}_3$ and $\mathcal{C}_2$ we have a similar situation but the impact is higher when we approach global messages for 7 groups. The performance of WAN_1 in the global message scenario is better compared to the LAN environment, where we only obtained approximately one fifth of the performance from local to global messages for 2 groups and it goes worst with more groups.

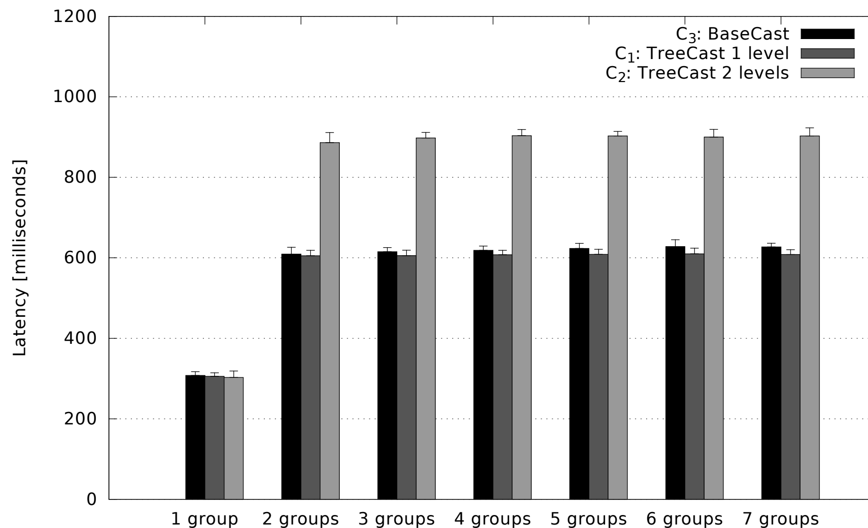

Fig. 5.8: Atomic multicast running in environment WAN_1. Latency when one client send messages to random destinations of increasing size. Bars show median latency and whiskers show 90-th percentile

If we analyze the latency (Fig. 5.8) we can observe the same behavior as in the LAN environment, considering again the new scale of values. We obtained 1 RTT in all configurations for local messages, 2 RTT's in $\mathcal{C}_1$ and $\mathcal{C}_3$ and 3 RTT's in $\mathcal{C}_2$ for global messages.

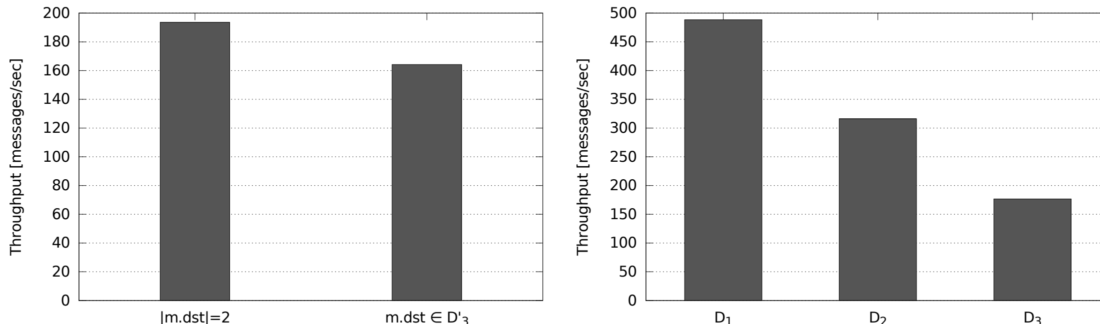

Fig. 5.9: Atomic multicast running in environment WAN_1. Performance impact of TreeCast in configuration $\mathcal{C}_2$. (Left) Clients sending arbitrary messages with $|m.dst| = 2$ versus messages with $m.dst \in \mathcal{D}'_3$. (Right) Clients sending messages with $m.dst \in \mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$. Bars show average throughput.

In Fig. 5.9 (Right) we can observe that the performance of messages sent to $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$ reduced in the same way as in the LAN environment. Again the degradation is caused by augmenting the overlay trees. In Fig. 5.9 (Left) we see a lower impact of throughput from messages with destinations of size 2 to messages in $\mathcal{D}'_3$: here the high delays don't affect the perfomance because of the low amount of destinations, allowing for a similar outcome in both scenarios.

5.3.3 WAN_2 environment

In this environment we add a new asumption to our hypotheses: we consider an application where groups are closely related to their region. This could make sense in distributed databases where the stored values are related to their local clients and it is very rare to have data queries from distant regions.

We will assume clients located in region R2 sharing it with groups g_4, g_5 and g_2. We can see in Fig. 5.3 that the delay of sending messages to those groups is 5ms. We will consider destinations of increasing size but only taking into account the sets of minimal geographic distance to R2 and call them **near destinations**. For example, when sending messages to destinations d of size 1 (i.e., local messages) we will only consider $d \in M_1 = \{\{g_4\}, \{g_5\}, \{g_2\}\}$ because they are the only destinations in the same region. For messages of destination size 2 we will use $d \in M_2 = \{\{g_4, g_5\}, \{g_4, g_2\}, \{g_5, g_2\}\}$. The other sets of destinations are defined as follow: $M_3 = \{\{g_4, g_5, g_2\}\}$, $M_4 = \{\{g_4, g_5, g_2, g_1\}\}$ and $M_n = \{d : d \in \mathcal{P}(\{g_1, g_2, ..., g_7\}) \wedge |d| = n \wedge g_1 \in d\}$ for $5 \leq n \leq 7$.

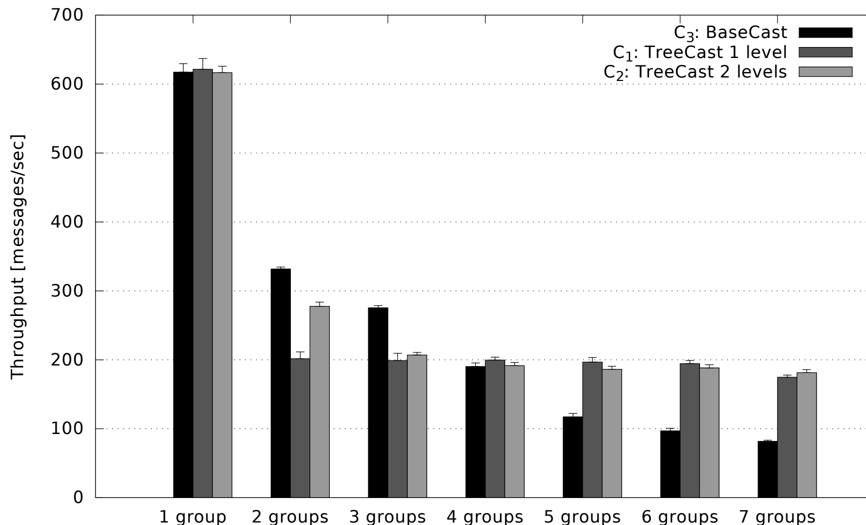

Fig. 5.10: Atomic multicast running in environment WAN_2. Throughput when 150 clients send messages to arbitrary destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval

In the experiment of Fig. 5.10 we executed the algorithms in configurations $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ sending messages to arbitrary destinations. In this scenario the obtained throughtput is slightly better than in WAN_1.$\mathcal{C}_3$ has a good performance for only 2 groups because groups are completly included in the same region, so sending messages with two destinations only crosses one different region. $\mathcal{C}_1$ has a throughput decrease of 50% compared to $\mathcal{C}_3$ since the first always involves the parent group, introducing a significant impact on performance. The good performance of $\mathcal{C}_3$ degrades fast when more groups are addressed: it needs to order messages across all the involved groups, forcing a lot of messages to cross multiple regions and obtaining half of the performance compared to the other configurations. $\mathcal{C}_1$ and $\mathcal{C}_2$ keep a stable performance in the global messages scenario thanks to the tree structure.

We want to see if we can do something better, especially regarding latency and throughput of the nearest destinations.

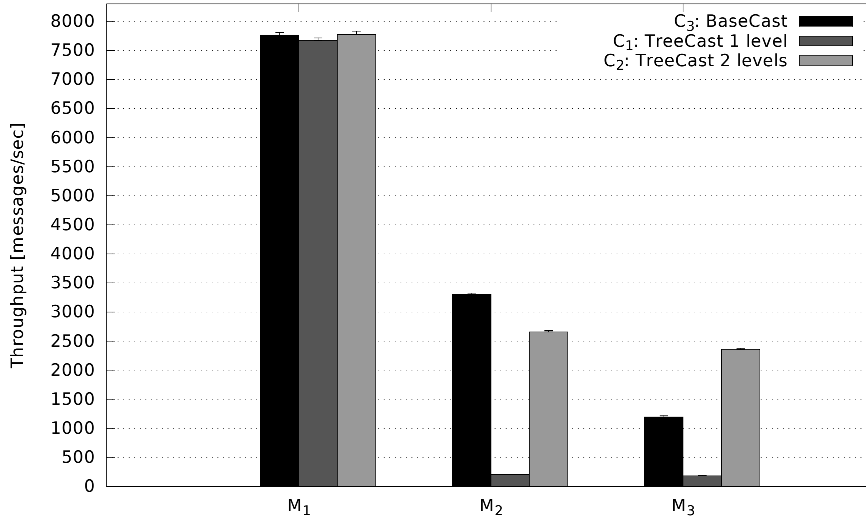

Fig. 5.11: Atomic multicast running in environment WAN_2. Throughput when 150 clients send messages to near destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval

In Fig. 5.11 we show the results of sending messages to destinations $M_1, ..., M_7$ in all the configurations. We can see a throughput of almost 8.000 msg/sec for local messages, and 2.500 msg/sec for messages addressed to M_2 and M_3. These results are one level of magnitude greater than the ones obtained in Fig. 5.10. $\mathcal{C}_1$ general performance is very low compared to others because g_1 (the only parent group) will order all the global messages, which will always cross different regions, degrading the protocol. The genuineness in $\mathcal{C}_3$ gives a good response for messages up to three groups but it gets worse when we need to include other distant regions.

In $\mathcal{C}_2$, we have three parents distributed in different regions, incrementing the chances of getting overlay trees that execute in the same region, which decrements the message interchange with high delay, thus obtaining better results. For messages with 4 or more destinations the throughput of $\mathcal{C}_2$ is similar to the throughput seen in WAN_1 but more stable and slightly better than the rest of the configurations (Fig. 5.12).

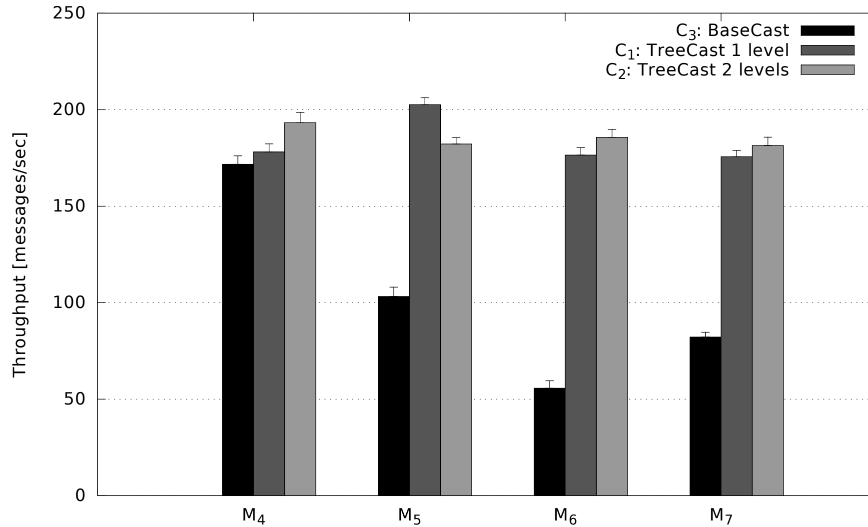

Fig. 5.12: Atomic multicast running in environment WAN_2. Throughput when 150 clients send messages to near destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval

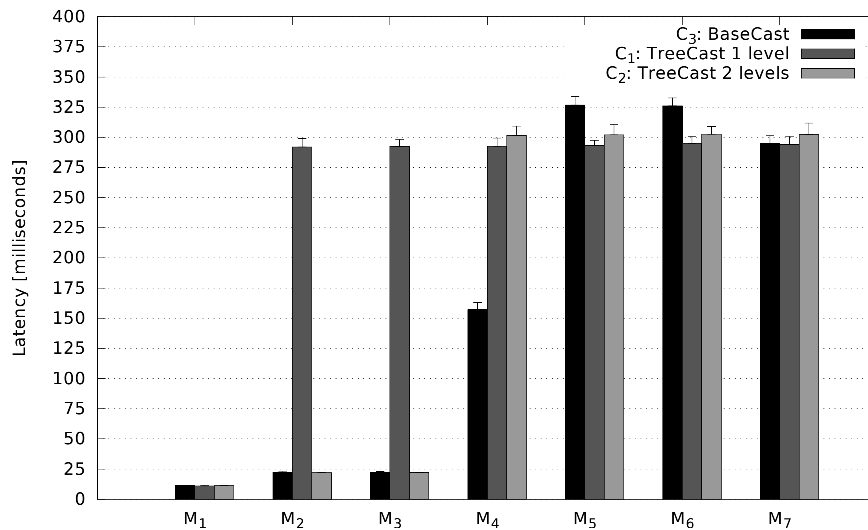

Fig. 5.13: Atomic multicast running in environment WAN_2. Latency when one client send messages to near destinations of increasing size. Bars show median latency and whiskers show 90-th percentile

The latency when sending messages to groups in the same region (i.e., M_1, M_2 and M_3) decreases from 600ms in WAN_1 to approximately 25ms (Fig. 5.13) in configurations $\mathcal{C}_2$ and $\mathcal{C}_3$. Again the topology of $\mathcal{C}_1$ is not appropriate for global messages, so it results in high delays with such configurations. This time the genuineness in $\mathcal{C}_3$ gives the best results with a maximum delay of 160ms for up to four groups.

6. CONCLUSIONS

6.1 Discussion

Atomic multicast is a fundamental communication abstraction in the design of scalable and highly available, strongly consistent distributed systems. We revisited the genuineness of atomic multicast protocols and tested whether relaxing genuineness can lead to better throughput or latency in different network topologies. Two approaches were studied: BaseCast, a genuine atomic multicast algorithm, and TreeCast, a partially genuine atomic multicast. Our implementations only differ on how the core of the protocols work, all the communication and consensus objects are shared, making our comparison more fair.

TreeCast shows a good balance between latency and throughput, but we need to have in mind how we build the topology tree. If our application prioritizes latency, a topology like TreeCast can generate big delays when the overlay trees are not well conditioned. We need a good knowledge of how the clients will behave to build the tree in a way that the most usual destinations are near in the tree. Otherwise, tall overlay trees are needed, and this increments the steps of the algorithm, resulting in high delay.

The results obtained show that genuineness doesn't necessarily relate to good throughput, especially when messages are addressed to a big majority of the groups in the system.

BaseCast performance decreases for every new destination, making genuineness a major drawback in this configuration. Every involved node needs to keep track of the message state to properly a-multicast it, resulting in big computations of all the nodes. In a WAN environment this property can give better results.

To guarantee good results in TreeCast we have to assume some hypotheses about how the clients will behave in terms of the most probable messages destinations. Then, when the assumed behavior is not met we can get really bad performance for messages delivered to a few groups. The latter gets worse if the height of the tree increases, making BaseCast a better option for messages with few recipients and arbitrary destinations.

6.2 Future work

Our two discussed protocols, BaseCast and TreeCast, only manage benign failures (i.e., crash failures). To be able to support arbitrary behavior (i.e., Byzantine failure) some other consensus library is needed (e.g., pBFT [6]). These kind of implementations are usually more complex but have become increasingly appealing in distributed and decentralized scenarios where there is a need for a trusted system (e.g., blockchain) and new applications become more and more sensitive to malicious behavior. These are usually large-scale environments including both a significantly large number of nodes and geo-distribution, so the genuineness of the communication protocol becomes even more important. This work would need a characterization of large-scale environments services and

benchmarks that capture the properties and operations of such services.

Regarding the implementation, the communication between nodes is done using blocking sockets, a very straightforward way for process communication. There are other libraries like Java NIO, a collection of Java programming language APIs that offer features for intensive I/O operations. This exceeded the scope of the thesis but would be an interesting improvement to have in mind.

LIST OF FIGURES

3.1 Steps of the BaseCast protocol . . . 7
3.2 An execution of TreeCast with three messages: m_1 is a-multicast to $\{g_2, g_3, g_5\}$, m_2 to $\{g_4, g_5\}$ and m_3 to g_7 . . . 10
3.3 An overlay tree with seven groups in a binary tree topology where g_1 is the root . . . 10
3.4 Trivial execution of the Paxos algorithm, in which no acceptor crashes and multiple proposers do not try to reserve the same consensus instance . . . 12

4.1 `Node` class diagram, an implementation of `AmcastNode` interface and its related objects. . . . 15
4.2 `ServerCommunication` extends a `Peer` class, which handles the connections with other replicas and clients. . . . 17

5.1 Environment WAN_1 where for every region there is one replica of each group. This configuration supports the failure of a whole region. . . . 20
5.2 TreeCast in $\mathcal{C}_1$: Seven groups with g_1 as root and the only parent of all other groups . . . 21
5.3 Environment WAN_2 running configuration $\mathcal{C}_2$ where every group is completly located in of the three different regions. We asume clients near regions R2 with a latency of 5ms and every group is located in a LAN. . . . 22
5.4 Atomic multicast running in a computer cluster provided by USI (LAN). Throughput when 150 clients send messages to random destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval . . . 23
5.5 Atomic multicast running in a computer cluster provided by USI (LAN). Latency when one client send messages to random destinations of increasing size. Bars show median latency and whiskers show 90-th percentile . . . 23
5.6 Performance impact of TreeCast in configuration $\mathcal{C}_2$. (Left) Clients sending arbitrary messages with $|m.dst| = 2$ versus messages with $m.dst \in \mathcal{D}'_3$. (Right) Clients sending messages with $m.dst \in \mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$. Bars show average throughput. . . . 24
5.7 Atomic multicast running in environment WAN_1. Throughput when 150 clients send messages to random destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval . . . 25
5.8 Atomic multicast running in environment WAN_1. Latency when one client send messages to random destinations of increasing size. Bars show median latency and whiskers show 90-th percentile . . . 25
5.9 Atomic multicast running in environment WAN_1. Performance impact of TreeCast in configuration $\mathcal{C}_2$. (Left) Clients sending arbitrary messages with $|m.dst| = 2$ versus messages with $m.dst \in \mathcal{D}'_3$. (Right) Clients sending messages with $m.dst \in \mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$. Bars show average throughput. . . . 26

5.10 Atomic multicast running in environment WAN_2. Throughput when 150 clients send messages to arbitrary destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval 27
5.11 Atomic multicast running in environment WAN_2. Throughput when 150 clients send messages to near destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval 28
5.12 Atomic multicast running in environment WAN_2. Throughput when 150 clients send messages to near destinations of increasing size. Bars show average throughput and whiskers show 95% confidence interval 29
5.13 Atomic multicast running in environment WAN_2. Latency when one client send messages to near destinations of increasing size. Bars show median latency and whiskers show 90-th percentile 29

BIBLIOGRAPHY

[1] Agarwal, D. A., Moser, L. E., Melliar-Smith, P. M., Budhia, R. K.: *The totem multiple-ring ordering and topology maintenance protocol.* ACM Trans. Comput. Syst., 16(2):93–132, May 1998.

[2] Babay, A., Amir, Y.: *Fast total ordering for modern data centers.* In ICDCS, 2016.

[3] Bezerra, E., Cason, D., Pedone, F.: *Ridge: high-throughput, low-latency atomic multicast.* In SRDS, 2015

[4] Bezerra, C. E., Pedone, F., Van Renesse, R.: *Scalable state-machine replication.* in DSN, 2014.

[5] Birman, K., Joseph, T.: *Reliable communication in the presence of failures.* Trans. on Computer Systems, 5(1):47-76, Feb. 1987.

[6] Castro, M., Liskov, B.: *Practical Byzantine Fault Tolerance.* ACM Transactions on Computer Systems, Vol. 20, No. 4, November 2002.

[7] Chandra, T. D., Toueg S.: *Unreliable failure detectors for reliable distributed systems.* Journal of the ACM, 43(2):225–267, 1996.

[8] Coelho, P., Schiper, N., Pedone, F.: *Fast Atomic Multicast* 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2017

[9] Coelho, P., Ceolin T., Bessani, A., Dotti, F., Pedone, F.: *Byzantine Fault-Tolerant Atomic Multicast* 48th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2018

[10] Corbett, J. C., Epstein M., et al: *Spanner: Google's globally distributed database.* in OSDI, 2012.

[11] Cowling, J., Liskov, B.: *Low-overhead distributed transaction coordination.* In USENIX ATC, 2012.

[12] Défago, X., Schiper, A., Urbán, P.: *Total order broadcast and multicast algorithms: Taxonomy and survey.* ACM Comput. Surv., 36(4):372–421, 2004

[13] Delporte-Gallet, C., Fauconnier, H.: *Fault-tolerant genuine atomic multicast to multiple groups.* In OPODIS, 2000.

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

[15] Fritzke, U., Ingels, P., Mostéfaoui, A., Raynal, M.: *Fault-tolerant total order multicast to asynchronous groups.* In SRDS, 1998.

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

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

[18] Hoang, L. L., Bezerra, C. E. B., Pedone, F.: *Dynamic scalable state machine replication.* in DSN, 2016.

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

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

[21] Li, B., Xu, W., Abid, M. Z., Distler, T., Kapitza, R.: *SAREK: optimistic parallel ordering in byzantine fault tolerance.* in EDCC, 2016.

[22] Marandi, P. J., Primi, M., Schiper, N., Pedone, F.: *Ring paxos: A high-throughput atomic broadcast protocol.* In Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on, pages 527–536. IEEE, 2010.

[23] Marandi, P. J., Primi, M., Pedone, F.: *Multi-ring paxos.* In DSN, 2012

[24] Rodrigues, L., Guerraoui, R., Schiper, A.: *Scalable atomic multicast.* In IC3N, 1998

[25] Schiper, N., Pedone, F.: *Optimal atomic broadcast and multicast algorithms for wide area networks.* In PODC, 2007.

[26] Schneider, F.: *Implementing fault-tolerant services using the state machine approach: A tutorial* ACM Computing Surveys, vol. 22, pp. 299-319, Dec. 1990.

[27] Schiper, N., Pedone, F.: *On the inherent cost of atomic broadcast and multicast in wide area networks.* In ICDCN, 2008.

[28] Sciascia, D., Pedone, F., Junqueira, F.: *Scalable deferred update replication.* In DSN, 2012