A Communication-Efficient Causal Broadcast Protocol

João Paulo de Araujo
Sorbonne Université, CNRS, INRIA, LIP6
Paris, France
joao.araujo@lip6.fr

Luciana Arantes
Sorbonne Université, CNRS, INRIA, LIP6
Paris, France
luciana.arantes@lip6.fr

Elias P. Duarte Júnior
Federal University of Paraná
Curitiba, Brazil
elias@inf.ufpr.br

Luiz A. Rodrigues
Western Paraná State University
Cascavel, Brazil
luiz.rodrigues@unioeste.br

Pierre Sens
Sorbonne Université, CNRS, INRIA, LIP6
Paris, France
pierre.sens@lip6.fr

ABSTRACT

A causal broadcast ensures that messages are delivered to all nodes (processes) preserving causal relation of the messages. In this paper, we propose a causal broadcast protocol for distributed systems whose nodes are logically organized in a virtual hypercube-like topology called *VCube*. Messages are broadcast by dynamically building spanning trees rooted in the message's source node. By using multiple trees, the contention bottleneck problem of a single root spanning tree approach is avoided. Furthermore, different trees can intersect at some node. Hence, by taking advantage of both the out-of-order reception of causally related messages at a node and these paths intersections, a node can delay to one or more of its children in the tree, the forwarding of the messages whose some causal dependencies it knows that the children in question can not satisfy yet. Such a delay does not induce any overhead. Experimental evaluation conducted on top of PeerSim simulator confirms the communication effectiveness of our causal broadcast protocol in terms of latency and message traffic reduction.

CCS CONCEPTS

• **Networks** → **In-network processing**;

KEYWORDS

Message Aggregation, Causal Order, Distributed Spanning Tree, Hypercube-like Topology

ACM Reference Format:
João Paulo de Araujo, Luciana Arantes, Elias P. Duarte Júnior, Luiz A. Rodrigues, and Pierre Sens. 2018. A Communication-Efficient Causal Broadcast Protocol. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225121

1 INTRODUCTION

In distributed and parallel applications, processes cooperate among themselves to perform some task, often requiring to communicate with each other as a single group. Therefore, a communication service which offers a message broadcast primitive that enables a node to send a message to all other members of the group is of great importance.

Due to the well-known logarithmic scalable properties of trees, several broadcast protocols organize the nodes of the system in a single static distributed spanning tree [19, 36, 42] that comprises all nodes. Every message to be broadcast is then disseminated from the root of this tree to the other nodes. However, this approach presents the drawback that the root can become a bottleneck since all message broadcasts start from it. Therefore, the ideal would be to spread the root load by having one spanning tree per node, i.e., every node of the system is a root of the spanning tree over which it broadcasts its own messages. The trees should cover all nodes, yet organized differently. Figure 1, where all nodes of the system broadcast messages, confirms our statement. If messages are disseminated over one single tree, there is a time where the root of the tree will start queuing them because it can not process all of them as fast as the input broadcast request rate. On the other hand, if each node has its own broadcast tree, the load of messages is better distributed among the nodes. Furthermore, reception latency scales well for an increasing number of nodes/messages.

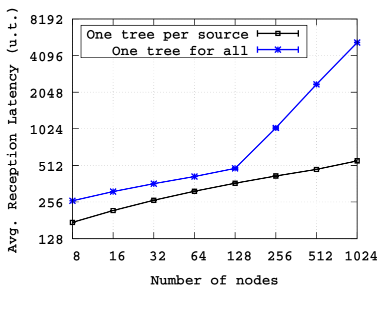


Figure 1: Reception latency (per unit of time) for one fixed tree and for one tree per source [13].

Besides a broadcast primitive that allows dissemination of information among nodes, many distributed/parallel applications require causal order of messages: broadcast messages must be delivered to all other nodes by respecting the causal relation of their respective broadcast events, i.e., the relation of cause and effect among messages of Lamport's *happened-before* relationship [22]. Introduced by Birman in the ISIS system [8], causal broadcast ensures that if two messages are causally related and have the same destination, then they are delivered to this node in their sending order. For instance, in a group discussion application, a causal broadcast protocol guarantees that no members of the group will see answers to a question before the question itself. It is worth emphasizing that causal message ordering is of prime interest to the design of many distributed applications. Examples of them are event notification systems [25], multimedia applications [7, 30], multi-user online games [16], systems that provide distributed replicated causal data consistency [4], distributed snapshots [1], distributed database [39], shared objects [29], publish/subscribe systems [13, 24], etc.

Operationally, direct communication between two processes should always be faster than indirect communication where messages are relayed via intermediate nodes, the famous *triangle inequality* end-to-end latency. However, disparity in the speed of communication links and network congestions can lead to *triangle inequality violation* (TIV) [2, 30]. It is worth noting that existing studies on TIV show that they are widespread and frequent [23, 41]. In case of TIV, node receives messages out of the causal order and, therefore, some delay and additional treatment are imposed before delivering them to the application in the correct order. Thereby, if a process p receives a message out of causal delivered order, this message should be held in and delivered to the application only after the missing messages are received and delivered by p to the application. For instance, in Figure 2, node 2 broadcasts message m_2. Upon delivering it, node 1 broadcasts message m_1 and just after delivering m_1, node 0 broadcasts message m_0, i.e., *broadcast* m_2 precedes *broadcast* m_1, which, in its turn, precedes *broadcast* m_0. However, at node 3, messages are received out of the causal order. Upon the reception of m_0, node 3 stores it in its local buffer and will deliver it only after receiving and delivering both m_2 and m_1.

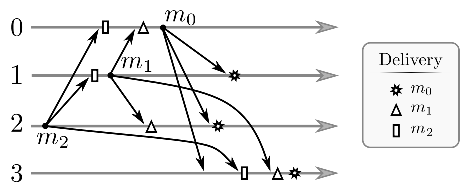


Figure 2: Propagation of three causally related messages and their reception and delivery orders.

In the present work, we propose a causal order broadcast protocol where, for disseminating a message, every node dynamically builds a spanning tree, rooted at itself, on top of *VCube* [14] which logically organizes nodes in a hypercube-like topology, presenting, thus, logarithmic properties. The tree rooted in each node is organized differently, i.e., the neighbors of a given node can vary according to the root of the tree. We should point out that, thanks to the *VCube* inference rules, the construction of different spanning trees present no overhead.

Our protocol also guarantees causal order of messages. Furthermore, it exploits the above mentioned TIV problem of networks for performance sake. The idea is that, even if the spanning trees are organized differently over *VCube*, parts of their paths may intersect, i.e. a node p can be the parent of a node k in different trees. Hence, as node p can deduce, without any overhead, every other node's spanning tree organization, p is aware of such intersections and then delay the forwarding, to one or more of its tree's children, of those messages whose causal dependencies it knows that these children can not satisfy them yet. As p is the responsible of forwarding the missing messages to those children, only upon receiving them, p combines these messages plus the delayed one into a single message and forwards it to the children in question. Therefore, the number of messages sent over the network is reduced, with no additional delay in their delivery latencies. We should point out that, contrarily to some existing approaches [5, 18, 20, 37] where messages are aggregated during a waiting time (implemented with timers and timeout) which entails extra delays to delivery latency, our approach does not induce any overhead neither degrades performance as it is based on the principle that the sending of a message to a node is worthless if the latter will not be able to deliver it. Interestingly that, due to such a reduction in the number of messages over the network, the average delivery latency is improved since there is less node contention.

We have implemented our causal broadcast protocol on top of the event-driven PeerSim simulator [27] and experiments confirm its effectiveness in terms of number of messages and transmissions, and average message delivery latency.

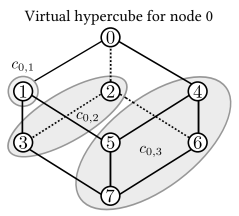
The rest of the paper is organized as follows. Section 2 introduces the hypercube-like topology *VCube*, followed by Section 3 that summarizes message causal ordering principle. Section 4 describes our message aggregation approach and our broadcast algorithm. Section 5 presents evaluation results from experiments conducted on PeerSim for different scenarios and metrics. Section 6 discusses some related work and, finally, Section 7 concludes the paper.

2 VCUBE

In *VCube* [14], a node i groups the other $N-1$ nodes in $d = \log_2 N$ clusters forming a d-*VCube*, such that the cluster number s ($s > 0$) has size 2^{s-1}. The ordered set of nodes in each cluster s is denoted by $c_{i,s}$ as follows, where $\oplus$ is the bitwise exclusive-*or* operator (xor).

$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i\oplus 2^{s-1},k} \mid k = 1,..,s-1$$

VCube is a distributed failure diagnosis system and it defines as the neighbors of a node i the first faulty-free node of each cluster s in $c_{i,s}$. Periodically, i tests the first node in the $c_{i,s}$ to check whether it is correct or faulty. Figure 3 shows node 0's hierarchical cluster-based logical organization of $N = 8$ nodes connected by a 3-*VCube* topology as well as a table which contains the composition of all $c_{i,s}$ of the 3-*VCube*. Let's consider node p_0 and that there are no failures. The clusters of p_0 are shown in the same figure. Each cluster $c_{0,1}$, $c_{0,2}$, and $c_{0,3}$ is tested once, i.e., p_0 only performs tests on nodes 1, 2, 4 which will then inform p_0 about the state of the other nodes of their respective cluster.



General $c_{i,s}$ table for 8 nodes

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figure 3: ***VCube*** **hierarchical organization.**

3 CAUSAL MESSAGE ORDERING

Causal order requires that the order in which messages are delivered to the application processes respects the causal relation between messages, i.e., the delivery of a message is dependent on the state of the system as viewed by the sender of the message at emission time. Therefore, causal order broadcast must ensure that if two messages are causally related and have the same destination they are delivered to the application in their sending order. In other words, if a process broadcasts a message m' after it has delivered another message m, then no process in the system will deliver m after m'.

The causal order relation between broadcasts and delivers of messages are based on the well-known Lamport's "happened before" [22] relation between events in distributed systems. Thus, denoting $\rightarrow$ the causal precedence or "happened before" relation, formally:

$$broadcast(m_1) \rightarrow broadcast(m_2) \Rightarrow delivery(m_1) \rightarrow delivery(m_2).$$

We should point out that the order imposed by causal broadcast is partial since non related messages might be delivered in different order by the processes.

As proposed in [34], our causal broadcast protocol uses logical vector clocks [15, 26], to track information about causal order. Every process i keeps a vector clock, denoted vc_i of size N. Each message m sent by i carries vc_i, where $m.vc_i[k]$ represents either the number of messages sent by i, if $k = i$, or the number of $k's$ broadcast messages delivered by i before it broadcasts m. For instance, considering the time diagram of Figure 2 with $N = 4$ nodes where $broadcast\ m_2 \rightarrow broadcast\ m_1 \rightarrow broadcast\ m_0$, $m_2.vc = (0, 0, 1, 0)$ and, thus, $m_1.vc = (0, 1, 1, 0)$, and $m_0.vc = (1, 1, 1, 0)$.

Upon receiving m from j, i must delay the delivery of m until (1) it has delivered all messages from j that precede m, and (2) it has delivered all messages delivered by j before the latter sends m. Formally:

$$\forall k \begin{cases} (1)\ m.vc_j[k] = vc_i[k] + 1, & \text{if } k = j \\ (2)\ m.vc_j[k] \leq vc_i[k], & \text{otherwise} \end{cases}$$

When process i delivers the message m sent by j, it updates its vector clock: $vc_i[j] = vc_i[j] + 1$.

It is worth remembering that Charron-Bost showed in [11] that causality can be characterized only by vector timestamp of size N.

4 CAUSAL AGGREGATION BROADCAST

In this section, we present the causal broadcast protocol with message aggregation that we propose. Firstly, we describe the adopted system model. Then, we explain how our protocol exploits both *VCube*'s organization to dynamically build broadcast spanning and the asynchronous nature of processes and communication channels in order to combine causally related messages into a single message, without degrading performance. Finally, we present the algorithm that implements our causal broadcast protocol.

4.1 System model and definitions

We consider a distributed system composed of a finite set of $\Pi = \{0, .., N-1\}$ nodes (users) with $N = 2^d$ processes, where $d > 0$ is the dimension of *VCube*. Each node has an unique identifier (id) and nodes communicate only by message passing. Each single node executes a task (process) and a user of the system corresponds to a node. Therefore, the terms node, user, and process are interchangeable in this work.

Nodes communicate by sending and receiving messages through bidirectional channels. The topology of the connected network (not necessarily fully-connected) must allow nodes to be logically organized as a hypercube interconnection network. Nodes do not fail and links are reliable. Thus, messages exchanged between any two processes are never lost, corrupted nor duplicated. The system is asynchronous, i.e., relative processor speeds and message transmission delays are unbounded.

We denote *source* of a message, the *id* of the node that broadcasts a message. We also distinguish between the arrival of a message (*reception*) at a process and the event at which the message is given to the application (*delivery*). Note that only the latter respects the causal order of broadcast messages, explained in the previous section.

4.2 Dynamic building of spanning trees

In our protocol, the broadcast of a message by i to all nodes is performed by dynamically building a spanning tree, rooted at i, on top of the virtual hypercube-like topology of *VCube*. In other words, it takes advantage of *VCube's* cluster hierarchy to build different trees, each comprising all nodes, but whose organization depends on the source. However, as we consider that nodes do not fail, *VCube's* failure diagnosis feature is not exploited (see future work in Section 7).

Consider $d = \log_2 N$ the dimension of *VCube* which is also the height ($h = d$) of the related spanning tree. For broadcasting a message m, node i sends m to the first node of each of its clusters $c_{i,s}, \forall s \leq h$, to which i is linked. Upon receiving m, each of these nodes j becomes the root of a subtree whose height is $h = s - 1$. Therefore, if j is not a leaf ($h \neq 0$), it applies the same sending procedure of i's and so on. For instance, based on the *VCube* of Figure 3, the spanning tree over which m_0 will travel due to its broadcast by node 0 is shown in the left side of Figure 4.

Auxiliary functions: In order to easily build spanning trees and detect intersection of their paths, we define the following functions, called by i, which exploit *VCube* virtual hypercube topology:

FirstChild(i,s): returns the first node in $c_{i,s}$ table (Figure 3), i.e., the node of c_i, s which is linked to i. For example, FirstChild(0,1) = 1, FirstChild(1,2) = 3, and FirstChild(1,3) = FirstChild(7,2) = FirstChild(4,1) = 5.

Cluster(i,j): returns the index s of the cluster of node i that contains node j, $(1 \leq s \leq \log_2 N)$. For instance, in Figure 3, Cluster(0, 1) = 1, Cluster(0, 2) = Cluster(0, 3) = 2, and Cluster(0, 4) = Cluster(0, 5) = Cluster(0, 6) = Cluster(0, 7) = 3.

Children(r,h): returns, with regard to the tree rooted at node r with height h, the first child of each cluster $c_{i,s}$ of i, $\forall s \leq h'$, where h' is the height of the subtree of i in the tree of r (see Algorithm 1). We denote this set the *children* of i in the spanning tree of r. The function is called by the broadcast protocol to either (1) obtain the children of i in the spanning tree of r or (2) to build spanning trees.

Algorithm 1 Children of i in r's tree

```
1: function Children(node $r$, height $h$)
2:     if $i = r$ then
3:         return FirstChild($i$, $s$) | $1 < s \leq h$
4:     else
5:         return Children(FirstChild(r,Cluster($r$, $i$)),Cluster(r, i) −1)
```

When i is equal to r, Children(r,h) simply returns its h children. Otherwise, the function recursively searches node i in the tree of r using the cluster of r where i is present. When the subtree rooted in i $(i = r)$ is found, its respective children are returned. For example, if node 4 wants to know its children in the tree rooted in node 2, it invokes Children$(2,3)$ which will recursively call Children$(6,2)$ $\rightarrow$ Children$(4,1)$ = $\{5\}$ (see right side of Figure 4).

Children(i,h) function is also used for the construction of spanning trees. In order to broadcast message m, by calling the function CO_Broadcast(m) (see Algorithm 2), i becomes the root of the spanning tree and sends m to its $\log_2 N$ children. Upon the reception of m, j, a child of i, becomes the root of a subtree of i's tree with height Cluster(j, i) −1. Note that the number of children of a node also decreases by one in relation to its parent's cluster. Hence, every node $k \in$ Children(j,Cluster(j,i) - 1), i.e., every child of j in relation to a tree where $j's$ parent is i, receives m from j and this procedure continues until m is received by all nodes that do not have children (leaves of the spanning tree). For instance, in order to broadcast message m_0 (see left side of Figure 4), node 0 calls Children$(0,\log_2 N)$ = $\{1,2,4\}$ and sends m_0 to them. Upon receiving m_0, node 1 does not forward m_0 since Children$(1,0) = \emptyset$, node 2 forwards it to Children$(2,1) = \{3\}$, and node 4 to Children$(4,2) = \{5,6\}$; Node 5 does not forward m_0 since Children$(5,0) = \emptyset$. Node 6 forwards it to Children$(6,1) = \{7\}$ while node 7 does not forward it since Children$(7,0) = \emptyset$.

4.3 Aggregating causally related messages

Although the spanning trees are organized differently, their nodes may have some common children, i.e., some parts of the paths of two messages may intersect at a node. By exploiting this spanning trees intersection feature, a node can delay, to one or more of its children, the forwarding of the messages whose some causal dependencies it knows that the children in question can not satisfy yet. Upon reception of these missing causally related messages, the node aggregates all of them into a single one and forwards the latter to the concerned child nodes. In other words, if (1) node i knows that m will not be able to be delivered by its child node k because i has not received/delivered yet some message m' that precedes m and (2) i is also responsible for forwarding m' to k, i will postpone sending m to k because k would not be able to deliver m. Node i will send all the missing messages m' and m to k aggregated in a single message only after receiving the former.

It is worth emphasizing that node i can deduce, without any overhead, the spanning tree of k. Furthermore, our approach does not entail any performance overhead or delivery latency degradation, i.e., even if the forwarding of m to k was delayed, such a postponement does not cause any extra delay in m's delivery by k.

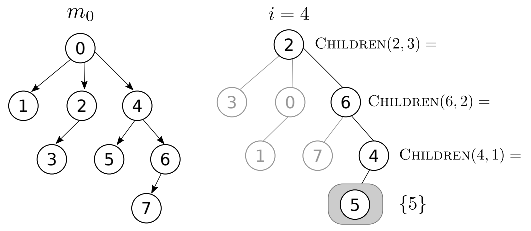


Figure 4: Function Children use to (left) build a spanning tree and (right) find a node's children in according to a given source.

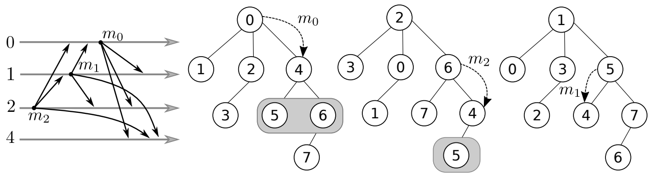


Figure 5: Example of spanning trees and intersection of tree paths

Let's consider Figure 5 where $m_2 \rightarrow m_1 \rightarrow m_0$. The broadcast of these messages by nodes 0, 2, and 1 dynamically builds different spanning trees, as shown in the right side of Figure 5 in a system with 8 nodes. Upon the reception of m_0, node 4 verifies that it has not either m_2 or m_1 yet. Thus, without the aggregation approach, node 4 would forward m_0 to its children in relation to the tree rooted in node 0, i.e., nodes 5 and 6. However, as observed in the figure, node 5 is 4's child in both m_0's and m_2's trees and node 4 knows it. Thereby, by applying our aggregation approach, m_0 is forwarded immediately to node 6, but not to node 5 because m_2 precedes m_0 and node 4 has not received m_2 yet. Upon reception of it, node 4 aggregates m_0 and m_2 within a single message and sends it to 5. Note that (1) node 4 does not wait for m_1 to send the aggregation message to 5 given that the latter is not a child of 4 in m_1's tree. In fact, node 5 is the parent of node 4 in m_1's tree; (2) if node 4 had received m_2 before m_0, the messages would not be aggregated to 5 since, in this case, upon reception of m_2, node 5 would be able to deliver it without depending on the reception of m_0.

4.4 Causal Broadcast Algorithm

Every node i keeps the following local variables:

- *vector_clock*: stores information about delivered messages;
- *vector_max*: keeps information about messages received by i that can be forwarded;
- *pending*: the set of messages which were received by i but that have not been delivered yet.

Algorithm 2 Causal broadcast at node i

```
1:  Init
2:    $vector\_clock[l] \leftarrow 0, \forall l = 0..N-1$
3:    $vector\_max[l] \leftarrow 0, \forall l = 0..N-1$
4:    $pending \leftarrow \emptyset$

5:  procedure CO_Broadcast(message $m$)
6:    $vector\_clock[i] \leftarrow vector\_clock[i] + 1$
7:    $vector\_max[i] \leftarrow vector\_clock[i]$
8:    $m.s \leftarrow i$
9:    $m.vc \leftarrow vector\_clock$
10:   CO_Deliver($m$)
11:   for all $k \in$ Children($i$, $\log_2 N$) do
12:     Send($\{m\}$) to $k$

13: upon receive $mSet$ from $j$
14:   for all $m \in mSet$ do
15:     $pending \leftarrow pending \cup \{m\}$
16:     while ($\exists\, m' \in pending \mid m'.vc[m'.s] = vector\_max[m'.s] + 1$) do
17:       $vector\_max[m'.s] \leftarrow vector\_max[m'.s] + 1$
18:     for all $k \in$ Children($i$,Cluster($i$, $j$) - 1) do
19:       $agg \leftarrow$ CheckAgg($k$, $m$)
20:       if $agg \neq \emptyset$ then
21:         Send($agg$) to $k$
22:   CheckDelivery( )

23: function CheckAgg($k$,$m$)
24:   $agg \leftarrow \emptyset$
25:   for all $m' \in pending \mid k \in$ Children($m'.s$, $\log_2 N$) do
26:     if $m'.vc[m.s] \geq m.vc[m.s]$ and $\nexists l : \left( \begin{array}{l} m'.vc[l] > vector\_max[l] \\ \text{and } k \in \text{Children}(l, \log_2 N) \end{array} \right)$ then
27:       $agg \leftarrow agg \cup \{m'\}$
28:   return $agg$

29: procedure CheckDelivery( )
30:   while $\left(\exists\, m' \in pending \mid \left( \begin{array}{l} (m'.vc[m'.s] = vector\_clock[m'.s] + 1) \\ \text{and } (m'.vc[k] \leq vector\_clock[k], \forall k \neq s) \end{array} \right)\right)$
      do
31:     CO_Deliver($m'$)
32:     $vector\_clock[m'.s] \leftarrow vector\_clock[m'.s] + 1$
33:     $pending \leftarrow pending \setminus \{m'\}$
```

When node i wants to broadcast message m, it calls the function CO_Broadcast(m) (lines 5-12), which increments i's own entry in the local vector clock (line 6), assigns the identifier of i and the value of its local vector clock to m, delivers m to itself, and forwards m (as a set that contains m) to i's $\log_2 N$ children.

Due to aggregation of messages, a node receives a set which contains one or more messages sent by its parent j in the tree (line 13). Each message m in the set is handled independently by the receiver i and included in the *pending* set (line 15).

Node i keeps track of message receptions from each other node, by maintaining the vector *vector_max* (lines 16-17) of size N. Each entry l of the vector keeps the sequence number of the last received message m' from l, such that all messages sent by l that precedes m' have also been received.

Then, considering the reception of m, i calls, for each of its child k in regard with m's spanning tree (line 18), the function CheckAgg(k, m) (lines 23-28) in order to aggregate all the messages in *pending* (including m), which do not have any missing pending precedence related to messages that must be sent to k by i (i.e., i is the parent of k with respect to the spanning tree of these messages): for every pending message m' where i is the parent of k in the spanning tree of m' (line 25), if m precedes m' (first condition of line 26) and i received all dependencies of m' to which i is responsible to forward to k (second condition of line 26), m' is added to the *agg* set. Otherwise, the forward of m' (which can be equal to m since the latter was added to the *pending* set) is postponed. If not empty, the set of aggregated message, which can be just m in the case of no possible aggregation, is then sent to k (line 21).

We note m_i^j the j^{th} message broadcast by i. Considering the *VCube* of Figure 3 and the spanning trees of Figure 4, let's suppose that node 2 broadcasts 3 messages and node 0 broadcasts 1 message such that: *broadcast* m_2^1 $\rightarrow$ *broadcast* m_2^2 $\rightarrow$ *broadcast* m_2^3 $\rightarrow$ *broadcast* m_0^1 and all messages have been received by 4, except m_2^2. In this case, $vector\_max[2] = 1$ even if m_2^3 was received. Since $m_0^1.vc[2] = 3$, the conditions of line 26 are satisfied only to m_2^1, that will be sent to node 5. On the other hand, upon reception of m_2^2, $vector\_max[2] = 3$, the conditions will be true for m_2^2, m_2^3, and m_0^1 which will be aggregated into a single message and sent to node 5. Such an aggregation takes place only for node 5. Node 4 directly sends m_0 to node 6 upon its reception.

The first condition of line 26 of CheckAgg(k, m) function is necessary in order avoid sending twice the same message. Let's take a second example where *broadcast* m_2^1 $\rightarrow$ *broadcast* m_0^1 $\rightarrow$ *broadcast* m_0^2 and that node 4 receives m_0^1, m_0^2, m_2^1 in this order. Upon reception of m_0^1, node 4 forwards it to 6 but not to 5 and, thus, m_0^1 is held in ($pending = \{m_0^1\}$). The same happens upon reception m_0^2 ($pending = \{m_0^1, m_0^2\}$). However, if the first condition was not included in line 26, m_0^1 would be sent again to node 6 since the second condition is satisfied. Upon reception of m_2^1, node 4 will send the 3 messages aggregated into a single one to node 5.

Lastly, for each message $m' \in pending$, i delivers all messages whose delivery is possible following the reception of m (function CheckDelivery, lines 29-33). A message can be delivered provided that the two conditions described in Section 3 are satisfied. Once a message is delivered, it is removed from *pending*. Note that the delivery of one message can trigger the delivery of other messages. This explains why all current remaining messages in *pending* are re-checked until no more message is delivered (line 30).

5 EXPERIMENTAL RESULTS

We have implemented the proposed causal broadcast algorithm on the top of the event-driven PeerSim simulator [27] and have then conducted several experiments with different configuration scenarios.

For sake of clarity, we denote *message* the data message of the application/user to be broadcast and *packet* the message of the broadcast protocol. A *packet* can, thus, aggregate several *messages*.

Based on the packet-switched network delay model of [21], we consider that each packet sent by a node to another consumes $t_{pc} + t_q + t_t + t_{pp}$ units of time ($u.t.$): t_{pc} accounts for the processing time of a message by a node, e.g., checksum verification, aggregation and routing decisions; t_q is the time a message must wait in the sending queue before being transmitted; t_t is the time necessary to transmit all bits of the packet to the link, and t_{pp} expresses how long it takes for a packet to transverse the link and reach the destination node. Assuming that there is no broadcast mechanism available in the system, if a message is sent to multiple destinations, a copy of it is inserted in the sending queue for each of the destinations.

We use a maximum transmission unit (MTU) of 1500 bytes, where 20 bytes represent the packet header (the minimum value used by the Internet Protocol [31]). The size of a message was set to 50 bytes, similarly to the payload size of control messages or messages carrying monitoring information. As messages are gradually aggregated into a packet, if the current size of the packet reaches the MTU size, not necessarily with all messages that must be aggregated into, our protocol implementation sends the packet and stores the missing messages in a new packet.

The number of nodes N vary from 8 up to 1024, in a power of two. Each simulation was executed 30 times and we present the average values. Nodes broadcast a new message in random time given by a Poisson distribution with interval rate $\lambda = 1000\ u.t.$ while the propagation time t_{pp} of a message follows a normal distribution with mean value $\mu = 100\ u.t.$ and standard deviation $\sigma = 25\ u.t.$ Still, based on [33], we set $t_{pc} = t_t = 1\ u.t.$, whereas the time a message stays queued (t_q) is a function of the rate of incoming/outgoing messages and can vary for each message. The following metrics are considered:

- *Number of packets:* the overall number of packets exchanged between nodes.
- *Number of messages per packet:* maximum number of messages that nodes aggregate into a single packet.
- *Size of packets:* size of the packet header plus $k \times$ (size of the vector clock plus the size of the message), where k is the number of aggregated messages.
- *Reception (resp., delivery) latency:* the time a message takes from its broadcast till it is received (resp., delivered) by a node.
- *Number of buffered messages:* number of messages, received by a node, which are held in before being delivered to the application.

Without loss of correctness in capturing causal order, we have implemented the vector clock compression algorithm proposed by Birman [9]. When broadcasting a new message, instead of including in it the N entry values of its current vector clock, a node includes just the values of those entries that have been modified since the last broadcast by the node.

5.1 Number of packets

A straightforward consequence of message aggregation is the reduction in the number of packets that transverse the links. In the simulations, for each execution, each node broadcasts one message (i.e., all the nodes are source). Hence, for a system with N nodes with no aggregation, the total number of packets sent over the network is $N \times (N-1)$, as each spanning tree has $N-1$ links. Table 1 shows the number of sent packets with and without aggregation. With aggregation, the greater the number of source nodes, the longer the paths and the higher the number of different paths (due to the organization of the trees), path intersections, and the possibility of causal relation between messages. Therefore, the number of message aggregations increases as well, leading, for instance, to in average 28.8% less transmissions with 1024 nodes.

Even if a great number of messages are not combined to others into a packet, the percentage of aggregation causes a substantial reduction in the number of packets traveling through the network, specially with 1024 nodes. This fact impacts other metrics, as discussed hereafter in this section.

Table 1: Average number of sent packets.

Nodes	*No aggr.*	*Aggr.*	*% of reduction*	*% of aggr.*
16	240	232	3.33	3.02
32	992	919	7.36	5.77
64	4032	3513	12.87	9.05
128	16256	13759	15.36	11.04
256	65280	49262	24.54	15.41
512	261632	191528	26.79	16.70
1024	1047552	745943	28.79	19.14

Another interesting metric to evaluate is the number of packets that actually contain more than one message. The last column of Table 1 shows that from 3% up to 19% of the packets have more than one message while Table 2 gives, for a scenario with 256 nodes, the percentage of the overall packets (second column) that have a given number of messages. As we can observe, 84.59% of the packets have no aggregation at all and those with two and three messages represent 9.16% and 3.12% of the transmitted packets, respectively. Likewise, considering only the packets with more than one message (third column), those with two or three messages account for 79% of these packets. On the other hand, only 1.2% of all packets carries more than 5 messages (up to the limit where four packets have aggregated 15 messages each).

Table 2: Distribution of the number of messages per packet for a 256 node scenario.

Messages per packet	*% of all packets*	*% of aggr. packets*
1	84.59	
2	9.16	59.46
3	3.12	20.25
4	1.26	8.18
5	0.66	4.31
(5,15]	1.2	7.8

5.2 Size of messages and packets

Besides the 20-byte header of a packet, every message included in the packet is associated with a vector clock. By applying Birman's compression algorithm in order to reduce the size of the vector clock, a message sent by i includes only the tuples $(k, vc_i[k])$, $0 \leq k < N$ such that $vc_i[k]$ has changed since the last broadcast of i. Each modified entry is represented by 4 bytes.

We have evaluated the size of messages' vector clocks and the number of bytes sent over the network in a scenario with 256 nodes. Note that since the size of a packet is bounded to 1500 bytes, an aggregated message may require more than one packet (each one with a 20-byte header).

Considering the aggregated approach, Figure 6 shows the percentage of messages whose vector clock carries a given number of dependencies. With no aggregation, the simulation presented the same behavior with a variation of up to 2.73% in the results.

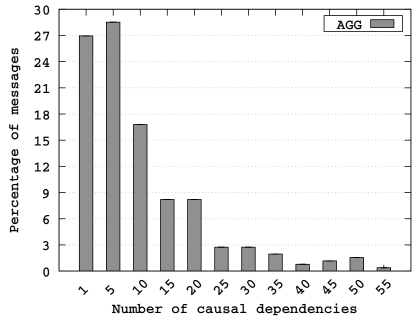


Figure 6: Distribution of the number of causal dependencies of a message in a scenario with 256 nodes with aggregation.

We observe in Figure 6 that 27% of the messages have no causal precedence, i.e., each of them carries only its own entry in the vector clock. However, despite the small size of their respective vector clocks, these messages cannot be aggregated by our approach since the latter only combines messages with causal relation. For the remaining messages, 28.5% (resp., 18.4%) of them contain no more than 4 (resp., 9) causal dependencies, and this percentage continues to drop till only 1 message which depends on 54 others. As each entry requires 4 bytes, 23% of the messages (those with more than 12 causal dependencies) spend more space for storing vector clock entries than the actual data from the application whose size is 50 bytes. On the other hand, the greater the number of vector clock entries, the more information gathered about causal order, which can result in more message aggregation.

Table 3: Distribution of the packets according to their size in bytes, for a scenario with 256 nodes.

Size (bytes)	*% of packets (Aggr.)*	*% of packets (No Aggr.)*
< 100	52.66	63.67
(100,200]	31.28	30.30
(200,300]	7.62	6.03
(300,400]	3.91	0.00
> 400	4.54	0.00

Another consequence of message aggregation is a better use of the available packet size and less packet header overhead. Each individual packet has a 20-byte header, no matter the number of messages it carries. Table 3 shows the distribution of different packet sizes in the same 256 node scenario with and without aggregation. The main observation is that when using our aggregation technique 8.45% of the sent packets are bigger (> 300 bytes) than all packets sent without aggregation. Using aggregation there is also a reduction in the number of packets of small sizes (< 100 bytes), specially because some messages which would be sent alone are grouped with others causally related ones into a single packet. Closer to the maximum packet size, aggregation presents only 0.29% of the packets with more than 1400 bytes. The reason for this low percentage is that a packet is forwarded whenever it is not possible to include one more message in it due to lack of space, which happened to 117 out of 49262 packets in the simulation analyzed in the table.

5.3 Reception and delivery latencies

We consider two different latencies metrics : (1) *reception* latency of a message is the time interval comprised from the broadcast of the message till it arrives at the destination node; (2) *delivery* latency of a message is given by the reception latency plus the *queuing time*, i.e., the additional time a message is held in at the destination node from its reception time till it is delivered to the application. For the experiments, every node broadcasts one message.

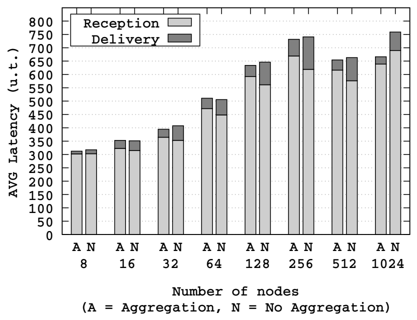


Figure 7: Average reception and delivery latencies.

Figure 7 depicts the average reception and delivery latencies with and without aggregation. A first observation concerns the variation in the reception latency when the number of nodes increases. Even if the number of nodes increases 128-times (from 8 up to 1024), the average reception latency is just 2.1 times higher with aggregation and 2.2 times without it (maximum increase of 3.9 and 4.1 times, respectively). This near-logarithmic behavior can mainly be explained by the use of spanning trees to broadcast messages.

As expected, the postponement of the forwarding of messages whose dependencies are missing leads to higher reception latencies. Therefore, in the same figure, we observe that reception latency is higher with aggregation when compared to no aggregation (except for 1024 nodes) and, as the number of nodes increases, so does the average reception latency. Furthermore, as discussed in Section 5.1, aggregation rate increases with the number of nodes. For instance, with 256 nodes, aggregation poses a reception latency in average 8.1% higher compared to the same scenario with no aggregation. The different behavior with 1024 nodes is related to the number of messages that the system must deal with. In such a scenario, average reception latency with aggregation is 7.4% smaller because the average time of message forwarding postponement becomes smaller than the overhead in time necessary to send packets containing a single message. On the other hand, with no aggregation, packets stay in average 53.4% more time in the sending queue before their sending request is processed. Such waiting time is around 50 $u.t.$, which is compliant with the difference in the same figure for the reception latency of the two approaches with 1024 nodes.

Relating to delivery latency, the results of the figure confirm our statement that delaying the forwarding of causal related messages does not degrade delivery latency but, actually, reduces it when compared to no aggregation. In networks up to 512 nodes, delivery latency difference with and without aggregation varies up to 3.2% (32 nodes), explained by the normally distributed t_{pp} (propagation time per hop). Hence, the only difference in time that aggregated messages can suffer from when compared to no aggregated messages is related to propagation or queuing times of the packets which contain them. In the scenario with 1024 nodes, there exists a greater difference between the two approaches: our causal aggregation broadcast delivers messages in average 12.2% faster. The reason is that, as previously discussed, messages are received faster with aggregation, and possibly several messages in one packet.

Another remark about Figure 7 concerns the time that messages are held in (*pending*) before being delivered to the application. Regardless the number of nodes, messages are, in average, held in longer with no aggregation than with aggregation. This difference ranges from 5% (16 nodes) up to 53.4% (1024 nodes) since, with aggregation, upon the reception, more messages can be delivered immediately, reducing, therefore, the time messages are held.

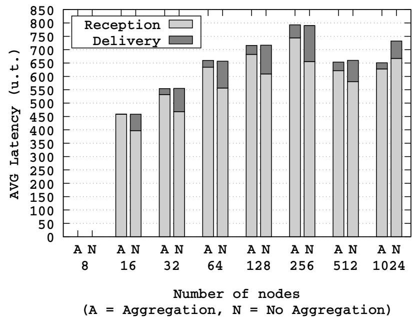


Figure 8: Average reception and delivery latencies for messages of packets that aggregated two messages.

In order to profile the impact of message aggregation in latency, for different network sizes, we consider only those packets that have two aggregated messages and compared them to the individual transmissions of the corresponding messages without aggregation, in exactly same scenarios. We can observe in Figure 8, that for networks with 8 nodes, there is no message aggregation. However, for the other network sizes, delivery latencies are the same (except for 1024 nodes for the reasons discussed before) since, for the aggregation approach, the reception latency increases (in average up to 13.6%) but the delay to delivery a message decreases (59% for 512 nodes, reaching up 74% for 64 nodes).

5.4 Distribution of pending messages

Figure 9 shows the number of buffered messages by each node (set *pending* of Algorithm 2) in a scenario with 1024 nodes, each of them broadcasting one message. With aggregation, more than 50% of the nodes (569) buffer at most only 50 messages, while with no aggregation the ratio drops to less than 25% of the nodes. On the other hand, there exist only 24 nodes, in the aggregation case, and 212 nodes, without it, that keep at some moment more than 250 messages, Such a difference is due to our aggregation approach that avoids unnecessary buffering.

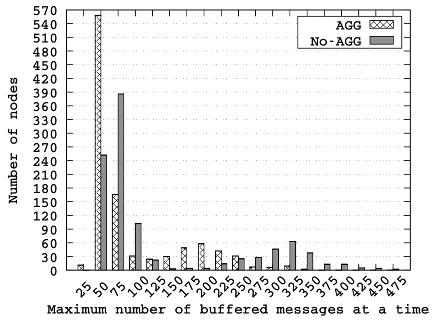


Figure 9: Distribution of the maximum number of messages buffered per node, with and without aggregation.

In Figure 10, we can observe how each node collaborates in the aggregation process in a simulation with 1024 nodes, for two message sizes: 50 and 5 bytes. For both sizes, the distribution of the maximum number of messages aggregated in a packet per node seems to follow a normal distribution although, for the 5-byte size, there are more nodes which have aggregated a higher number of messages per packet. The reason for such a difference is the limitation in the number of messages that a packet can hold: the smaller the size of the message, the greater the number of messages that it can keep. Every node participates in the aggregation process and most of them with a close maximum aggregation size. For the 50-byte size, 95.5% of the nodes have aggregated at some moment between 7 and 11 messages while no node has aggregated more than 14 messages. For the 5-byte size, 833 nodes (81.3%) have aggregated between 8 and 12 messages.

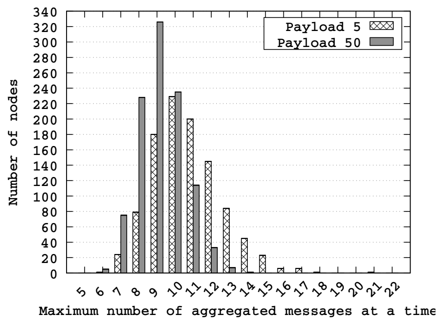


Figure 10: Distribution of the maximum number of messages aggregated per node for messages of size 5 and 50 bytes.

5.5 One tree versus multiple trees

We have also conducted some simulations where all nodes broadcast messages through the same single spanning tree rooted at node 0. Furthermore, in order to simulate out-of-order message receptions, we have varied latencies of 0's spanning tree links each time a link is used, following a Gaussian distribution, as if messages took differently routes at each broadcast. The first remark is that node 0 becomes a bottleneck and reception latencies have the same behavior of Figure 1, presented in the Introduction. We have also evaluated the average number of aggregated messages and number of delayed messages (in buffer waiting for delivery) for both approaches (*unique* and *multi* trees). The results are gathered in Figure 11. With a unique tree, there are at least 86% (resp.,84%) fewer aggregations (resp., delayed messages), performed by 32 nodes (resp., 128 nodes). Such results confirm that our aggregation approach performs better with one tree per source compared to a single one because the former naturally exploits existing delays induced by different paths. The smaller number of aggregations for single tree is due to the fact that out-of-order message receptions at nodes is limited, in this case, only to latency variations of the common links over which all messages travel, which also justifies the reduced time messages are held in before delivery. For unique tree with 1024, the number of aggregations decrease due to contention in the root of the tree.

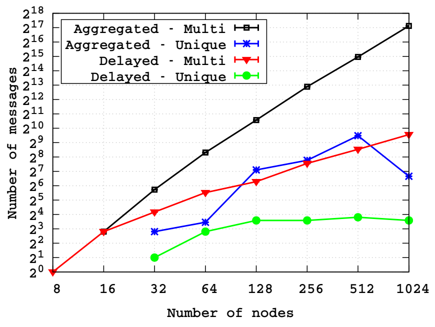


Figure 11: Number of aggregated and delayed messages with one spanning tree per source and a single rooted spanning tree (logarithmic scale).

6 RELATED WORK

Over the years, several attempts have been proposed to reduce the amount of information necessary to ensure causal order delivery either by modifying the underlying topology [3, 10] or using different types of logical clocks [28, 32, 38].

In [3], the nodes of a system are logically organized in a tree in such a way that a node directly communicates only with a few other nodes. Hence, a node uses just the information about the receptions from those nodes to respect causal delivery and to timestamp its own messages. Blessing et al. [10] go further by eliminating the use of meta-data carried by the messages. They exploit application-defined causal order [6] in order to organize the actors (processes) of an application into a tree topology that guarantees causal order delivery. The path used by the “causing” message must somehow be included in the path of the “caused” ones. However, the model for organization of the tree is time-costly and application-dependent.

Singhal et al. [38] observe that over a series of successive events at a same process, only a few entries in the vector clock are likely to be modified. Based on such a behavior, in their approach, a process *i* sends to a process *j* only the entries of its vector clock that changed since the last message *i* sent to *j*. Their approach reduces the size of the information related to causal order included in the messages, but fails to characterize causality when messages from the same source arrive out of order. Prakash et al. proposed in [32] the *causal barrier*, a structure which keeps information only about direct dependencies. The key advantage of the *causal barrier* is that, since the tracking of message causal ordering does not depend on nodes' identifiers (per node vector), the broadcast protocol might scale and tolerates more easily changes in system membership. A new type of clock based on plausible clocks [40] is presented by Mostefaoui et al. [28]. Motivated by the observation that, for some scenarios, a system can deliver most of the messages in the causal order without any explicit control, the authors propose a probabilistic approach to reduce the size of vector clocks at the cost of a small rate of errors in the causal delivery. Even if all these logical clocks reduce the size of causal order information included in messages, they are not suitable for the implementation of our causal broadcast protocol since our aggregation mechanism aims at combining as much as possible causally related messages into a single message. The ideal is, therefore, that nodes have knowledge about the entire and exact chain of causal dependencies of a received message, and not incomplete or partial ones.

Many distributed systems and application on different research areas such as Peer-to-Peer Overlay Networks [17, 18, 37], Internet of Things (IoT) [20, 35], Wireless Sensor Networks (WSN) [5], Parallel Discrete Event Simulation (PDES) [12], among others, exploit the possibility of combining/aggregating information or messages into a single message in order to reduce communication cost improving, therefore, performance.

In [18], a structured peer-to-peer routing protocol combines several lookup messages into a single one with the goal of reducing the average number of hops of messages. In [17], the circular logical identifier space of the P2P system is divided into slices, coordinated by a leader node. The latter collects all membership change notifications sent from the nodes of its slice during a period of time and, aiming at reducing bandwidth usage, aggregates them into a message before sending them to the other slice leaders. Similarly, in [37], a message bundling technique improves network throughput by reducing the number of packet transmissions and mitigates the load of nodes on an overlay that forwards messages.

Chetlur et al. [12] propose to optimize the communication subsystem of Time-Warp simulators, which suffer from high overheads due to frequent communication, by dynamically aggregating, within a single message, those messages with the same destination that must be sent in close temporal proximity. Considering that static overhead is independent of message size, the authors state that it is more efficient to communicate two data items using a single physical message than using two separate messages.

The main motivation of data aggregation in WSN and IoT is to provide energy savings through reducing the number of message transmission. However, it may have an impact on other performance metrics such as latency, load processing, or fault tolerance.

Our message aggregation strategy differs from all the above ones since it does not rely on waiting time for performing message aggregation which may increase delivery latencies. It exploits possible path intersections of different distributed spanning trees along with the causal relation among messages in order to aggregate them. Thus, even if the forwarding of a message is postponed there is no additional delay in its delivery by the destination nodes.

7 CONCLUSION

We have presented a causal broadcast protocol where nodes are logically organized in a hypercube-like topology and broadcast messages are disseminated over dynamically built spanning trees rooted at the respective source nodes. The multiple spanning trees avoid root contention bottleneck problem of existing protocols that broadcast using a single root tree. By exploiting both the TIV problem and intersection of trees' paths, a node may buffer out-of-order messages and forward them to its child (or children) in the related spanning tree only when they become deliverable. Such an aggregation mechanism does not induce any overhead since the sending of a message to a child node is worthless if the latter will not be able to deliver it upon reception.

Evaluation results of our experiments on the simulator PeerSim corroborate the communication effectiveness of our multi spanning trees message aggregation approach. Combining messages into a single packet reduces packet traffic as well as average delivery latencies since there is less node contention. Moreover, when receiving a packet with more than one message, a node is more likely to deliver them all to the application faster, reducing, therefore, non deliverable messages held in time.

Future directions in our work include adding reliability to our causal broadcast protocol (CBCAST) in order to tolerate node failures, exploiting $VCube's$ diagnosis features to detect faults, as well as handling dynamic node membership.

ACKNOWLEDGMENTS

This work was supported by a scholarship (PhD – GDE) from CNPq (Brazil) and by *Fundação Araucária/SETI* (Brazil) under the project 45112, grant 144/15.

REFERENCES

[1] A. Acharya and B. R. Badrinath. 1992. Recording Distributed Snapshots Based on Causal Order of Message Delivery. *Inf. Process. Lett.* 44, 6 (1992), 317–321.

[2] F. Adelstein and M. Singhal. 1995. Real-time causal message ordering in multimedia systems. In *Proceedings of 15th ICDCS*. 36–43.

[3] N. Adly and M. Nagi. 1995. Maintaining Causal Order in Large Scale Distributed Systems Using a Logical Hierarchy. In *Proc. IASTED Int. Conf. on Applied Informatics*. 214–219.

[4] M. Ahamad, P. W. Hutto, and R. John. 1991. Implementing and programming causal distributed shared memory. In *11th ICDCS*. 274–281.

[5] K. Akkaya, M. Demirbas, and R. S. Aygun. 2008. The Impact of Data Aggregation on the Performance of Wireless Sensor Networks. *Wirel. Commun. Mob. Comput.* 8, 2 (Feb. 2008), 171–193.

[6] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. 2012. The Potential Dangers of Causal Consistency and an Explicit Solution. In *3rd ACM Symp. on Cloud Comp. (SoCC '12)*. 22:1–22:7.

[7] R. Baldoni, M. Raynal, R. Prakash, and M. Singhal. 1996. Broadcast with Time and Causality Constraints for Multimedia Applications. In *EUROMICRO*. IEEE Computer Society, 617–624.

[8] K. Birman and T. A. Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.* 5, 1 (1987), 47–76.

[9] K. Birman, A. Schiper, and P. Stephenson. 1991. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.* 9, 3 (Aug. 1991), 272–314.

[10] S. Blessing, S. Clebsch, and S. Drossopoulou. 2017. Tree Topologies for Causal Message Delivery. In *Proceedings of the 7th ACM SIGPLAN Intl. Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2017)*. 1–10.

[11] B. Charron-Bost. 1991. Concerning the Size of Logical Clocks in Distributed Systems. *Inf. Process. Lett.* 39, 1 (July 1991), 11–16.

[12] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey. 1998. Optimizing communication in Time-Warp simulators. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*. 64–71.

[13] J. P. de Araujo, L. Arantes, E. P. Duarte Jr., L. A. Rodrigues, and P. Sens. 2017. A Publish/Subscribe System Using Causal Broadcast over Dynamically Built Spanning Trees. In *29th Intl. Symp. on Computer Arch. and High Perf. Comp. (SBAC-PAD)*. 161–168.

[14] E. P. Duarte Jr., L. C. E. Bona, and V. K. Ruoso. 2014. VCube: A Provably Scalable Distributed Diagnosis Algorithm. In *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '14)*. 17–22.

[15] C. J. Fidge. 1988. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conf.* 55–66.

[16] J. S. Gilmore and H. A. Engelbrecht. 2012. A Survey of State Persistency in Peer-to-Peer Massively Multiplayer Online Games. *IEEE Transactions on Parallel and Distributed Systems* 23, 5 (May 2012), 818–834.

[17] A. Gupta, B. Liskov, and R. Rodrigues. 2004. Efficient Routing for Peer-to-peer Overlays. In *Proceedings of the 1st Conf. on Symp. on Networked Systems Design and Implementation (NSDI'04)*. USENIX Association, Berkeley, CA, USA, 14.

[18] N. Hidalgo, L. Arantes, P. Sens, and Xavier X. Bonnaire. 2010. An Aggregation-Based Routing Protocol for Structured Peer to Peer Overlay Networks. In *AP2PS 2010 - 2nd Intl. Conf. on Advances in P2P Systems*. 76–81.

[19] K. Kim, S. Mehrotra, and N. Venkatasubramanian. 2010. FaReCast: Fast, Reliable Application Layer Multicast for Flash Dissemination. In *Middleware (Lecture Notes in Computer Science)*, Vol. 6452. Springer, 169–190.

[20] A. Koike, T. Ohba, and R. Ishibashi. 2016. IoT Network Architecture Using Packet Aggregation and Disaggregation. In *2016 5th IIAI Intl. Congress on Advanced Applied Informatics (IIAI-AAI)*. 1140–1145.

[21] J. F. Kurose and K W. Ross. 2012. *Computer Networking: A Top-Down Approach* (6th ed.). Pearson.

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

[23] C. Lumezanu, R. Baden, N. Spring, and B. Bhattacharjee. 2009. Triangle inequality variations in the internet. In *Internet Measurement Conf.* ACM, 177–183.

[24] C. Lumezanu, N. Spring, and B. Bhattacharjee. 2006. Decentralized Message Ordering for Publish/Subscribe Systems. In *Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 162–179.

[25] C. H. Lwin, H. Mohanty, and R. K. Ghosh. 2004. Causal Ordering in Event Notification Service Systems for Mobile Users. In *ITCC (2)*. 735–740.

[26] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*. 215–226.

[27] A. Montresor and M. Jelasity. 2009. PeerSim: A scalable P2P simulator. In *2009 IEEE Ninth Intl. Conf. on Peer-to-Peer Computing*. 99–100.

[28] A. Mostéfaoui and S. Weiss. 2017. *A Probabilistic Causal Message Ordering Mechanism.* Springer Intl. Publishing, Cham, 315–326.

[29] M. Perrin, A. Mostefaoui, and C. Jard. 2016. Causal Consistency: Beyond Memory. In *Proceedings of the 21st ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '16)*. Article 26, 12 pages.

[30] C. Plesca, R. Grigoras, P. Queinnec, G. Padiou, and J. Fanchon. 2006. A coordination-level middleware for supporting flexible consistency in CSCW. In *14th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing*. 6 pp.–.

[31] J. Postel. 1981. *Internet Protocol.* STD 5. RFC Editor. http://www.rfc-editor.org/rfc/rfc791.txt

[32] R. Prakash, M. Raynal, and M. Singhal. 1996. An efficient causal ordering algorithm for mobile computing environments. In *Proceedings of 16th Intl. Conf. on Distributed Computing Systems*. 744–751.

[33] R. Ramaswamy, N. Weng, and T. Wolf. 2004. Characterizing network processing delay. In *GLOBECOM '04*, Vol. 3. 1629–1634 Vol.3.

[34] A. Schiper, J. Eggli, and A. Sandoz. 1989. A New Algorithm to Implement Causal Ordering. In *3rd Intl. Workshop on Distributed Algorithms*. 219–232.

[35] L. Schmidt, N. Mitton, D. Simplot-Ryl, R. Dagher, and R. Quilez. 2011. DHT-based distributed ALE engine in RFID middleware. In *2011 IEEE Intl. Conf. on RFID-Technologies and Applications*. 319–326.

[36] F. B. Schneider, D. Gries, and R. D. Schlichting. 1984. Fault-Tolerant Broadcasts. *Sci. Comput. Program.* 4, 1 (1984), 1–15.

[37] K. Shudo. 2017. Message bundling on structured overlays. In *2017 IEEE Symp. on Computers and Communications (ISCC)*. 424–431.

[38] M. Singhal and A. Kshemkalyani. 1992. An efficient implementation of vector clocks. *Inform. Process. Lett.* 43, 1 (aug 1992), 47–52.

[39] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*. 172–183.

[40] F. J. Torres-Rojas and M. Ahamad. 1999. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing* 12, 4 (1999), 179–195.

[41] G. Wang, B. Zhang, and T. S. Eugene Ng. 2007. Towards network triangle inequality violation aware distributed systems. In *Internet Measurement Conf.* 175–188.

[42] Y. Wang, J. Fan, X. Jia, and H. Huang. 2012. An algorithm to construct independent spanning trees on parity cubes. *Theor. Comput. Sci.* 465 (2012), 61–72.