FlexCast: Genuine Overlay-based Atomic Multicast

Eliã Batista*
Università della Svizzera italiana
Lugano, Switzerland

Paulo Coelho
Universidade Federal de Uberlândia
Uberlândia, Brazil

Eduardo Alchieri
Universidade de Brasília
Brasília, Brazil

Fernando Dotti
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil

Fernando Pedone
Università della Svizzera italiana
Lugano, Switzerland

ABSTRACT

Atomic multicast is a communication abstraction where messages are propagated to groups of processes with reliability and order guarantees. Atomic multicast is at the core of strongly consistent storage and transactional systems. This paper presents FlexCast, the first genuine overlay-based atomic multicast protocol. Genuineness captures the essence of atomic multicast in that only the sender of a message and the message's destinations coordinate to order the message, leading to efficient protocols. Overlay-based protocols restrict how process groups can communicate. Limiting communication leads to simpler protocols and reduces the amount of information each process must keep about the rest of the system. FlexCast implements genuine atomic multicast using a complete DAG overlay. We experimentally evaluate FlexCast in a geographically distributed environment using gTPC-C, a variation of the TPC-C benchmark that takes into account geographical distribution and locality. We show that, by exploiting genuineness and workload locality, FlexCast outperforms well-established atomic multicast protocols without the inherent communication overhead of state-of-the-art non-genuine multicast protocols.

CCS CONCEPTS

• **Computer systems organization** → *Reliability*; *Availability*; *Redundancy*; *Distributed architectures.*

KEYWORDS

Atomic multicast, Consensus, Fault tolerance

ACM Reference Format:
Eliã Batista, Paulo Coelho, Eduardo Alchieri, Fernando Dotti, and Fernando Pedone. 2023. FlexCast: Genuine Overlay-based Atomic Multicast. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3590140.3629122

*The author is also affiliated with Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil.

1 INTRODUCTION

Atomic multicast is a communication abstraction that propagates messages to groups of processes with reliability and order guarantees. Agreeing on the order of messages in the presence of failures is a notoriously difficult problem [14]. Yet, message ordering is at the core of strongly consistent storage and transactional systems (e.g., [7, 27, 28]). Some systems implement strong consistency using an ad-hoc ordering protocol (e.g., [7, 9]). Atomic multicast encapsulates the logic for ordering messages and thereby reduces the complexity of designing fault-tolerant strongly consistent distributed systems.

In light of their important role, it is not surprising that many atomic multicast protocols have been proposed in the literature (e.g., [10, 11, 15, 23, 24]). These protocols can be classified according to two criteria: (a) genuineness (or lack of) and (b) process connectivity.

Genuineness. In a genuine atomic multicast protocol, only the message sender and destinations communicate to order a multicast message [18]. Some non-genuine atomic multicast protocols order messages using a fixed group of processes or involving all groups, regardless of the destination of the messages. In geographically distributed settings, a genuine atomic multicast protocol can better exploit locality than a non-genuine protocol since messages addressed to nearby groups do not introduce communication with remote groups. Moreover, because a group only receives messages that are addressed to the group, in a genuine atomic multicast protocol groups do not incur communication overhead from relaying messages to the destinations. This is important in geographically distributed environments where communication across wide-area links represents an important cost (e.g., Amazon Web Services).

Connectivity. Most atomic multicast protocols assume that processes can communicate directly with one another. Alternatively, processes communicate following an *overlay*, which determines which processes can exchange messages with which other processes. Imposing limits on communication has advantages. For example, overlays can represent the structure of administrative domains, simplify the design of protocols, and reduce the amount of information each process must keep about the rest of the system (e.g., key management in Byzantine fault tolerant protocols [5]).

Combining genuineness and overlays is challenging. Existing atomic multicast protocols focus on one aspect or the other but not both. For example, all existing genuine atomic multicast protocols assume a fully connected overlay. Hierarchical protocols, which structure communication between groups as a tree, are not genuine.

For example, in ByzCast [5], a multicast message is first sent to the lowest common ancestor of the message destinations, and then proceeds down the tree until it reaches all destinations. ByzCast's logic is simple and processes in a group only need to keep information about their parent and children. However, it is not genuine since a message addressed to the children of group g, but not to g, are first sent to g and then propagated to g's children, violating genuineness.

Figure 1 quantifies ByzCast's communication overhead, computed as one minus the ratio between the number of messages that a group delivers (i.e., messages addressed to the group) and the number of messages the group receives as part of communication imposed by the tree overlay, and expressed as a percentage. On average, groups incur on almost 10% of communication overhead. Some groups, however, are more penalized than others, depending on their position in the tree. In particular, about 23% and 36% of the communication of groups 5 and 9, respectively, is overhead. This is in contrast to genuine atomic multicast protocols, which have no communication overhead.

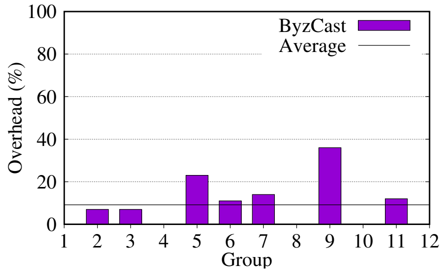


Figure 1: Communication overhead in a hierarchical protocol when executing the gTPC-C benchmark with tree T_1 and 90% of locality (more details in Section 5); overhead, expressed as a percentage, is computed for each group as 1 minus the ratio between number of messages delivered and number of messages received by the group.

Our contribution. This paper proposes FlexCast, the first genuine overlay-based atomic multicast protocol. FlexCast assumes a complete directed acyclic graph (C-DAG) overlay. Multicast messages are sent to the lowest common ancestor (*lca*) of the message destinations. The *lca* then propagates the message to all other destinations in one communication step, without involving any groups that are not a message's destination. FlexCast uses a sophisticated history-based protocol to order messages. First, each process builds a history with all messages the process has delivered. This history is propagated to other processes in the C-DAG, so that processes can ensure consistency (e.g., no two processes order two messages differently). Simply following other processes' histories is not enough to ensure consistent order due to indirect dependencies. Indirect dependencies happen for a few reasons. For example, if process x orders message m_1 before message m_2 and process y orders m_2 before message m_3, then process z must order m_1 before m_3 as a consequence of dependencies created by processes x and y involving m_2, a message not addressed to z. FlexCast is well-suited to equip geographically replicated systems as it exploits locality.

We have implemented FlexCast and evaluated it in an emulated wide-area network that mimics Amazon's EC2. To experimentally evaluate FlexCast, we propose gTPC-C, a variation of the well-known TPC-C benchmark that integrates geographical distribution. In the original TPC-C benchmark, a transaction operates on items in a main warehouse and with a certain probability on items from additional warehouses as well. gTPC-C models real-world wholesale supply systems in which transactions are directed to the customers' nearest warehouse and items not present in this warehouse are requested from the next closest warehouse and so on. In gTPC-C, customers and warehouses are geographically distributed. To account for locality, a customer's main warehouse is the closest one to the customer's location and multi-warehouse transactions have higher probability to involve warehouses located near the main warehouse. Our results show that, by exploiting locality, FlexCast can reduce latency by up to 42% to 46% when compared to state-of-the-art atomic multicast protocols in a geographically distributed environment. Moreover, as a genuine atomic multicast protocol, FlexCast has no communication overhead.

The rest of the paper is structured as follows. Section 2 presents the system model and definitions used in the paper. Section 3 reports on related works. Section 4 presents a detailed description of FlexCast, starting with a high level description of the protocol, then detailing the algorithms, and addressing practical concerns and fault tolerance. Section 5 provides an experimental evaluation of FlexCast. Section 6 concludes the paper.

2 SYSTEM MODEL AND DEFINITIONS

This section presents our system model and recalls the definition of atomic multicast.

2.1 System model

We consider a message-passing distributed system consisting of an unbounded set of client processes $C = \{c_1, c_2, ...\}$ and a bounded set of server processes $S = \{p_1, p_2, ..., p_n\}$. We define the set of server groups as $\Gamma = \{G_A, G_B, ..., G_N\}$, where for every $g \in \Gamma$, $g \subseteq S$. Moreover, groups are non-empty and disjoint [5, 17, 18, 25]. Processes are *correct* if they never fail or *faulty* otherwise. In either case, processes do not experience arbitrary (i.e., Byzantine) behavior. We assume the system is partially synchronous [13]: 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.

2.2 Atomic multicast

Atomic multicast is a fundamental communication abstraction in reliable distributed systems. It encapsulates the complexity of reliably propagating and ordering messages. With atomic multicast, a client can multicast messages to different groups with the guarantee that the destinations will deliver messages consistently. In the following, we precisely capture these reliability and ordering guarantees.

A client atomically multicasts an application message m to a set of groups by calling primitive $multicast(m)$, where $m.sender$ denotes the process that calls $multicast(m)$, $m.id$ is the message's unique identifier, and $m.dst$ is the groups m is multicast to. A server delivers message m calling the primitive $deliver(m)$. If $|m.dst| = 1$

we say that m is a *local* message; if $|m.dst| > 1$ we say that m is a *global* message.

We define the relation $\prec$ on the set of messages server processes deliver as follows: $m \prec m'$ iff there exists a process that delivers m before m'. If $m \prec m'$ or $m' \prec m$, we say that there is a dependency between m and m'.

Atomic multicast satisfies the following properties [19]:

- *Validity*: If a correct process p multicasts a message m, then eventually all correct server processes $q \in g$, where $g \in m.dst$, deliver m.
- *Agreement*: If a process p delivers a message m, then eventually all correct server 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.
- *Prefix order*: For any two messages m and m' and any two server 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'.
- *Acyclic order*: The relation $\prec$ is acyclic.

In a genuine atomic multicast protocol, only the sender and the destinations of a message coordinate to order the message. A genuine atomic multicast protocol does not depend on a fixed group of processes and does not involve processes unnecessarily. More precisely, a genuine atomic multicast algorithm should guarantee the following property [18].

- *Minimality*: If a process p sends or receives a message in run R, then some message m is multicast in R, and p is $sender(m)$ or in a group in $m.dst$.

3 RELATED WORK

An early atomic multicast protocol is attributed to D. Skeen [3]. In this protocol, a multicast message m is first propagated to m's destinations. Upon receiving the message, a destination assigns the message a local timestamp and sends the local timestamp to the other message destinations. When a destination has received timestamp from all message destinations, it computes the message's final timestamp as the maximum among all of the message's local timestamps. Destinations deliver messages in order of their final timestamp. This protocol is genuine but does not tolerate failures.

Several atomic multicast protocols extend Skeen's ordering technique to tolerate failures [6], [15], [17], [22], [23]. In all these protocols, the idea is to implement destinations as groups of processes. Thus, messages are addressed to one or more process groups, instead of a set of processes, as in the original protocol. Although some processes in a group may fail, each group acts as a reliable entity, whose logic is replicated within the group using state machine replication [26]. Recent protocols aim at reducing the cost of replication within groups while keeping Skeen's original idea of assigning timestamps to messages and delivering messages in timestamp order. FastCast [6] improves performance by optimistically executing parts of the replication logic within a group in parallel. WhiteBox atomic multicast [17] uses the leader-follower approach to replicate processes within groups. RamCast [22] relies on distributed shared memory (RDMA) to reduce latency. Since in all these protocols processes communicate directly with one another, we refer to them as *distributed* atomic multicast protocols (see Table 1).

Class	Type	Examples
Distributed	genuine	[3, 6, 11, 15, 17, 22, 23]
Hierarchical	non-genuine	[5, 16, 20]
C-DAG overlay	genuine	FlexCast (this paper)

Table 1: Different classes of atomic multicast protocols.

Delporte and Fauconnier [11] present a genuine distributed atomic multicast protocol that does not rely on exchanging of timestamps to order messages. The protocol assigns a total order to groups and relays messages sequentially through their destination groups following this order. A multicast message m is initially sent to the lowest group in $m.dst$ according to the total order. When the group receives m, it uses consensus to order and deliver m inside the group, then m is forwarded to the next group in $m.dst$, according to the total order of groups. A group that delivers m can only order the next message once it knows m is ordered in all groups in $m.dst$, which is after it receives an END message from the last group in $m.dst$. Although the dissemination of the message follows an order, the END message returns to each group involved and therefore the protocol is a distributed atomic multicast protocol. Besides needing $n + 1$ steps to deliver a message, where n is the number of destinations of the message, since groups remain locked until the END message arrives, this protocol is affected by the convoy effect [1].

Some protocols restrict process communication by means of a tree overlay that determines how groups can communicate (e.g., [5, 16]). To order a message m using a tree, m is first sent to the lowest common ancestor group among those in $m.dst$, in the worst case the root of the overlay tree. Then, m is successively ordered by the lower groups in the tree until it reaches all groups in $m.dst$. An important invariant is that lower groups in the tree preserve the order induced by higher groups. Although simple, this protocol is not genuine since a message may need to be ordered by a group that is not in the destination set of the message. While the tree-based protocol proposed by Garcia-Molina and Spauster [16] does not tolerate failures, ByzCast [5] can withstand Byzantine failures.

The Arrow protocol [20] is a non-fault tolerant tree-based protocol that targets open groups. It emerges from the combination of a reliable multicast protocol with a distributed swap protocol. Arrow assumes a graph G and a spanning tree T on G. Initially, each node v in T has $link(v)$ that is its neighbour in T or itself if v is a sink (initially only the root of T). To multicast m a node v sends a message through $link(v)$, which is forwarded to the root of the tree. By definition, the root has sent the last message before m. As the message is forwarded, edges change direction and v becomes the new root (that has sent the last message, which now is m). Although genuine, this procedure may result in swap messages traversing the diameter of T and only then a multicast, using an underlying reliable multicast, is issued.

Restricting communication as in a tree may lead to simpler atomic multicast algorithms. Moreover, if communication needs to be authenticated, as in Byzantine fault-tolerant protocols, a tree overlay requires fewer keys to be maintained and exchanged between processes than a distributed fully connected protocol. Finally,

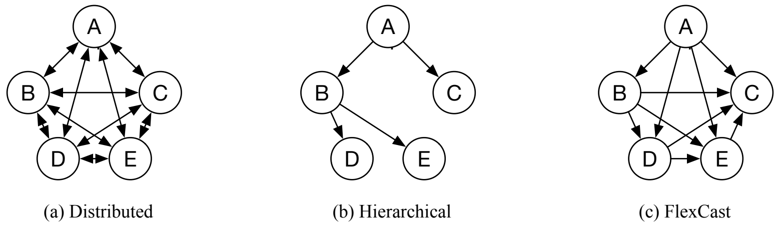


Figure 2: Three communication patterns used in atomic multicast protocols involving groups $A, B, ..., E$: (a) distributed, (b) hierarchical, and (c) FlexCast, the approach presented in this paper. In the graphs, directed edge $g \rightarrow h$ means that group g can send messages to group h, and h can receive messages from g but not send messages to g.

a fully connected protocol is a reasonable assumption in systems that run within the same administrative domain (e.g., Google's Spanner [14]). In other contexts (e.g., decentralized systems), however, multiple entities from different administrative domains collaborate but do not wish to establish connections with all other domains. Hereafter, we refer to protocols based on a tree as *hierarchical* atomic multicast protocols.

Figure 2 shows three cases of interest. All genuine atomic multicast algorithms we are aware of are distributed (Figure 2 (a)). A tree (Figure 2 (b)) is the minimum connectivity needed by any atomic multicast protocol to support an arbitrary workload (i.e., messages can be multicast to any set of groups), as removing one edge from the tree results in a partitioned graph. Hierarchical protocols, however, are not genuine. For example, in Figure 2 (b), a message multicast to groups B and C will first be ordered at A, and then propagated and ordered by B and C. This paper proposes the first overlay-based genuine atomic multicast protocol.

4 GENUINE OVERLAY-BASED ATOMIC MULTICAST

In this section, we present FlexCast's basic idea and detailed algorithm, and conclude with practical considerations and a discussion on fault tolerance. FlexCast's correctness is presented in an extended version of this paper [2].

4.1 General idea

Groups in FlexCast are structured as a complete directed acyclic graph (C-DAG), as the example in Figure 2 (c). We assume there is a total order among groups. Each group is assigned a unique rank in $0..(n-1)$, where n is the number of groups. The C-DAG topology is such that there is a directed edge from each group with rank i to each group with rank j if $i < j$. In this graph, i's *ancestors* have lower rank than i and i's *descendants* have higher rank than i.[1] Figure 2 (c) shows a C-DAG with nodes ordered from lowest to highest as: A, B, D, E, C.

A client atomically multicasts a message m by sending m to m's lowest common ancestor (lca). The lca of a multicast message is the group with the lowest rank among the destinations of the message. At its lca, m is directly delivered and propagated to m's other destination groups (by definition the lca has direct edges with each other destination group in $m.dst$). Similarly to a tree-base atomic multicast, in a C-DAG, a group must respect the dependencies created by its ancestors and propagate dependencies to its descendants. In a C-DAG, however, a group may have multiple ancestors and dependencies can be created by any of them. An important challenge is to ensure that dependencies are properly communicated down the C-DAG without violating the minimality property of genuine atomic multicast. FlexCast uses three strategies to accomplish this, as explained next.

Strategy (a): First, every group keeps track of a *history*, a graph where messages are vertexes and their relative order are edges. A vertex contains a message's id and destinations. Messages delivered at a group are recorded in its history and build a total order within the graph. When a group propagates a message to another one, its history is included. The destination group extends its history with the histories that it receives from other groups and messages it delivers. The history then becomes a graph. More specifically, since ordering is respected (discussed next), the history is a DAG. Destination groups use the history to ensure that messages are delivered consistently across the system.

To understand the need for exchanging histories, consider the scenario depicted in Figure 3 (a), where group A is the lca of messages m_1 (multicast to A and C) and m_2 (multicast to A and B), and group B is the lca of m_3 (multicast to B and C). Since A delivers m_1 before m_2 (i.e., $m_1 \prec m_2$) and B delivers m_2 before m_3 (i.e., $m_2 \prec m_3$), C must deliver m_1 before m_3 to avoid a cycle among delivered messages. But C receives m_3 from B before it receives m_1 from A. By receiving B's history, C knows that it should deliver m_1 and then m_3 to avoid cycles.

Unfortunately, including histories in forwarded messages is not enough to avoid cycles. Intuitively, this happens because not all dependencies are captured in the communication of application messages between groups. There are two cases to consider, depending on whether the group that creates the dependency is aware that it must propagate the dependency to its descendants or not.

Strategy (b): To motivate the case where a group is aware that it should send dependencies to its descendants, consider the execution in Figure 3 (b). In this case, B delivers m_1 before m_2, and C receives m_2 from A (with an empty history) and then m_1 from B (with an empty history since B did not know about m_2 when it sent m_1 to C). Yet, C must deliver m_1 before m_2. FlexCast ensures proper order in such cases as follows. If group g and its descendant h are in the destination of a message m and g is not m's lca, then g sends an ACK

[1]We use the terms "lower" and "higher" groups to denote relative positions of groups in this rank, and "lowest" and "highest" group of a subset of groups, also referring to this rank. "Ancestors" of a group g denote the set of groups lower than g, while "descendants" respectively higher.

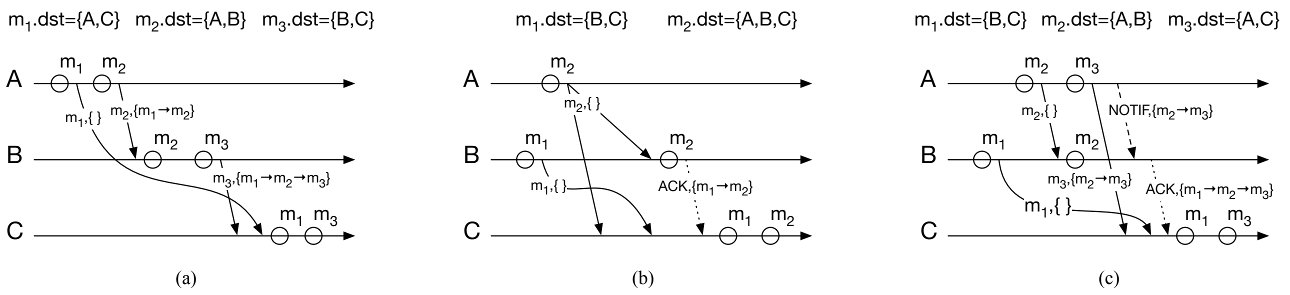


Figure 3: Executions of FlexCast illustrating the use of (a) histories, (b) ACK messages, and (c) NOTIF messages in an overlay where $A \rightarrow B, A \rightarrow C$ and $B \rightarrow C$. (Legend: a full arrow is the propagation of an application message, a circle is the delivery of a message, a dotted arrow is an ACK message, and a dashed arrow is a NOTIF message).

message to h with g's history. Conversely, if h receives a message m and h has an ancestor that is in m's destination, but is not m's lca, h waits for g's ACK message.

Strategy (c): To motivate the case where a group is not aware that it should send dependencies to its descendants, consider the execution in Figure 3 (c). In this case, group A sends m_3 and its history (i.e., m_2 precedes m_3) to C, and B sends m_1 and an empty history to C (i.e., because the dependency between m_1 and m_2 happens in B after B communicates with C). B does not send C the information that m_1 precedes m_2 since m_2 is not addressed to C. Yet, C must deliver m_1 before m_3. To handle this case, when a group determines that a descendant d must forward its history down the C-DAG, it sends a NOTIF message to d so that d can communicate its dependencies to other groups.

More precisely, when a group g, the lca of a message (or another destination in $m.dst$) is about to forward message m (respectively, an ACK message regarding m) and there is a group h such that: (i) h is not in $m.dst$; (ii) h is a descendant of g and an ancestor of group r in $m.dst$; and (iii) there is a message in g's history addressed to h, then g sends a NOTIF message regarding m to h. If group h receives a NOTIF message regarding m, it sends ACK messages to all its descendants $k \in m.dst$. Moreover, inductively, if there is a message h' in h's history with the same restrictions above, h notifies h'. This induction naturally finishes since there is a total order on groups.

4.1.1 Why it is genuine. To argue that FlexCast is genuine, first notice the following aspects discussed about *Strategies (a)* and *(b)*:

- when m is multicast, it enters the overlay at $m.lca()$ (see Algorithm 1), which is by definition a destination of m;
- $m.lca()$ propagates m to its further destinations in $m.dst$; and
- each destination d (other than $m.lca()$) sends ACK messages to groups in $m.dst$ higher than d.

From the above, it follows that the communication described involves exclusively groups in $m.dst$.

Now, consider the *Strategy (c)* and notice that:

- a group $g \in m.dst$ can send a NOTIF message to a group $h \notin m.dst$ provided that g previously sent a message to h, i.e. some message was multicast to h in run R; and
- inductively, h notifies h' only if some message was multicast from h to h' in run R.

From the above, it follows that groups not in $m.dst$ exchange messages only if they communicated in run R, keeping minimality (see definition in Section 2.2).

4.2 Detailed protocol

Algorithm 1 presents the basic data structures used in FlexCast. Each group knows the C-DAG topology and has a communication channel to each descendant group (i.e., a FIFO reliable point-to-point link). As a consequence, each process has an input queue for each input channel from ancestor groups (line 14). Each queue contains not-yet-delivered messages sent by the respective ancestors.

A message has a unique id (line 2), a set of destination groups (line 3), and an arbitrary payload (line 4), provided by the application. The protocol stores pending messages along with a set of respective ACK messages (line 5) and a set of notified groups (line 6), both detailed later. Function $m.lca()$ (line 7) returns the lowest group in $m.dst$.

A group g has the history it learns from each of its ancestors and the messages it delivers (line 15). The set of messages delivered in g is a subset of messages in the history (line 16). The history builds a DAG with dependencies in $hst.D$. As notification messages may not be immediately delivered according to criteria to be detailed later, a group also has a set of pending notification messages (line 17).

When group g communicates with a descendent group h, g informs only the difference in g's history with respect to the last message g sent to h. Therefore, for each descendent h, g keeps track of what part of its history it has already sent to h (line 18).

To atomic multicast message m, a client sends m to $m.lca()$. Algorithm 2 presents the events triggered at a group when receiving each one of the three types of messages in our protocol: (i) MSG is a client message; (ii) ACK is an acknowledge message; and (iii) NOTIF is a notification message. Algorithm 3 presents the core functions used in Algorithm 2.

In FlexCast, the lca delivers a multicast message as soon as it receives the message. Thus, the lca imposes its delivery order on all its descendant groups through information disseminated in histories and auxiliary messages. Upon receiving a message m, if g is the lca (line 1), g can deliver m immediately (line 2).

Algorithm 1 Types and data structures, for each group g

```
 1: Type $Message$: every message $m$ has:
 2:     $m.id$                                   {m's global unique id}
 3:     $m.dst$                      {m's destinations, a subset of groups}
 4:     $m.payload$                          {provided by the application}
 5:     $m.acks \leftarrow \emptyset$                          {a set of received acks}
 6:     $m.notifList \leftarrow \emptyset$                      {a set of notified groups}
 7:     $m.lca() : func$                      {returns the lca in m.dst}
 8: Type (history) $H$:                                    {a history is }
 9:     $H = (M, D, lastDlvd)$       {messages, dependencies, last one}
10:     $M$ : set of $Message$                 {a pair $(m_1, m_2) \in D$ means ...}
11:     $D : M \times M$               {$m_1$ ordered before $m_2$: $m_2$ depends of $m_1$}
12:     $lastDlvd : M \cup \{\perp\}$                {the last message delivered}
13: Group $g$ variables:
14:     $queues \leftarrow [\emptyset, \ldots, \emptyset]$               {an empty queue per ancestor}
15:     $hst \leftarrow H(\emptyset, \emptyset, \perp)$                 {the initial history of group g}
16:     $deliveredInG \subseteq hst.M$          {the messages in hst delivered in g}
17:     $pendNotif \leftarrow \emptyset$                    {a set of pending notifications}
18:     $\forall$ $h$ higher than $g$, $hst(h) \leftarrow H(\emptyset, \emptyset, \perp)$          {the history of g
                                                  informed to each h so far}
```

Algorithm 2 Events, for each group g

```
 1: upon receiving $[\text{MSG}, m, history] \wedge g = m.lca()$
 2:     $a\text{-}deliver(m)$
 3: upon receiving $[\text{MSG}, m, history] \wedge g \neq m.lca()$
 4:     $update\text{-}hst(history)$
 5:     $queues[m.lca()].enqueue(m)$
 6:     $reprocess\text{-}queues()$
 7: upon receiving $[\text{ACK}, m, history]$ from ancestor $a$
 8:     $update\text{-}hst(history)$
 9:     $queues[m.lca()].get(m.id).acks.add([\text{ACK from } a])$
10:     $queues[m.lca()].get(m.id).notifList.\text{merge}(m.notifList)$
11:     $reprocess\text{-}queues()$
12: upon receiving $[\text{NOTIF}, m, history]$
13:     $update\text{-}hst(history)$
14:     $deps \leftarrow open\text{-}dependencies()$
15:     if $deps \neq \emptyset$ then
16:         $pendNotif.add([\text{NOTIF}, m, deps])$
17:     else
18:         $send\text{-}descendants(m, \text{ACK})$
```

When non *lca* groups receive a MSG (line 3) first they update their local history with the history received together with m (line 4), enqueue m in the corresponding ancestor's queue (line 5), and reprocess all ancestors' queues (line 6), since this message may carry the information needed to deliver other messages.

When receiving an ACK message (line 7), g updates its local history (line 8), and associates the ACK to the multicast message m in the *lca*'s queue that originated the ACK (line 9). Since an ACK may identify further groups to be notified, the message's list of notified groups is updated accordingly (line 10). Group g then reprocesses all queues (line 11).

When receiving a NOTIF message (line 12), g updates its local history (line 13), sends the necessary ACK messages (line 18), and possibly sends notification messages to its descendants as well, as detailed later. However, if the local history contains a message m' addressed to g that was not delivered yet, then g waits until it delivers m' before sending the ACK messages, and appends the NOTIF in the pending notifications set (line 16), avoiding propagating incomplete dependencies.

Algorithm 3 Main logic, for each group g

```
 1: update-hst ($ah : H$)                              {ancestor's history ah}
 2:     $hst.M \leftarrow hst.M \cup ah.M$           {messages and dependencies are}
 3:     $hst.D \leftarrow hst.M \cup ah.D$              {intergated to the group's hst}
 4: hst-add ($m : Message$)
 5:     $hst.M \leftarrow hst.M \cup \{m\}$                  {add m, if not yet in hst}
 6:     $hst.D \leftarrow hst.D \cup \{(hst.lastDlvd, m)\}$          {build total order in}
 7:     $hst.lastDlvd \leftarrow m$                 {msgs delivered at this group}
 8:     $deliverdInG \leftarrow deliverdInG \cup \{m\}$
 9: open-dependencies (): set of Messages
10:     return $\{\forall\ m \in hst.M \mid g \in m.dst \wedge m \notin deliveredInG\}$
11: diff-hst($h$ : a higher group) : $H$        {g's history not informed to h so far}
12:     let $hstTmp.M \leftarrow hst.M \setminus hst(h).M$
13:     let $hstTmp.D \leftarrow hst.D \setminus hst(h).D$
14:     let $hstTmp.lastDlvd \leftarrow hst.lastDlvd$
15:     $hst(h) \leftarrow hst$          {history sent to h is updated to current history of g}
16:     return $hstTmp$
17: depend ($m, m' : Message$): boolean
18:     return $(m', m) \in hst.D \vee$
19:         $\exists m'' \mid (m', m'') \in hst.D \wedge$ depend$(m, m'')$
20: a-deliver ($m : Message$)
21:     $hst\text{-}add(m)$
22:     if $g = m.lca()$ then
23:         $send\text{-}descendants(m, \text{MSG})$
24:     else
25:         $queues[m.lca()].dequeue()$
26:         $send\text{-}descendants(m, \text{ACK})$
27:     if $\exists[\text{NOTIF}, n, deps] \in pendNotif \mid m \in deps$ then
28:         $deps \leftarrow deps \setminus m$
29:         if $deps = \emptyset$ then
30:             $pendNotif \leftarrow pendNotif \setminus [\text{NOTIF}, n, deps]$
31:             $send\text{-}descendants(n, \text{ACK})$
32: send-descendants ($m : Message, mType \in \{\text{MSG}, \text{ACK}\}$)
33:     $send\text{-}notifs(m)$
34:     for all $descendant\ d \in m.dst$ do
35:         send $[mType,\ m,\ diff\text{-}hst(d)]$ to $d$
36: send-notifs ($m : Message$)                          {send NOTIF to groups}
37:     for all $descendant\ d \mid d \notin m.dst$ do
38:         if $\exists d' \in m.dst \mid d$ is ancestor of $d'$
            and $hst.containsMsgTo(d)$ then
39:             send $[\text{NOTIF},\ m,\ diff\text{-}hst(d)]$ to $d$
40:             $m.notifList.append(d)$              {m carries the notified groups}
41: reprocess-queues ()
42:     do:
43:         $delivered \leftarrow false$
44:         for all $q \in queues$ do
45:             if $can\text{-}deliver(q.head())$ then
46:                 $a\text{-}deliver(q.head())$
47:                 $delivered \leftarrow true$
48:     while $delivered$
49: can-deliver ($m : Message$)
50:     if $ancestors\text{-}to\text{-}ack(m) \not\subseteq ancestors\text{-}that\text{-}acked(m)$ then
51:         return $false$
52:     if $\exists\ m' \in hst.M \mid g \in m'.dst\ \wedge\ m' \notin deliveredInG \wedge$
                                    $depend(m, m')$ then
53:         return $false$
54:     return $true$
55: ancestors-to-ack ($m : Message$): set of Groups
56:     return (ancestors of $g$ in $m.dst \setminus m.lca()$) $\cup$
                $queues[m.lca()].get(m.id).notifList$
57: ancestors-that-acked ($m : Message$): set of Groups
58:     return queues[m.lca()].get(m.id).acks
```

In Algorithm 3, when g delivers a message, it adds the message to its history (line 4). The total order of delivered messages is built having the new message depend on the last message delivered (lines 6 and 7). We use set *deliveredInG* to identify messages delivered in g (line 8). *deliveredInG* is a subset of *hst.M* and is used to identify possible open dependencies in the history (line 9). An open dependency happens when a message addressed to *g* is included in *g*'s history but not yet delivered. Operation *diff-hst* (line 11) is an optimization: only the new parts of a history are sent to each descendent. Operation *depend* (line 17) computes *m*'s possible transitive dependency on m' in *hst*.

When a message can be delivered (line 20), the group adds the message to its local history (line 21). An *lca* group sends the message to its descendants (line 23), while non-*lca* groups remove the message from the ancestor's queue (line 25) and send the corresponding ACK messages to their descendants (line 26). All groups verify whether delivering this message may unblock pending notifications (line 27).

Function *send-descendants* (line 32) is part of *Strategies (a)* and *(b)* discussed in Section 4.1. To send MSG *m* (or ACK *m*), the *lca* (or a descendant), first sends possible notification messages to its descendants that are not in *m.dst*. Function *send-notifs()* implements *Strategy (c)*: it searches past messages and evaluates if notifications are needed, including the notified groups in *m*'s notification list (lines 33 and 36-39). Then, *m* is sent to all other destinations in *m.dst* (line 35), carrying the list of notified groups along with the history with information needed by each destination (*diff-hst*).

Function *reprocess-queues()* (lines 41-48) is called upon receiving MSG and ACK messages (see Algorithm 2, lines 6 and 11). In both cases, it iterates through ancestor's queues and tries to deliver messages. It keeps iterating while messages can be delivered due to updated dependency information. The delivery of messages in non-*lca* groups is defined in function *can-deliver(m)* (line 49). The first condition (line 50) checks whether *g* received ACK from all needed ancestors: (i) all ancestors (except the *lca*) in *m.dst*; (ii) all ancestors (not in *m.dst*) NOTIF-ied about message *m*, which were informed to *g* either through MSG or ACK. Recall that a notified group, besides sending ACK can further notify other groups. In Algorithm 2, line 10, *notifList* accumulates all notified ancestors that have to ACK *m*. The list of ancestors that have acked is kept in *ancestors-that-acked* (line 57). Having the complete information on *m*, the second condition (line 52) ensures that any message m' that precedes *m* and is addressed to *g* has already been delivered before *m*'s delivery.

4.3 Practical considerations

The protocol as described so far does not include garbage collection. In our FlexCast prototype, however, we prune local histories associated with each ancestor group. A distinguish process periodically multicast a $flush$ message to all groups. Once a group delivers this message, it knows that all messages that precede $flush$ can be garbage collected. The intuition behind this mechanism is that to deliver a message *m* from a specific ancestor, all dependencies before *m* must be resolved and do not need to be re-evaluated in the future. To further reduce communication, histories sent with messages do not enclose the ever-growing system history. FlexCast sends only a *diff* of the history for each descendant group. The idea is implemented by keeping track of the last message of the local history sent to each descendant *d* and, in subsequent messages to *d*, sending a history that contains only the newest messages added since the last communication to *d*.

4.4 Tolerating failures

FlexCast uses the same approach used in other atomic multicast protocols to tolerate failures (e.g., [6], [15], [17], [22], [23], [5]), that is, processes within a group are kept consistent using state machine replication. This means that processes in a group can fail as long as enough processes remain operational within the group. Consequently, groups do not fail as a whole and must remain connected (i.e., no network partition). Tolerating the failure of a group requires additional system assumptions [25].

The implications of this approach on the number of correct processes per group and process communication depend of the particular consensus protocol used to implement state machine replication within a group. For example, Paxos [21] requires a majority of correct processes within each group and can tolerate message losses. In our prototypes, we rely on TCP connections to ensure reliable communication.

5 EVALUATION

In this section, we explain the evaluation rationale, describe the environment and the benchmarks used, present the results, and summarize the main lessons learned.

5.1 Evaluation rationale

We compare FlexCast to a distributed atomic multicast protocol and a hierarchical atomic multicast protocol using single-process groups (i.e., no failures are tolerated) in all three protocols. In doing so, our evaluation focuses on the inherent costs of three classes of atomic multicast protocols (see Table 1) and avoids overhead introduced by replication. We use Skeen's protocol as distributed atomic multicast because its ordering mechanism is used by several other protocols (e.g., [6], [15], [17], [22], [23]). Moreover, when groups contain a single process, FastCast [6] and Whitebox [17] atomic multicast protocols behave as in Skeen's protocol. Skeen's protocol is genuine, can order messages in two communication steps, which has been shown to be optimum [24], and assumes that any two groups can communicate. We choose ByzCast as hierarchical atomic multicast protocol. ByzCast is non-genuine and imposes a tree overlay on communication, the minimum overlay that ensures a connected system. In single-process groups, ByzCast does not introduce any overhead particular to tolerating malicious behavior. We implemented prototypes of all protocols in Java.

Our experimental evaluation aims to understand the behavior of the considered protocols in geographically distributed deployments subject to realistic workloads. Our workload extends the well-established TPC-C benchmark to accommodate locality, a common property in geo-distributed systems. In these settings, we seek to answer the following questions: (i) What is the impact of different overlays on FlexCast and hierarchical protocols? (ii) How quickly can a protocol order messages addressed to two or more groups? (iii) What is the communication overhead of hierarchical protocols? (iv) What is the communication cost of atomic multicast protocols?

5.2 Environment and deployment

The experimental setup was configured with 12 server machines and 24 client machines, connected via a 1-Gbps switched network, in CloudLab [12]. The machines are equipped with eight 64-bit

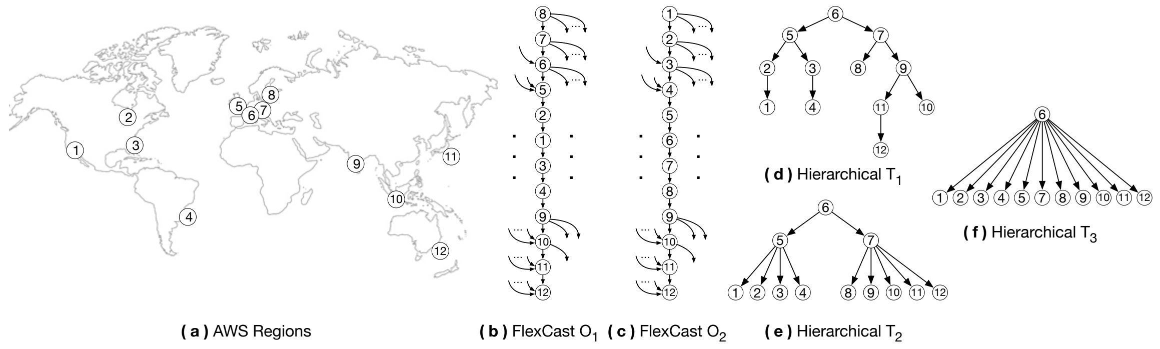


Figure 4: AWS regions and different overlays used in our experimental evaluation.

ARMv8 cores at 2.4 GHz, and 64GB of RAM. The software installed on the machines was Linux Ubuntu 20.04 (64 bits) and 64-bit Java virtual machine version 11.0.3. Machines communicate via TCP.

We consider an emulated wide-area network that models Amazon Web Services (AWS): Each group represents an AWS region and we experimented with a deployment of 12 AWS regions, as depicted in Figure 4 (a). The emulated latencies among regions are based on real measurements in AWS [4]. Enough client processes (to saturate our FlexCast implementation) are uniformly distributed along the 24 client machines that represent each region/group, and they send requests to the nearest group. Upon delivering a message, each message destination replies to the message's sender (client).

5.3 gTPC-C Benchmark

We developed gTPC-C, a geographically distributed benchmark inspired by the well-established TPC-C benchmark [8]. We translate TPC-C warehouses into groups, deployed in one or more AWS regions, and TPC-C transactions into messages multicast to their corresponding warehouses.

According to the TPC-C benchmark, clients can generate the following transactions (with a certain probability): new order (45%), payment (43%), order status (4%), delivery (4%), or stock level (4%). The last three transactions are single-warehouse (local), resulting in a message multicast to the client's home warehouse. Since all multicast protocols perform the same when ordering a message multicast to a single group, in our latency measurements we only consider global transactions, which result in messages addressed to multiple warehouses. Consequently, this workload only contains new order and payment transactions, always involving two or more warehouses. New order transactions can have from 5 to 15 items, where each item has a 2% probability of being issued to a warehouse that is not the client's home warehouse, as defined by TPC-C.

To capture locality, when choosing an additional warehouse to the client's home warehouse, the client picks the nearest warehouse to its home warehouse with a configurable high probability, the *locality* rate; otherwise, the client chooses the next nearest warehouse, and so on, up to the farthest warehouse to the client's home warehouse. Our criteria to define locality is inspired by a common wholesale supplier policy that when an item is not available in the nearest warehouse to a client (i.e., the home warehouse), it is shipped from the closest warehouse that has the item. This locality specification implies that most messages are addressed to only two warehouses (same as in standard TPC-C), and some to three. Very few are addressed to more than three groups, therefore we do not consider these messages in our experiments.

Clients operate in a closed loop issuing one transaction at a time and are deployed in the same region as their home warehouse. Each experiment lasts for a period of approximately one minute, in which clients collect and store latency data. We discard the first and last 10% of the data collected during the experiment to avoid possibly noisy data during warm up and end of execution.

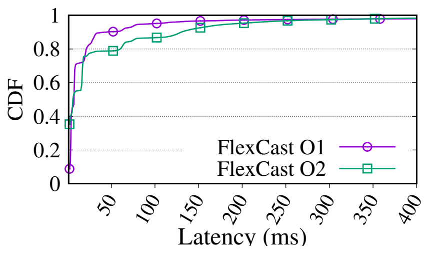
5.4 The effect of overlays

In the first set of experiments, we investigate the role of overlays on FlexCast and hierarchical protocols. We compare the latency experienced by clients of two FlexCast overlays, and three hierarchical overlays (trees), as depicted in Figure 4.

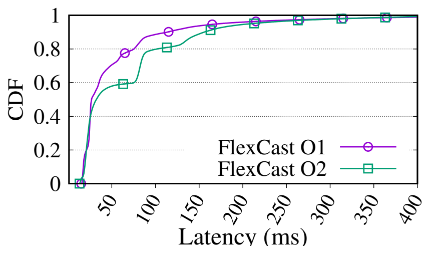
Trees T_1, T_2 and T_3 contain different numbers of inner nodes. In principle, a larger number of inner nodes provides better distribution of communication overhead among these nodes. Trees with many inner nodes, however, may lead to additional communication steps when ordering messages. For overlays O_1 and O_2, we initially selected a starting node (i.e., central node 8 in O_1 and left-most node 1 in O_2). Then, the closest node to the initial one, the closest node to the second chosen node, and so on. Since O_1 and O_2 are complete DAGs, a node is connected to all nodes that succeed it (e.g., the first node is connected to all nodes).

Figure 5 and Table 2 present the results. We report the latency per group addressed by the message. The latency of the first (respectively, second and third) destination corresponds to the first (respectively, second and third) response the client receives from the groups addressed by the message. O_1 shows better performance than O_2 for all destinations. This happens because O_1 better exploits locality: higher nodes in the DAG have the lowest latencies in the geographical distribution. Hereafter, we evaluate FlexCast using overlay O_1.

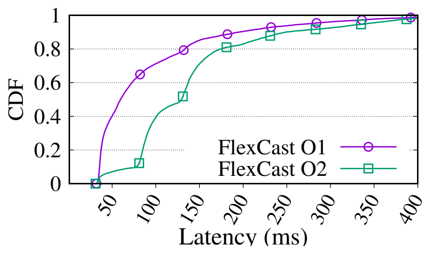
Differently than FlexCast, whose performance is largely dependent on the overlay, a hierarchical protocol is not so sensitive to the chosen tree (but see also the discussion in Section 5.6), although the trees do have an impact on the performance. T_1 shows slightly



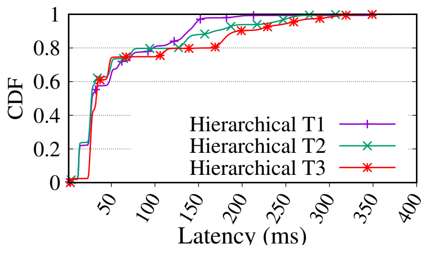
(a) 1st destination



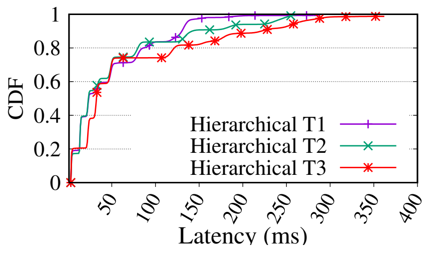
(b) 2nd destination



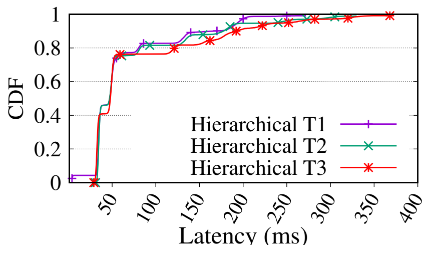
(c) 3rd destination



(d) 1st destination



(e) 2nd destination



(f) 3rd destination

Figure 5: Latency per destination group when varying overlays in FlexCast and a hierarchical protocol, gTPC-C with 90% locality.

better performance in all destinations than T_2 and T_3. This is due to the communication overhead (further discussed in Section 5.8) of involving non-destination groups, and also the bottleneck effect of involving the tree root on T_3 for all messages in the system. From these results, we select T_1 to represent a hierarchical protocol in the rest of our evaluation.

5.5 Throughput

In the second set of experiments, we assess the overall performance of our standard gTPC-C, including local and global messages, when deployed in a configuration with 99% locality rate. We conduct

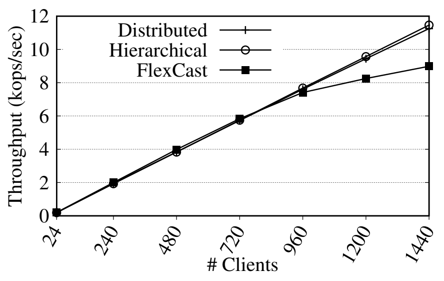


Figure 6: Throughput vs. number of clients with 99% locality.

multiple experiments while gradually increasing the number of clients and measure the total number of transactions ordered by each protocol. Figure 6 presents the results. Although FlexCast was designed to optimize latency, it can maintain the same throughput as the other protocols up to its saturation point. This effect can be seen by the slight bend of the throughput curve of FlexCast starting with 960 clients. In the experiments presented next, we consider configurations with 240 clients. This is justified by the fact that none of the algorithms is subject to queuing effects, which would interfere with their inherent latency.

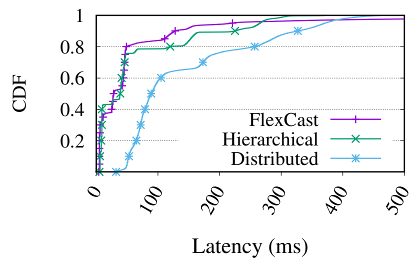
5.6 Latency

In the third set of experiments, we increase the locality rate and measure the latency experienced by the clients when receiving a response from each of the destinations of a global multicast message. Figure 7 and Table 3 present the results. FlexCast outperforms both a distributed and hierarchical protocols in the latency of the first destination group for all three experimented locality rates. We attribute this behavior to the fact that FlexCast benefits from two aspects that reduce the cost of ordering messages in the first destination in a distributed scenario: *(i) Communication steps:* while in a distributed protocol groups addressed by a message need to exchange timestamps before a destination group can deliver a message, in FlexCast the first destination group in the DAG (i.e., the *lca* of the message) can deliver the message as soon as it receives the message from a client; the hierarchical protocol also benefits from this aspect, however, in ByzCast, the *lca* of a message may not be a message destination since it is not a genuine protocol. *(ii) Locality rate:* having a workload with a high locality rate increases the number of messages that FlexCast can deliver using fewer communication steps than both other protocols. This gives FlexCast an advantage since the cost for a communication step may take tens of milliseconds in geographical settings.

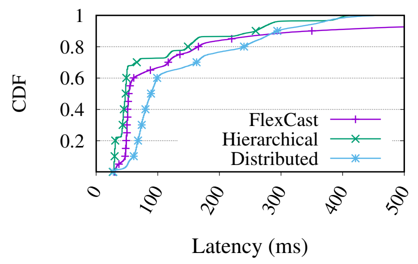
In the second destination, FlexCast performs worse than the hierarchical protocol and outperforms the distributed protocol. As in the discussed above, hierarchical protocols need only one extra communication step to order a message at the second destination, while the distributed protocol, in addition to require destination groups to communicate, is also exposed to the convoy effect, which further slows down the delivery of messages [17]. In the third

		Destination								
		1st			2nd			3rd		
	Overlay	90p	95p	99p	90p	95p	99p	90p	95p	99p
FlexCast	O_1	144.0	279.0	1403.1	398.0	829.0	2243.42	1406.0	2195.0	4542.5
	O_2	156.0	350.0	790.22	416.0	652.0	2006.83	1028.0	1681.5	3112.9
Hierarchical	T_1	229.0	267.0	311.0	261.0	288.0	403.0	307.0	386.0	408.0
	T_2	233.0	269.0	311.0	215.0	249.1	351.0	261.0	338.0	375.28
	T_3	311.0	398.0	544.0	381.0	480.0	622.0	397.0	531.6	621.0

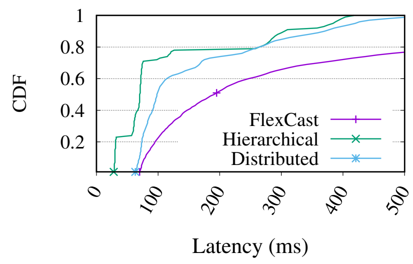
Table 2: Latency percentiles in milliseconds for each destination group when varying the overlay in FlexCast and the tree in the hierarchical protocol, gTPC-C with 90% locality.



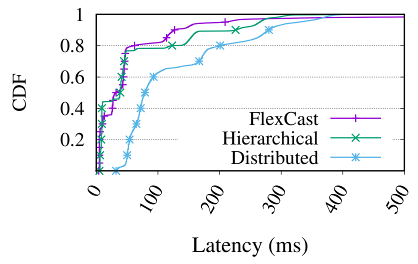
(a) 1st destination, 90% Locality



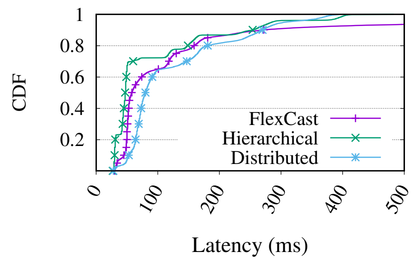
(b) 2nd destination, 90% Locality



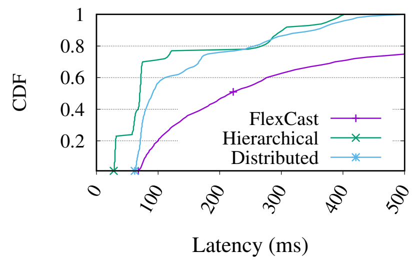
(c) 3rd destination, 90% Locality



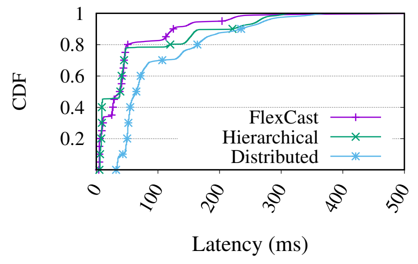
(d) 1st destination, 95% Locality



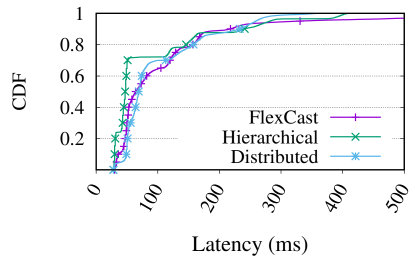
(e) 2nd destination, 95% Locality



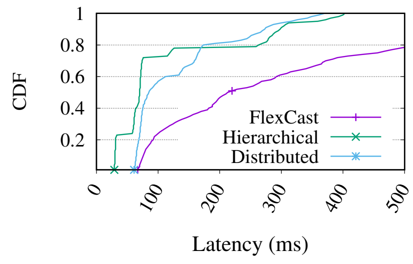
(f) 3rd destination, 95% Locality



(g) 1st destination, 99% Locality



(h) 2nd destination, 99% Locality



(i) 3rd destination, 99% Locality

Figure 7: Latency per destination group when varying locality rate.

destination, FlexCast latency increases and the simplicity of a hierarchical protocol algorithm pays off. In both the second and third destinations, FlexCast may need extra communication steps to receive the necessary ACK messages to deliver a multicast message *m*, evaluate possible dependencies, and wait for dependencies to be solved (i.e., waiting for the delivery of previous messages ordered before *m* in ancestor groups). Although FlexCast performs worse than both hierarchical and distributed protocols in the third destination, messages addressed to three (or more) groups are rare in gTPC-C, a characteristic inherited from TPC-C.

As a consequence of FlexCast's C-DAG overlay and the fact that each client in the gTPC-C benchmark is associated with the nearest warehouse, clients send most of their messages to their home warehouse and to the next nearest warehouse. The rate at which this phenomenon happens is regulated by the configured locality.

		Destination								
		1st			2nd			3rd		
	Locality	90p	95p	99p	90p	95p	99p	90p	95p	99p
FlexCast	90%	144.0	279.0	1403.1	398.0	829.0	2243.42	1406.0	2195.0	4542.5
	95%	131.0	217.0	1146.0	288.0	671.4	2192.64	1307.2	2231.65	4211.55
	99%	132.0	218.0	764.0	227.0	458.0	1562.09	1404.9	1975.7	3583.92
Hierarchical	90%	229.0	267.0	311.0	261.0	288.0	403.0	307.0	386.0	408.0
	95%	226.0	265.0	307.0	255.0	286.0	403.0	306.0	381.0	405.0
	99%	224.0	264.0	303.0	243.0	284.0	402.0	303.0	376.2	406.84
Distributed	90%	335.0	377.0	452.0	299.0	367.0	444.0	373.0	423.0	527.7
	95%	284.0	349.0	417.0	275.0	339.0	406.98	365.0	407.0	528.0
	99%	241.0	279.0	370.0	238.0	263.0	355.0	309.5	367.0	415.3

Table 3: Latency percentiles in milliseconds for each destination when varying the locality rate for all protocols.

Therefore most messages in the workload have a disjoint destination set. This increases FlexCast's advantage over a distributed protocol when messages are addressed to two groups if the groups are placed consecutively in the C-DAG. The hierarchical protocol also benefits from locality, although as a non-genuine protocol, it introduces communication overhead, quantified in Section 5.8. The locality rate also helps to decrease the number of auxiliary messages (i.e., ACK and NOTIF) needed by FlexCast to ensure consistency in the global total order, since interdependencies will be relatively fewer in such a scenario. Table 3 shows the latency percentiles (90, 95 and 99) of all destinations when varying the locality rate for all techniques. Although the hierarchical protocol shows on average a better performance when aggregating the latencies of all destinations, FlexCast is more sensitive to locality. In the first destination, FlexCast's reduces 90p latency by 9% when increasing locality from 90% to 99%, while the hierarchical protocol reduces by 3%. Despite its higher latency, the distributed protocol reduces latency by up to 29% when increasing locality from 90% to 99%.

5.7 The cost of exchanging histories

In this section, we evaluate the amount of information required by each protocol to implement atomic multicast. All protocols propagate the message payload, as defined by gTPC-C, and protocol-specific information, which in the case of FlexCast includes histories. Figure 8 displays our findings. In each chart, the first graph (top) represents the number of messages received by each node per second. The second graph (middle) shows the average message size per node. Unlike the other protocols with fixed average sizes, FlexCast shows an increase in average message size as nodes ascend the C-DAG topology (see Figure 4). This is due to higher nodes requiring more history data from their ancestors. The third graph (bottom) shows the overall data exchanged by nodes per second.

In summary, our experiments indicate that FlexCast exhibits distinctive behavior, with higher nodes in FlexCast's C-DAG exchanging a higher amount of data than lower nodes. This results in larger messages compared to the other protocols. On average, a node exchanges 68.5 Kbytes per second in the distributed protocol, 66 Kbytes per second in the hierarchical protocol, and 79 Kbytes per second in FlexCast.

5.8 The overhead of non-genuineness

In this section, we assess the communication overhead of non-genuine hierarchical protocols (Figures 1 and 9). Intuitively, communication overhead captures the amount of communication involving a group due to multicast messages not addressed to the group. We express communication overhead as a percentage and define it as 1 minus the ratio between the number of payload messages delivered by a group and the number of payload messages received by the group during an execution of the protocol. We focus on payload messages as these are typically larger than auxiliary messages used in a protocol.

The overhead across groups depends on the tree overlay and the workload. But while all inner groups in a tree are potentially subject to communication overhead, leaf groups have no overhead since they are always in the destinations of messages they receive. Locality also plays a role in communication overhead. A tree can benefit from locality by directly connecting groups that are near each other. This is the motivation behind tree T_1: as locality increases, T_1's overhead decreases, since communication will more likely involve directly connected groups (see Table 4).

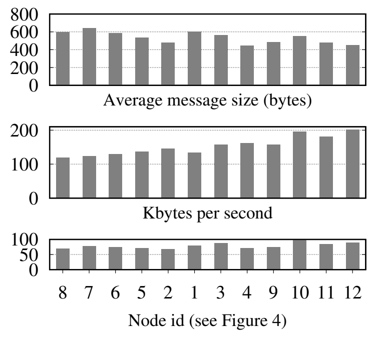
Tree T_3 has lower communication overhead than T_1, but this comes at the cost of penalizing group 6 (i.e., T_3's root), which has to endure 56% of overhead. In T_1, groups 5 and 9 present high overhead as they are roots (lowest common ancestors) of different subtrees that represent separate geographical regions (America and Asia). The tree root does not have much overhead since locality is high in groups within the Europe region. The same is observed in T_2, where groups 5 and 7 of disjoint subtrees present the highest overheads.

Tables 2 and 4 suggest a tradeoff: trees with the lowest latencies are subject to higher overhead on average, while trees with worse performance have lower communication overhead on average.

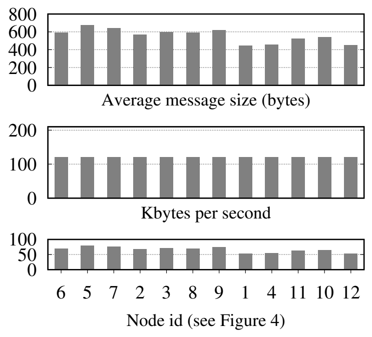
5.9 Summary

We draw the following conclusions from our evaluation.

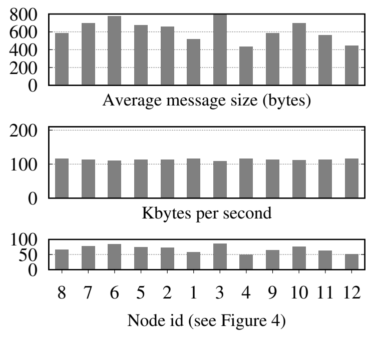
- FlexCast is more sensitive to the chosen overlay than the hierarchical protocol when it comes to latency. The chosen tree, however, has an impact on the hierarchical protocol's communication overhead.
- FlexCast consistently outperforms the distributed protocol (a genuine algorithm) in all configurations experimented.



(a) FlexCast

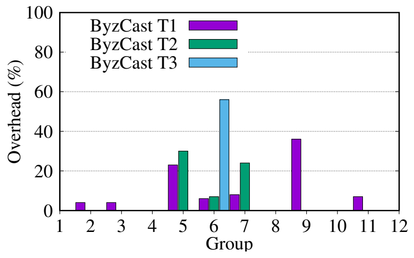


(b) Hierarchical

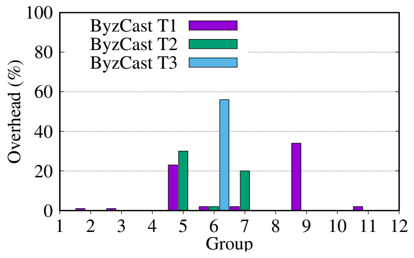


(c) Distributed

Figure 8: The amount of information exchanged by each protocol (99% locality, 720 clients).



(a) 95% Locality



(b) 99% Locality

Figure 9: Communication overhead of each group in hierarchical protocols with 95% and 99% of locality.

Overlay	Locality	Mean overhead	Max
T_1	90%	9.16% (11.18)	36%
	95%	7.33% (11.12)	36%
	99%	5.41% (11.06)	34%
T_2	90%	5.75% (11.31)	30%
	95%	5.08% (10.50)	30%
	99%	4.33% (9.90)	30%
T_3	90%	4.66% (16.16)	56%
	95%	4.66% (16.16)	56%
	99%	4.66% (16.16)	56%

Table 4: Mean overhead, standard deviation, and maximum overhead in hierarchical trees when varying the locality rate.

FlexCast performs better than the hierarchical protocol in the first destination group and worse in the latency of the second and third destinations. However, messages addressed to three (or more) groups are rare in TPC-C and gTPC-C. As a genuine protocol, FlexCast has no communication overhead (Section 5.8), in contrast to a hierarchical protocol.

- The hierarchical protocol has a tradeoff between latency and communication overhead. Although communication overhead is inherent to non-genuine atomic multicast protocols, in the hierarchical protocol, trees with the best performance have the highest overhead and vice-versa.

6 CONCLUSION

We propose FlexCast, the first genuine overlay-based atomic multicast protocol. As overlay-based, FlexCast accounts for reduced connectivity in different deployment scenarios. As genuine, it favors geographical locality and avoids communication overhead. To combine both aspects, FlexCast assumes a complete DAG overlay. Since messages may enter the overlay at different groups (nodes) of the DAG, each group takes local ordering decisions.

One interesting challenge solved by FlexCast and not yet addressed by other atomic multicast protocols is how to ensure global acyclic order out of local ordering information from different groups. This is achieved using a sophisticated history-based protocol. We present FlexCast's design, its implementation, and propose a new benchmark to evaluate it: gTPC-C integrates geographical distribution and locality to the well-known TPC-C benchmark. FlexCast shows important latency reduction in geographically distributed settings when compared to a latency-optimum genuine atomic multicast algorithm and a non-genuine hierarchical protocol.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Antonis Katsarakis, for their valuable feedback. This work was partially supported by the Swiss National Science Foundation (# 175717), Fundação de Amparo à Pesquisa do Estado Do Rio Grande do Sul—FAPERGS PqG 07/21, Conselho Nacional de Desenvolvimento Científico e Tecnológico—CNPq Universal 18/21, PUCRS-PrInt, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brazil, Finance Code 001, and FAPDF through EDITAL 08/2023—FAP Participa.

REFERENCES

[1] T. Ahmed-Nacer, P. Sutra, and D. Conan. 2016. The Convoy Effect in Atomic Multicast. In *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE Computer Society, Los Alamitos, CA, USA, 67–72. https://doi.org/10.1109/SRDSW.2016.22

[2] E. Batista, P. Coelho, E. Alchieri, F. Dotti, and F. Pedone. 2023. *FlexCast: genuine overlay-based atomic multicast.* https://arxiv.org/abs/2309.14074

[3] Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.* 5, 1 (jan 1987), 47–76. https://doi.org/10.1145/7351.7478

[4] Cloudping. 2022. *AWS Latency Monitoring Website.* https://www.cloudping.co/grid

[5] Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. 2018. Byzantine Fault-Tolerant Atomic Multicast. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* 39–50. https://doi.org/10.1109/DSN.2018.00017

[6] Paulo Coelho, Nicolas Schiper, and Fernando Pedone. 2017. Fast Atomic Multicast. In *DSN.*

[7] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2012. Spanner: Google's globally-distributed database. In *OSDI.*

[8] Transaction Processing Performance Council. 1996. TPC benchmark C Standard Specification. *http://www. tpc. org/tpcc/spec/tpcc_current. pdf* (1996).

[9] James Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference.* USENIX, Boston, MA, USA.

[10] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.* 36, 4 (2004).

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

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC).* 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323. https://doi.org/10.1145/42282.42283

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson. 1985. Impossibility of distributed consensus with one faulty processor. *J. ACM* 32, 2 (1985), 374–382.

[15] Jr. Fritzke, U., P. Ingels, A. Mostefaoui, and M. Raynal. 1998. Fault-Tolerant Total Order Multicast to Asynchronous Groups. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems.* 228–234.

[16] H. Garcia-Molina and A. Spauster. 1989. Message ordering in a multicast environment. In *[1989] Proceedings. The 9th International Conference on Distributed Computing Systems.* 354–361. https://doi.org/10.1109/ICDCS.1989.37965

[17] Alexey Gotsman, Anatole Lefort, and Gregory Chockler. 2019. White-Box Atomic Multicast. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 176–187.

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

[19] Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems.* Technical Report. USA.

[20] Fabian Kuhn and Rogert Wattenhofer. 2004. Dynamic Analysis of the Arrow Distributed Protocol. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Barcelona, Spain) *(SPAA '04).* Association for Computing Machinery, New York, NY, USA, 294–301. https://doi.org/10.1145/1007912.1007962

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

[22] Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo R. Coelho, and Fernando Pedone. 2021. RamCast: RDMA-based atomic multicast. *Proceedings of the 22nd International Middleware Conference* (2021).

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

[24] Nicolas Schiper and Fernando Pedone. 2008. On the inherent cost of atomic broadcast and multicast in wide area networks. In *International conference on Distributed computing and networking (ICDCN).* 147–157.

[25] Nicolas Schiper and Fernando Pedone. 2008. Solving Atomic Multicast When Groups Crash. In *International Conference On Principles Of Distributed Systems (OPODIS).* Springer, 481–495.

[26] F. B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[27] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. 2012. Scalable deferred update replication. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on.* IEEE, 1–12.

[28] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* 1–12.