Localized Reliable Causal Multicast

(extended abstract of the MSc dissertation)

Válter Emanuel Trecitano da Costa Santos
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Prof. Luís Eduardo Teixeira Rodrigues

***Abstract*—This paper addresses the problem of offering reliable causal multicast in a setting where nodes are organized in an overlay network and use this network to disseminate information among each other. The use of overlay networks for this purpose is widely used when the number of nodes is large. For instance, many publish-subscribe system use an overlay of message brokers to support the exchange of information among publishers and subscribers. To the best of our knowledge, previous multicast algorithms for overlay networks either do not enforce causal order or, in order to do so, require nodes to keep metadata (for instance, sequence numbers) for all senders and are, therefore, inherently non-scalable. In this paper we propose a novel localized algorithm to implement reliable causal multicast, where each node is only required to keep metadata regarding nodes in its neighbourhood (with a radius that is a function of the number of faults that need to be tolerated). Experimental results show that our algorithm can achieve significant improvements over non-localized alternatives, and can even outperform localized algorithms that do not offer causal order.**

I. Introduction

Reliable multicast is a classical problem in distributed systems, and a fundamental building block of distributed fault-tolerant systems [1], [2], [3]. Typically, reliable multicast protocols offer not only delivery guarantees but also ordering properties. Relevant ordering properties in this context are FIFO order [4], [5] and causal order [1], [6], [7]. The problem is relevant for systems of all scales, from small-scale systems with less than a dozen replicas (such as a replicated server in a single data center) to large-scale systems with hundreds or thousands of participants (such as large scale publish-subscribe system). This paper addresses the problem of ensuring reliability and causal order for large scale systems.

Techniques to implement reliable multicast with causal order for small-scale systems are now well studied, and several widely available systems, both academic and commercial, offer this service [1], [8], [9]. However, providing causal order for large scale systems is much more challenging. This happens because causal order protocols are required to maintain metadata (such as sequence numbers) to keep track of causal dependencies among messages. This metadata usually has the form of set of vector clocks [1] or a matrix clocks [6], that have an entry for each sender in the system. All these approaches are suitable for systems with small numbers of nodes but are inherently non-scalable. Some approaches offer causal order with less metadata, but are not able to deliver messages unless all nodes periodically send messages [10] or require nodes to be reliable and not fail (by replicating all nodes in the system) [7]. Again, none of the later approaches is practical in large-scale systems. This paper explores a different alternative, that requires each node to maintain metadata for a constant number of nodes in the system, regardless of the system scale. To the best of our knowledge this is the first localized algorithm to enforce causal order.

This work assumes that the nodes in the system are organized in an overlay network, that can be modelled as an acyclic graph. The idea of organizing nodes in an overlay network to support multiple variants of group communication in large-scale systems has been extensively applied, particularly in the implementation of publish-subscribe systems, where the communication among publishers and subscribers is supported by an overlay of message brokers [11], [4]. In the overlay network, each node only maintains direct links with a small number of *neighbours* and should not be required to have information regarding the global system membership. An algorithm that requires each node to know only its k-neighbourhood (where k is the maximum distance between the node and any of its neighbours) is said to be *localized*. Previous work has proposed a localized algorithm for reliable propagation of information in publish-subscribe systems [5] but it only supports FIFO order and, furthermore, restricts the communication pattern, by preventing each publisher from having more than one message in transit at any given time. This paper's resulting algorithm, which has been named LoCaMu (Local Causal Multicast), is substantially more powerful because it offers causal order and allows multiple messages to be in transit, fully exploring message pipelining in the overlay. In LoCaMu, in order to tolerate f faults in a given part of the overlay, each node is required to maintain metadata for nodes in its $(2f+1)$-neighbourhood.

LoCaMu has been implemented and simulations have been used to assess its performance against other algorithms that can offer causal order and also against [5], which does not enforce causal order but, as LoCaMu, uses a localized approach. Experimental results show that by keeping the metadata required to enforce causality localized, the system can scale without increasing the size of the metadata, even when several nodes are publishing messages concurrently, unlike the related work. Thus, LoCaMu can be successfully used in large scale systems where, due to the size of their metadata, previous works cannot be implemented in practice.

II. Related Work

The literature on reliable causal multicast is extensive and covers many different designs that target a wide range of settings. On the one hand, we can find systems designed to support the development of fault-tolerant systems based on the replicated state machine model and its variants [12]. These systems typically provide strong guarantees of reliability, and different ordering policies, including FIFO, causal, and total order. These strong properties come at a cost: most of these systems use algorithms where nodes communicate directly with each other, run multiple rounds of all-to-all coordination, and maintain control information that grows linearly, or even quadratically, with the system size [1], [6]. These algorithms are, therefore, inherently non-scalable, even when using sophisticated techniques to reduce the amount of control information exchanged [1]. On the other hand, we have systems designed to support the dissemination of information among very large number of participants, often relying on peer-to-peer overlays [11] or epidemic protocols [13], which are known to be scalable but that offer only probabilistic or even no guarantees of reliability and ordering.

In this paper we are interested in algorithms that can offer strong guaranties while still being able to scale. A noteworthy application of algorithms with these characteristics are publish-subscribe systems [14], where we can have large numbers of publishers and subscribers, that exchange multicast messages, and for which reliability and ordering are relevant properties [4]. Although there are several possible strategies to implement publish-subscribe systems, most works that aim at achieving scalability rely on the use of an overlay network of message brokers, that are used to propagate information among publishers and subscribers [15], [16], [17], [18], [19]. A fundamental problem that emerges when nodes are organized in an overlay network is how to handle faults, in particular what to do when a broker fails. In the next paragraphs we discuss the main techniques that have been proposed in the literature to address this problem. We can cluster previous works to address faults in broker overlays into three main approaches:

Virtual reliable nodes (VRN): This approach consists of having each logical node in the overlay to be implemented by a set of replicated servers such that, even if a server fails, the (logical) overlay node remains correct. Using this approach, the topology of the overlay can be assumed to be static, and the properties of this topology can be used to enforce desirable properties. For instance, if the broker overlay is acyclic and channels between brokers are FIFO, it becomes trivial to ensure that event propagation respects causal order[7]. Examples of systems that use this approach are Saturn[7] and Gryphon[4]. Saturn uses a consensus algorithm, such as Paxos[20], to implement each logical node. Gryphon also uses replication but requires weaker coordination among the replicas of each node; this has the advantage of offering lower latency, but forces each node to maintain state for each sender that exists in the system to tolerate message loss. In any case, replicating every node of the overlay is not feasible when the system becomes very large.

Table I
LoCaMu vs Related Work

Algorithm	Approach	Avoids Replicas	Parallel	Causality	Local
CBCAST [1]	RP	✓	✓	✓	✗
Causal Barriers [6]	RP	✓	✓	✓	✗
D.N.[5]	RP	✓	✗	✗	✓
Saturn [7]	VRN	✗	✓	✓	✗
Gryphon [4]	VRN	✗	✓	✗	✗
Plumtree [21]	OR	✓	✓	✗	✗
Thicket [22]	OR	✓	✓	✗	✗
LoCaMu	RP	✓	✓	✓	✓

On-line overlay reconfiguration (OR): This approach consists in reconfiguring the overlay when a broker fails, where one node can connect directly to any other node in the system. This approach is used in systems such as Plumtree[21] or Thicket[22]. Unfortunately, ensuring reliability and order of messages in face of the overlay reconfiguration may not be trivial, in particular because most reconfiguration algorithms cannot guarantee that reconfiguration is local; since paths before and after the reconfiguration may change substantially, messages may be lost and delivered out-of-order unless global information is used to recover from the fault.

Redundant paths (RP): This approach consists in using an overlay that has redundant paths, such that message delivery can be ensured without reconfiguring the overlay, even if a node fails. The key difference with the previous approach is that nodes have information about all redundant paths they may need, but do not know about every possible overlay reconfiguration. Examples of this strategy are [23], [17], [5], [24]. Although the use of redundant paths increases reliability, it makes it harder to enforce FIFO and causal order, because related messages can follow different paths and be received in unintended orders. To the best of our knowledge, Delta-Neighbourhood[5] is the only algorithm that uses redundant paths and is able to enforce an ordering policy based on localized information. However, it is only able to enforce FIFO order. Furthermore, it restricts the communication pattern, by preventing each publisher from having more than one message in transit at any given time.

Table I summarizes a comparison of the main features of the systems mentioned above. Previous work offers only a partial solution to the problem that we address in this paper. LoCaMu, however, is able to provide fault tolerance, causal order, pipeline multiple messages from the same source, and use only localized information. In this way it combines guarantees of quality of service with the ability to scale.

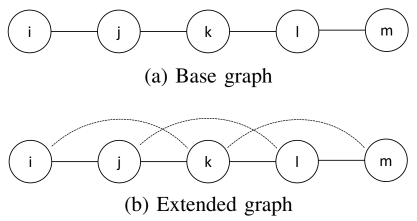


Figure 1. Base graph and the corresponding extended graph ($f = 1$)

III. LoCaMu

In this section we describe LoCaMu. Before we describe the algorithm in detail, we introduce a number of concepts required to understand how the algorithm operates.

A. Base Graph and Extended Graph

It is assumed that nodes are organized in an overlay network that can be modelled by undirected graph $G = (v, e)$, where vertices v represent the system nodes and edges e the communication channels between these vertices. This graph, which is referred to as the base graph, is acyclic and connected. Let v_i be a vertex of the graph. $\mathcal{V}(G, k, v_i)$ represents the set of neighbours of the vertex v_i which are at a distance no longer than k in the graph (that is, if $v_j \in \mathcal{V}(G, k, v_i)$, then the shortest path in the base graph between v_i and v_j has at most k edges). The vertices in $\mathcal{V}(G, 1, v_i)$ are denoted as direct neighbours of v_i (in the base graph). G has no redundant edges and the failure of any node partitions the graph.

In order to tolerate f failures in a given neighbourhood, the base graph G is augmented with additional edges, creating an extended graph G^f. The additional edges connect each vertex v_i with all vertices in $\mathcal{V}(G, f+1, v_i)$, that is, for each $v_j \in \mathcal{V}(G, f+1, v_i)$, an edge is added to the extended graph, connecting v_i and v_j. The additional edges in the extended graph create redundant paths that may be used if a path in the base graph becomes unavailable. Figures 1a and 1b illustrate a base graph and the corresponding extended graph for $f = 1$. In fault-free runs, the base graph is used to propagate the messages in a order that respects causality. If a node (such as f in Figure 2) fails, then the additional edges are used to propagate the message. Even when redundant paths are used, messages are routed along the paths defined by the base acyclic graph, such that cycles are avoided. When these redundant paths are used, messages may be duplicated or re-ordered. As it will be seen, LoCaMu maintains the required metadata to ensure that messages are reliably delivered in order even in runs where faults occur.

B. Addressing Model

Nodes are assumed to be organized into groups that are used to define the destination address of a message; each message being addressed to a single group. The communication uses a publish-subscribe model, where information about group members is propagated in the extended graph through *subscription* messages. This allows each node to maintain a routing table that, for each group, indicates which neighbours participate in the forwarding of the messages. The size of this table depends on the number of groups and not on the number of nodes in the system.

Given a message m, which has a recipient group $g(m)$, a given node i can query its routing table to identify the sub-set of its neighbours in the extended graph that are involved in routing m. This sub-set is called the *targets* of m in i, denoted $\mathcal{T}(m, i)$. This is illustrated in Figure 2, where Figure 2a depicts the base graph and the group members for message m_1 that needs to be forwarded by node i and Figure 2b shows the set of target nodes $\mathcal{T}(m_1, i)$ for that message for $f = 1$.

C. Fault Model

Nodes are assumed to be capable of failing and subsequently recover, and their state is kept in persistent memory. The system is asynchronous and a faulty node may remain unavailable for an arbitrary amount of time. Finally, it is assumed that at any instant of time, in any neighbourhood of size $2f + 1$ in the base graph, there are at most f nodes unavailable. A node that never fails is denoted as correct.

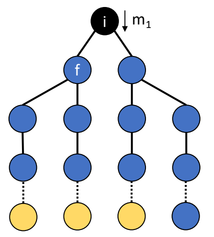
Each node of the system is assumed to have access to an eventually perfect failure-detector [25], [26], that reports if its neighbours are available or unavailable. This detector ensures that if a neighbour becomes unavailable, the node is eventually notified of this fact (and can avoid sending messages in vain while the faulty node remains unavailable). Similarly, when the neighbour recovers, the node is also notified (so that it can use that neighbour again).

Using the output of the failure detector, every node i maintains a set *activeConnections* as follows. Every time the set of faulty nodes in its $(f + 1)$-neighbourhood changes, i recomputes the shortest path to every other node in $\mathcal{V}(G, f+1, v_i)$. Then, the nodes that are both the first non-failed node and hop on any of these paths are added to the *activeConnections* set. When propagating a message in the network, each node only forwards messages to the nodes maintained in this set. This limits the amount of redundant messages that circulate in the network. Figure 2c shows the set of active connections at the black node when its direct neighbours in the base graph are active and Figure 2d shows how this set is updated when one of these direct neighbours (in this case, node f) becomes unavailable.

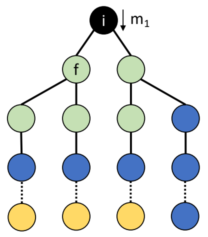
LoCaMu is fault tolerant in the sense that a message m sent to a set of vertices V is delivered to all correct nodes $v_j \in V$ (that is, all recipients who are not or that will not fail), regardless of the failure of other nodes or the eventual recovery of nodes that failed. LoCaMu also ensures that nodes that are temporarily unavailable will end up receiving all messages; of course, this only happens when they recover.

D. Localized Information and Safe Neighbourhood

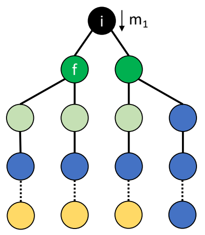
LoCaMu is a localized algorithm, where each node is only required to maintain metadata regarding messages sent or received by other nodes into some *safety neighbourhood*. For this reason, when a message is sent and tagged with



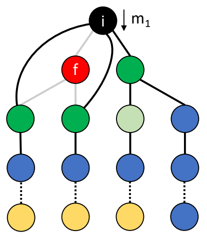
(a) Base graph (blue) and m_1 recipients (yellow)



(b) Targets for m_1, $\mathcal{T}(m_1, i)$ (light green)

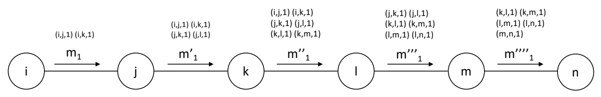


(c) Active connections for m_1 (dark green)

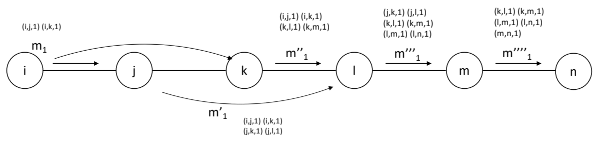


(d) Active connections for m_1 (dark green) after a node disconnects (red node)

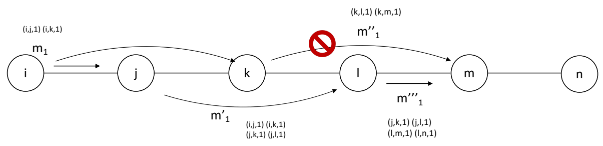
Figure 2. Message targets and active connections.



(a) Creation and propagation of identifiers



(b) Duplicate detection and merging of identifiers



(c) Invalid paths

Figure 3. Message identifiers and safety neighbourhood ($f = 1$)

metadata, it only needs to be tagged with information regarding nodes that belong to the safety neighbourhood of the recipient. In the case of LoCaMu, the safety neighbourhood includes all nodes that are at most $2f + 1$ hops away in the base graph. More precisely, the safety neighbourhood of node i, denoted $\mathcal{V}^S(i)$ is $\mathcal{V}^S(i) = \mathcal{V}(G, 2f+1, i)$. The next subsections provide an intuitive rationale for the size of the safety neighbourhood in LoCaMu and how it relates to the possible paths that can be followed by messages.

E. Message Identifiers

Non-localized algorithms typically assign an unique identifier to a message when the message is generated. This unique identifier is then preserved as the message is propagated in the network. The most common way to generate such unique identifiers is to use a tuple that includes the unique identifier of the source node and a sequence number generated at the source. A fundamental drawback of this solution is that, to perform tasks such as detecting duplicates or enforcing order properties, nodes in the system need to keep track of sequence numbers generated by every other node in the system. These approaches are, therefore, not localized (and not scalable).

LoCaMu is localized as nodes are only required to keep track of identifiers generated by nodes in their safe neighbourhood. Thus, if a message is generated by a remote node, outside the safe neighbourhood, the identifier generated by the source cannot be used; it needs to be replaced by some identifier that is meaningful locally. In consequence, in LoCaMu messages do not have a single identifier that is preserved across the system. Instead, messages are assigned multiple identifiers as they are forwarded in the network, where each of these identifiers is only valid in a given neighbourhood. Still, LoCaMu is able to keep track of how these identifiers are related, such that it is able to eliminate duplicates and to deliver messages in the right order.

The way message identifiers are created, propagated, and replaced, is key to the operation of LoCaMu. To better understand the process of propagating message identifiers, an example is used. Figure 3a shows a network that consists of a line, and illustrates the propagation of a message that is propagated from node i to node n. In this example the number of tolerated faults is $f = 1$ and, therefore, the set of targets for the message are the next two nodes in the line (i.e, $\mathcal{T}(m, i) = \{j, k\}$, $\mathcal{T}(m, j) = \{k, l\}$, etc). Note that in this case, the safety neighbourhood of node m no longer includes node i (given that the distance from i to l is more than $2f + 1$), thus m will not process metadata produced by node i. This results in each message only having identifiers from at most $2f + 1$ different nodes.

Each node keeps a separate sequence number for each target and the tuples *(source, target, sequence number)* are used as messages identifiers. Note that, in the example, when message m is generated, it is assigned two identifiers, namely $(i, j, 1)$ and $(i, k, 1)$, one identifier for each link that can be used to propagate the message. To avoid redundant messages in the network, the message is not sent immediately by both links. In this example, the message is just propagated in the base graph (and edges from the extended graph are just used if faults occur). Then, when the message is propagated by node j it gets assigned two new identifiers, $(j, k, 1)$ and $(j, l, 1)$, and now carries a total of 4 different identifiers. This process is repeated every time a message is forwarded in the network. However, at each step, each sender only forwards identifiers that belong to the safe neighbourhood of the recipient. Thus, when node l forwards message m to node l, it no longer includes the identifiers generated by node i (similarly, when l retransmits the message it no longer includes the identifiers generated by node j).

F. Duplicate Detection and Merging Identifiers

As noted above, LoCaMu does not require a perfect failure detector. Thus, at any given time, different nodes may make a different assessment on the correctness of a given neighbour. Also, nodes may falsely suspect that a neighbour is down and trigger the propagation of the message via the redundant edges defined by the extended graph. This may cause different copies of the message to be propagated by different paths, as illustrated in Figure 3b where two copies of message m_1 are received by node l via different paths, namely, the copy m_1' is received via the path that uses vertices $\{i, j, l\}$ and copy m_1'' via $\{i, k, l\}$. In this case, m_1' does not carry the identifiers added by node k and m_1'' does not carry the identifiers added by node j. Note that, in this example, node j can still detect that m_1' and m_1'' are duplicates of the same original message, given that there is at least one common identifier carried by both copies (more precisely, identifiers $(i, j, 1)$ and $(i, k, 1)$ are common to both copies). Also, when node l propagates the message by sending m_1''', it can tag the copy with all the identifiers known to it (the set of known identifiers results from "merging" the set of identifiers carried by m_1' and m_1'').

This example also shows why node l needs to maintain metadata produced by node i (and thus, i needs to be in the safe neighbourhood of l). If this was not the case, in this run, where a single node is suspected as being failed, l would not be able to detect that m_1' and m_1'' are duplicates of the same original message.

G. Safe Paths

The correctness of LoCaMu is rooted on the *Safe Paths Invariant*, which is defined as follows:

Invariant 1. *Any path of $2f + 1$ nodes is considered safe if there are at most f faulty nodes. Otherwise it is unsafe.*

This means that if two nodes (say i and l in the example of Figure 3) are at $2f + 1$ hops away in the base graph, it should be possible to propagate a message from i to l in a path that contains at least $f + 1$ neighbours. Using the example of Figure 3b, a copy of m_1 arrives to l via a path that uses $\{i, k, l\}$ (skipping j) and another copy arrives to l via a path that uses $\{i, j, l\}$ (skipping k). These paths are said to be *safe*.

H. Causal Past and Causal Order

To enforce causal order every node keeps a log (simply called the node's *past*) of the identifiers of all the messages it has received and processed in the past. The algorithm enforces that messages transmitted via a given edge are delivered in FIFO order. Thus, in practice, because causal order is transitive, only the most recent identifier generated by any node needs to be preserved in the node's past. Also, nodes are only required to keep in the past information from nodes in their safe neighbourhood. When messages are sent they are tagged with the casual past of the sender. By comparing the causal past of a message with its own causal past, a recipient can check if the message can be processed without violating causal order. If some message m cannot be immediately processed when it is received because some other messages in its causal past are missing, the message is stored in a buffer, until it can be processed.

I. Detailed Algorithm

Overview: Messages carry two header fields: a set of identifiers (as described above) and a causal past. When a message is received it is not processed until two conditions are met: i) the message is being processed in FIFO order with regard to other messages from the same sender (message identifiers are used to make this check) and ii) the message is being processed in causal order (the causal past of the message is used to make this check). When a message m becomes ready to be processed one checks if the message is being received for the first time or if another copy of the same message has been previously received via another path (again, message identifiers allow to detect duplicates, as illustrated before). If the message is a copy, its identifiers are merged with the identifiers of the previous copy. After a message is processed, if the message needs to be forwarded (i.e, if the routing table indicates that there are recipients downstream), the message is scheduled for retransmission. Messages are only retransmitted via safe paths. In the next paragraphs a more detailed description of the algorithm is provided. Pseudo-code is provided in Algorithm. 1.

Node Data Structures: Each node maintains three data structures, listed below:

- The first is a matrix of sequence numbers called *Past*, denoted by P_i, with one entry for each pair of nodes in $\mathcal{V}^S(i)$. P_i captures the causal past of the i node. Consider that the position $P_i[j][k] = s$ ($s \neq 0$). This means that the state of i depends causally on a message sent from node j to node k with the sequence number s.
- The second is a matrix of *received* identifiers, R_I. Each entry $R_i[j][k]$ is an ordered set that contains the identifiers of all messages sent by node j to node k that have already been processed by node i. The objective of R_i is to detect duplicate messages, since in faulty runs, the same message can be received via different paths. While every entry is a set, each set is regularly garbage-collected, which is explained in the full thesis.
- The third is the *sent* set that contains all buffered messages.

Algorithm 1 LoCaMu

```
▷ Retransmits a message to the relevant nodes
function (RE)TRANSMIT($m$) at node $i$
    for $k \in \mathcal{T}(m, i) \cap activeConnections$ do
        if PATHISSAFE($m$, $k$) then
            SEND($m$, $k$)
▷ Forwards a message to the interested nodes
function FORWARD($m$) at node $i$
    $D_m \leftarrow P_i$
    for $k \in \mathcal{T}(m, i)$ do
        $I_m[i][k] \leftarrow D_m[i][k] + 1$
        $D_m[i][k] \leftarrow D_m[i][k] + 1$
    (RE)TRANSMIT($m$)
    $sent \leftarrow sent \cup \{m\}$ ▷ For future retransmissions
▷ Updates the Past and Received matrices of a node using message $m$
function UPDATENODESTATE($m$) at node $i$
    for $k, l \in I_m \cap R_i$ do
        $R_i[k][l] \leftarrow R_i[k][l] \cup I_m[k][l]$
    for $k, l \in \mathcal{V}^S(i)$ do
        $P_i[k][l] \leftarrow \text{MAX}(P_i[k][l], D_m[k][l], I_m[k][l])$
▷ Updates any missing identifiers in a sent message
function MERGEIDENTIFIERS($m$) at node $i$
    if $\exists m' \in sent : \exists k, l : I_m[k][l] = I_{m'}[k][l]$ then
        for $k, l \in \mathcal{V}^S(i)$ do
            $I_{m'}[k][l] \leftarrow \text{MAX}(I'_m[k][l], I_m[k][l])$
▷ Checks if any buffered messages can be processed
function UNBUFFER at node $i$
    for $m \in buffer$ do
        if $\forall k : D_m[k][j] \leq P_i[k][j]$ then
            $buffer \leftarrow buffer \setminus \{m\}$
            RECEIVE($m$)
▷ Receives a message from FORWARD($m$)
function RECEIVE($m$) at node $i$ sent by node $j$
    if $\exists k : D_m[k][i] > P_i[k][i] \vee \exists l : l < I_m[j][i] \wedge l \notin R_i[j][i])$ then
        ▷ Message out of order
        $buffer \leftarrow buffer \cup \{m\}$
    else
        UPDATENODESTATE($m$)
        if $\exists k, l : I_m[k][l] \in R_i[k][l]$ then
            ▷ Duplicate copy
            MERGEIDENTIFIERS($m$)
        else
            ▷ First copy
            if $i$ belongs to $g(m)$ then DELIVER($m$)
            FORWARD($m$)
        UNBUFFER()
▷ A node generates a message
function PUBLISH($m$)
    FORWARD($m$)
```

Message Control Fields: Each message contains two control fields, listed below:

- The first, called *identifiers* (denoted as I_m), is an array of up to $2f+1$ vectors, containing the known sequence numbers that were assigned to m by each node in any given path.
- The second, called *dependencies* (denoted by D_m) carries the causal past of m known by the forwarder; D_m is of size $|\mathcal{V}^S(i)|^2$.

Message Reception: When a node i receives m, which was sent by j, i performs the steps presented in the RECEIVE function in Algorithm 1. In short, the node verifies if all causal dependencies have been satisfied (that is, the messages in the past have already been delivered). If that is the case, then i updates its P_i and R_i and then, if the message is not a duplicate, delivers m if i is interested in it and *forwards* m. If it is a duplicate and the original copy was forwarded then the original copy's identifiers are updated with the duplicate's identifiers. Note that, after the identifiers have been merged, some paths for the original message may become safe.

Message Forwarding: When a node i wants to forward a message m for the first time, the following manipulations are made to its own state and to the metadata of message m: First, i puts $D_m = P_i$. Then i queries its routing table to get the targets of m, $\mathcal{T}(m,i)$. Then, the values of $P_i[i][j]$ $\forall j \in \mathcal{T}(m,i)$ are incremented by one and a new vector with the changed entries is added to I_m. m is then sent to the targets present on *activeConnections* (i.e., to $\mathcal{T}(m,i) \cap$ *activeConnections*), provided the resulting path is safe, as described in Section III-G

J. Extra details

The following details are not mentioned in this extended abstract but are included in the full thesis: Message Retransmission, Message Stability, R_i matrix garbage collection, message metadata optimization, correctness proofs.

IV. Evaluation

LoCaMu is evaluated in order to answer the following questions:

- What is the performance of LoCaMu when compared to a localized algorithms (even if it only offers FIFO order)?
- What is the performance of LoCaMu when compared to other algorithms that enforce causal order?
- How does LoCaMu compare with previous works in term of signaling and memory overhead?

A. Experimental Settings

LoCaMu is compared with CBCAST [1] and Causal Barriers [6] (that ensure causal order but are not localized) and with Delta-Neighbourhood[5] (that is localized but only only ensures FIFO). Nodes are organized in an overlay network with the topology of a binary tree. It is assumed that there is a multicast group for each subtree of the overlay. This means that there is a large group that contains all nodes, then two smaller groups with the nodes on the left subtree and the nodes of the right subtree, respectively, and then 4 smaller groups the result from further dividing the tree, and so forth. This is a simple setup that ensures that the experiments combine groups of all sizes (up to groups of just two members), while showcasing the advantages of locality. Simulations were used to perform the evaluation because this allowed us to experiment large overlays that would be otherwise impossible to test. For this the Peersim [27] simulator was used, with extensions to simulate edges with realistic latency and bandwidth constraints. In the experiments, each node has a limited bandwidth that is shared among all its edges. All links have an average latency of 1ms (note that the latency only affects the throughput of [5], not the throughput of LoCaMu, [1], or [6]).

Regarding the performance metrics, *maximum individual throughput* was the most used metric, which considers a single sender, and the *maximum aggregated throughput*,

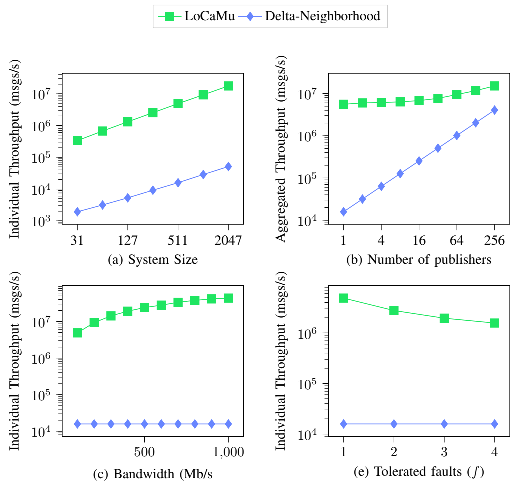


Figure 4. LoCaMu vs Delta-Neighbourhood under different settings

which considers several different senders. By definition, aggregated throughput is the total number of messages delivered to all nodes in the overlay. All the algorithms used in the evaluation are able to capture causality with roughly the same degree of accuracy. Thus, the differences in performance, if any, will be mainly caused by the amount of metadata that messages need to carry. This metadata consumes bandwidth and affects the latency of message propagation. Since the overlay is a tree, the maximum aggregated throughput is limited by the bandwidth of the root node. Therefore, it is expected that algorithms that use less metadata can achieve higher aggregated throughput than algorithms that use more metadata. However, algorithms such as Delta-Neighbourhood impose additional restrictions on message pipelining, that prevent nodes from using effectively the available network bandwidth.

Finally, it is important to emphasize that all algorithms use a number of optimizations in their implementation, which are detailed in the thesis. For fairness, in these experiments, all 3 protocols use their optimized versions.

The following experiments use a base configuration consisting of a) the system size was set to 511 nodes; b) the bandwidth of the nodes was set to 100 Mb/s; c) the payload size was set to 32 bytes (this value was chosen based on Facebook's production TAO system [28], where 32 bytes represents the 90 percentile of data size that is stored); d) the number of tolerated faults in each neighbourhood was set to $f = 1$; e) the node degree was set to 3 (a binary tree). In each experiment one of these variables is changed, keeping the remaining parameters unchanged. Information regarding the number of publishers and groups is declared for each experiment.

B. *LoCaMu vs Delta-Neighbourhood*

LoCaMu and Delta-Neighbourhood are similar in the sense that both use localized information, therefore it is important to compare them. Since the latter offers less guarantees (only supports FIFO), it should perform better than LoCaMu (that needs to keep track of causal dependencies among multiple senders). As will be shown, Delta-Neighbourhood is, in fact, able to offer better maximum aggregated throughput if there are enough publishers and enough nodes in the system, but performs poorly when the maximum individual throughput is considered. This happens because in Delta-Neighbourhood a node is not allowed to send a new message before the previous one is received.

In these experiments, messages are sent to a single group that includes all nodes and by default there is only one publisher that publishes messages as fast as possible, saturating the bandwidth of the root node. Figure 4a shows the maximum individual throughput as the system size is changed. As this size increases, the diameter of the network grows and it takes longer for a message to reach its destinations. Since, Delta-Neighbourhood does not support pipelining, the throughput of this protocol scales poorly with the system size; LoCaMu does not have such drawback. In Figure 4b, the number of senders is changed in order to measure the maximum aggregated throughput. Since LoCaMu exploits pipelining, it is able to approximate the maximum network capacity even with a single sender. Delta-Neighbourhood has less individual throughput but, as it uses much less metadata, it supports a much higher aggregated throughput, growing linearly with the number of publishers and eventually overcoming LoCaMu's. In Figure 4c, the bandwidth available to the nodes is changed. Again, given that the maximum individual throughput of Delta-Neighbourhood is constrained by the end-to-end latency, it cannot benefit from the extra available bandwidth, while in LoCaMu a single sender can fully exploit the available bandwidth. Finally in Figure 4d, the f value is changed; because the size of the metadata in LoCaMu is a function of the size of the safe neighbourhood, and this size grows exponentially with f, the performance of LoCaMu drops notably for large values of f.

V. LoCaMu vs Causal Multicast algorithms

Causal order is only relevant when there are multiple publishers, hence it makes no sense to measure the maximum individual throughput of an isolated node. Therefore, this section concentrates on assessing the maximum aggregated throughput achieved by the different algorithms. For these experiments the same base configuration is used as before. However, to make sure that all nodes send messages and causal dependencies are established among these messages, the throughput of each publisher is limited to 100 msg/s. Note that the size of the metadata maintained by LoCaMu and Causal Barriers is similar, regardless of the number of groups in the system. CBCAST uses a vector clock for each group, thus the metadata changes with the number of groups. The experiments were ran with 1 group and with 255 groups. Two different bandwidths were also used, in order to see the impact of the metadata increase on the throughput.

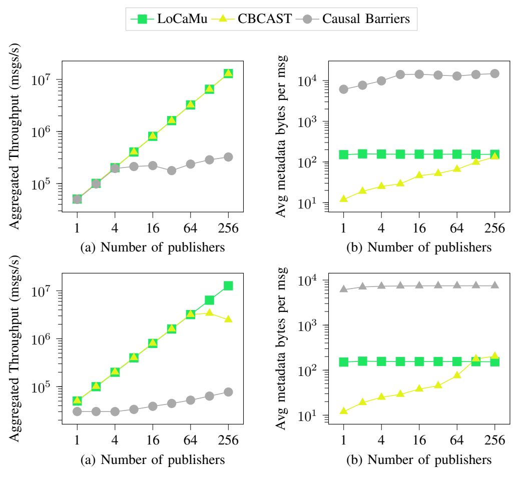


Figure 5. Throughput comparison (left) in a system with 511 nodes with 1 group between the causal algorithms and the corresponding average metadata size per message (right). Bandwidth is 100 Mbps (top) vs 10 Mbps (bottom)

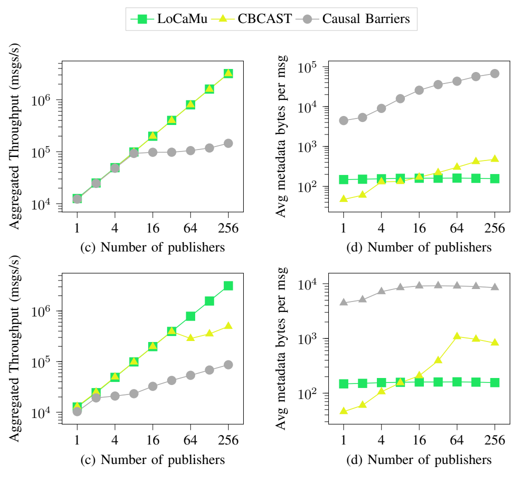


Figure 6. Throughput comparison (left) in a system with 511 nodes with 255 groups between the causal algorithms and the corresponding average metadata size per message (right). Bandwidth is 100 Mbps (top) vs 10 Mbps (bottom)

A. Single Group

Figure 5a and Figure 5c compare the maximum aggregated throughput achieved by the different algorithms when all nodes send messages to a single group that includes all members. As before, as more publishers are added to the system, the aggregate throughput continuously increases up to a point where the maximum flow of the network is reached. Figure 5b and Figure 5d show the corresponding average metadata present in each message when the number of concurrent publishers is increased. By analysing these two last figures, Causal Barriers is shown to use a very high amount of metadata and while CBCAST initially uses less metadata than the other algorithms due to its optimizations compressing the clocks, as the number of parallel publishers increases, the effectiveness of the optimizations decreases, until it eventually starts requiring more metadata than LoCaMu. This in turn results in its aggregate throughput decreasing as seen in Figure 5c, where, due to the network having a smaller bandwidth, the network is saturated.

B. Various Groups

Figure 6a and Figure 6c compare the maximum aggregated throughput achieved by the different algorithms when all nodes send messages to the existing 255 groups of different size (as explained in IV-A). Figure 6b and Figure 6d represent the corresponding average message metadata. Causal Barrier' and LoCaMu's average metadata is similar when there is one group or several groups, as their metadata is independent of the number of groups, so these algorithms' performance is similar to the previous section. CBCAST, instead, scales much worse, because messages are required to carry several vector clocks (in the worst case, as many as the number of groups that exist in the system).

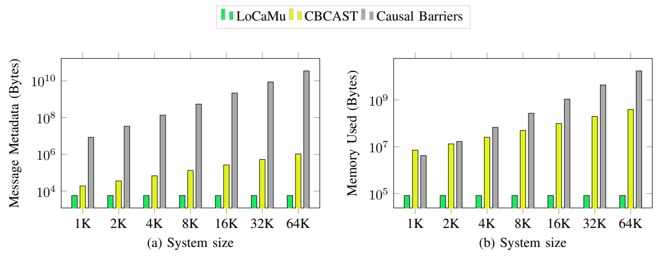


Figure 7. Metadata worst case and memory required (causal algorithms). Default settings: $f = 1$, node degree = 2, nodes = $65,536$

VI. Metadata size comparison

From the results of the previous experiments, it is clear that the size of the metadata exchanged in the header field of messages has a significant impact on the performance of the different algorithms. Detailed data is now provided for the amount of metadata required by each algorithm.

A key aspect that the reader should retain is that Causal Barriers and CBCAST are inherently non scalable: the size of the metadata required by the algorithms quickly becomes unfeasible to manage in practice as the size of the system grows. Recall that in Causal Barriers the size of metadata may be roughly quadratic with the system size, for CBCAST is linear with the system size and linear with the number of groups, and with LoCaMu is exponential with the size of the safe neighbourhood. The size of the safe neighbourhood is a function of f and of the degree of the nodes in the overlay, but does not depend on the entire system size. For instance, for a system with $1,024$ nodes, assuming that every node has sent a message to every other node, Causal Barriers would be required to send a message with over 2^{20} entries. Assuming 8 bytes per entry (an entry is a tuple that includes an identifier and a sequence number), Causal Barriers uses 8 MB. In the same setting, CBCAST would require a vector clock with size 4 KB for a single broadcast group. In turn, LoCaMu requires 540 bytes assuming a node degree 3 and $f = 1$, *no matter the system size*. Figure 7 illustrates these facts with more detail. Here, consider the existence of $1,500$ groups and network size of $65,536$ nodes. A Zipf-like distribution is used for assigning the size of the group membership such that $\mathit{Members}(g) = [Ng^{-1.25}+0.5]$, where g is the unique number of the group (from 1 to $1,500$ in this particular case) and N is the amount of nodes in the system. This results in the sum of the amount of members of every group being $237,895$. This distribution was used for the membership of the groups, because it typically used in others works that consider large-scale publish-subscribe settings (such as [11]).

Figure 7a and Figure 7b show the size of the header fields and the amount of local memory used by each algorithm, as the size of the system increases. These plots consider a node degree of 3 and $f = 1$, which show that LoCaMu scales much better than the opposing algorithms.

VII. Conclusions

This paper has presented LoCaMu, an algorithm that guarantees message delivery with causal order in a publish-subscribe system built on top of a broker overlay. LoCaMu is the first algorithm to offer causal order and fault tolerance while using localized information, i.e. each node maintains state regarding only the nodes on its neighbourhood and not every other node on the system. Thus, LoCaMu can be used in large scale systems, while previous work requires so much metadata that any practical implementation becomes infeasible. For systems with hundreds of nodes, where the previous work can still be applied, LoCaMu shows clear advantages, given that it makes a much better use of the available bandwidth.

Acknowledgments

This work was partially supported by FCT – Fundação para a Ciência e a Tecnologia, through projects UID/CEC/50021/2019 and COSMOS (funded by the OE with ref. PTDC/EEI-COM/29271/2017 and by Programa Operacional Regional de Lisboa and FEDER with ref. Lisboa-01-0145-FEDER-029271).

References

[1] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *TOCS*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

[2] J. Lin and S. Paul, "RMTP: a reliable multicast transport protocol," in *INFOCOM*, San Francisco (CA), USA, Mar. 1996, pp. 1414–1424 vol.3.

[3] I. Gupta, R. van Renesse, and K. Birman, "Scalable fault-tolerant aggregation in large process groups," in *DSN*, Göteborg, Sweden, Jul. 2001, pp. 433–442.

[4] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach, "Exactly-once delivery in a content-based publish-subscribe system," in *DSN*, Bethesda (MD), USA, Jun. 2002, pp. 7–16.

[5] R. Kazemzadeh and H. Jacobsen, "Partition-tolerant distributed publish/subscribe systems," in *SRDS*, Madrid, Spain, Oct. 2011, pp. 101–110.

[6] R. Prakash, M. Raynal, and M. Singhal, "An efficient causal ordering algorithm for mobile computing environments," in *ICDCS*, Hong Kong, May 1996, pp. 744–751.

[7] M. Bravo, L. Rodrigues, and P. van Roy, "Saturn: a distributed metadata service for causal consistency," in *EuroSys*, Belgrade, Serbia, Apr. 2017, pp. 111–126.

[8] R. V. Renesse and et. al, "Horus: A flexible group communications system," Cornell University, Ithaca (NY), USA, Tech. Rep., Mar. 1995.

[9] JGroups, http://www.jgroups.org/index.html, accessed: 2019-06-29.

[10] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *SOCC*, New York (NY), USA, Nov. 2014, pp. 1–13.

[11] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *NGC*, London, UK, Jul. 2001, pp. 30–43.

[12] F. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *CSUR*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[13] J. Leitão, J. Pereira, and L. Rodrigues, "Hyparview: A membership protocol for reliable gossip-based broadcast," in *DSN*, Edinburgh, UK, Jun. 2007, pp. 419–429.

[14] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *CSUR*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[15] Y. Huang and H. Garcia-Molina, "Publish/subscribe in a mobile environment," *Wireless Networks*, vol. 10, no. 6, pp. 643–652, Nov. 2004.

[16] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," in *ICDCS*, Austin (TX), USA, Jun. 1999, pp. 262–272.

[17] R. Chand and P. Felber, "Xnet: a reliable content-based publish/subscribe system," in *SRDS*, Florianópolis, Brazil, Oct. 2004, pp. 264–273.

[18] A. Carzaniga, D. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *TOCS*, vol. 19, no. 3, pp. 332–383, Feb. 2003.

[19] J. P. d. Araujo, L. Arantes, E. P. Duarte, L. A. Rodrigues, and P. Sens, "A publish/subscribe system using causal broadcast over dynamically built spanning trees," in *SBAC-PAD*, Campinas, Brazil, oct 2017, pp. 161–168.

[20] L. Lamport, "The part-time parliament," *TOCS*, vol. 16, no. 2, pp. 133–169, May 1998.

[21] J. Leitão, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," in *SRDS*, Beijing, China, Oct. 2007, pp. 301–310.

[22] M. Ferreira, J. Leitão, and L. Rodrigues, “Thicket: A protocol for building and maintaining multiple trees in a p2p overlay,” in *SRDS*, New Delhi, Punjab, India, Oct. 2010, pp. 293–302.

[23] A. Snoeren, K. Conley, and D. Gifford, “Mesh based content routing using xml,” in *SOSP*, Banff, Alberta, Canada, Jan. 2001, pp. 160–173.

[24] B. Nédelec, P. Molli, and A. Mostéfaoui, “Causal broadcast: How to forgetl,” in *OPODIS*, Hong-Kong, China, Dec. 2018.

[25] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer, 2006.

[26] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *JACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[27] A. Montresor and M. Jelasity, “PeerSim: A scalable P2P simulator,” in *P2P*, Seattle (WA), USA, Sep. 2009, pp. 99–100.

[28] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, “Tao: Facebook’s distributed data store for the social graph,” in *ATC*, San Jose (CA), USA, Jun. 2013, pp. 49–60.