Tree Topologies for Causal Message Delivery

Sebastian Blessing
Department of Computing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

Sylvan Clebsch
Microsoft Research
United Kingdom
sylvan.clebsch@microsoft.com

Sophia Drossopoulou
Department of Computing
Imperial College London
United Kingdom
s.drossopoulou@imperial.ac.uk

Abstract

Causal message delivery, i.e. the requirement that messages are delivered in an order respecting their causal (logical) dependencies, is often mandated in the distributed setting. So far, causal message delivery has been implemented by augmenting messages with meta data information that allows the receiver (or the platform) to re-order, and if necessary hold back, messages upon receipt before processing.

We propose that causal message delivery can be achieved by construction, simply by organizing the nodes of the distributed application into a tree topology, and without the need for any meta data in the messages.

We present our ideas informally through an example application and then develop a formal model and prove that causal message delivery is preserved in tree-based networks.

CCS Concepts • **Computer systems organization** → **Distributed architectures**; • **Theory of computation** → *Operational semantics*;

Keywords network topologies; causal messaging

ACM Reference Format:
Sebastian Blessing, Sylvan Clebsch, and Sophia Drossopoulou. 2017. Tree Topologies for Causal Message Delivery. In *Proceedings of 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3141834.3141835

1 Introduction

The ordering of events in distributed applications is an important artifact. The ability to observe it requires a relationship between messages that induces the idea of "happens before" [16]. This ordering can be either *total* [16] or partial. Partial orderings can be categorized in *causal* dependencies [7] or *explicitly* stated relationships [5].

In the context of causal messaging, we say that each message is an *effect* and every message that was received or sent prior to that message is a *cause* of that effect. Causal ordering is preserved if every cause is enqueued before its effect and is therefore less restrictive than a total order, cheaper to achieve but strong enough for most applications [22]. A substantial amount of the research in the field of distributed systems focused on *detecting* [16, 20] causality violations or *enforcing* [6, 22] ordered message delivery. Merely being able to detect causality violations puts more complexity into the application and eventually requires recovery protocols to continue processing. Hence, there is a case for preferring enforcement over detection.

Motivation In current practice, programmers are required to design an application to tolerate uncoordinated and inconsistent orderings of messages or need to implement a distributed algorithm to explicitly prevent such re-orderings from ever occurring. These techniques, however, either come with linear (or even quadratic) space complexity over the set of involved processes/nodes or employ some central coordinator that, depending on the application, may be highly contended and performance critical. That is, there is a trade off between tracking causal dependencies and achieving high availability and/or scalability.

Since *enforcing* causal message delivery is potentially expensive, and increasingly so in large distributed systems, reducing the overhead that comes with it can have a substantial impact on scalability, without being forced to give up as much on availability as traditional approaches. Central to our work is a discussion on to which extend messages need to be augmented with meta data (if at all) and whether running a distributed algorithm to ensure causality is a *necessity*.

Background Observing the order of events in distributed systems is a complex and powerful concept that forms the basis for a wide range of applications, such as distributed snapshots [4, 17], version vectors and conflict detection in distributed databases [12, 27], causal consistency [1, 2] as well as causal broadcast [6]. Causality is a generalization of FIFO ordering [29]. The subtle difference between FIFO and causal ordering is that the former only applies to messages having the same sender, whereas causality applies to

AGERE'17, October 23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5516-2/17/10…$15.00
https://doi.org/10.1145/3141834.3141835

messages from any process or node [16]. Besides having provided a definition of the "happens-before" [16] relationship and being able to observe it, one of the most important findings of Lamport's work on ordering events in distributed systems [16] is that reasoning about "happens before" does not require synchronous coordination between processes. Moreover, one of the most interesting properties is that once causal ordering is established, many other concepts and their implementation become much simpler [8].

Causality means that processes must decide whether a message can be received or if it needs to be held back as there might be dependent messages in flight but out of order. Detecting that receiving a message is safe is a complex task. This decision and collecting the required meta data can be space and time consuming [5, 9, 14, 23]. Moreover, inspecting physical timestamps is not enough, since time is not necessarily moving forward uniformly on all nodes in a distributed system.

A classic approach to this problem is to implement a message service based on Lamport clocks, using logical time to order messages on every node [16, 20]. Lamport clocks record causal dependencies by tagging every message sent with a *single* increasing integer value. As a result, Lamport clocks have a linear space requirement (over all involved processes) and do not allow for completely independent messages. The mechanism described in [16] induces a total order over all messages by making sure that every site of the system has received at least one message from every other site before progressing. That is, $O(n)$ acknowledgment messages. This is immensely expensive and implies more dependencies than truly exist. Moreover, a total ordering is too strict for certain applications [12], where detecting that some message cannot possibly be preceded by another message is key for application correctness. This issue can be addressed by vector clocks [13, 16], where every process maintains an integer value for every other process of the system. However, the price to pay is quadratic space complexity and the amount of nodes involved in a system is static and needs to be known a priori. Dynamic topologies are addressed by interval tree clocks in [3].

Charron-Bost provided a surprising result and theoretical argument [9], saying that the amount of space required for the necessary meta data to detect causal dependencies in distributed systems is at least linear. That is, there is no better time stamping algorithm based on time vectors smaller than $O(n)$ that "truly characterizes causality" [9, 14, 23]. The fundamental concepts have been developed in the late 1970s, and from then on research has mainly focused on developing protocols to tackle the amount of (global) space required as well as decoupling the effects causal ordering has on unrelated messages in terms of messaging performance. There is no free lunch and there are only a few strategies to reduce these overheads.

The most intuitive idea would be to restrict the number of participants. This is exactly the strategy distributed databases use in the context of version vectors [12] by intelligently picking the items for which causal dependencies are tracked, e.g. data item replicas instead of all servers. Since the number of replicas is usually substantially smaller than the number of involved machines in a distributed system, this is a powerful optimization in practice [18, 19].

Inducing full causality assumes that all messages matter equally. Generally, for most types of applications, this is not the case. Hence, another optimization would be to explicitly specify relationships between items [5]. In the best case, this can reduce the local space required to $O(1)$. In the context of applications where a total order of events is not paramount, such as Twitter, this is an effective and powerful trade off between performance and the non deterministic order of messages. The advantage of these approaches is that they have no negative impact on availability and only give up on causal dependencies that are not significant to the specific application [25, 26].

Reducing the number of concurrent participants in a vector clock based system [13] is possible and saves space, but sacrifices some amount of availability. A system with only one concurrent participant implies a total order of events [15]. This is even the case in data race free multi-core environments [10].

Some *eventually consistent* applications may drop "happens before" ordering entirely. Systems belonging to this application domain are not subject to the ideas presented in this work.

Contributions Our technique *enforces* (rather than *detecting*) causal message delivery by arranging the nodes in a tree topology, resulting in a mechanism that eliminates the need to collect meta data for detecting causal relationships entirely, and that can enforce causal order rather than merely detecting violations after the fact. We either physically connect individual nodes to a tree network or put a networking service in place to arrange sockets accordingly. During runtime of the system, no main memory or CPU time is occupied with the task of deciding whether or not a message can be delivered. Instead, they can be processed *immediately* upon receipt. However, structured networks, such as trees, do limit the choices that can be made between network configurations. Moreover, for the mechanism presented in this paper to work, we do assume that sending messages between actors on a single node respects causal order.

We expect that substantial code reductions are possible when developing distributed applications, as software systems can implicitly rely on causally ordered events. In other words, in tree networks, causality is an invariant and an inherent property of a systems behavior.

The key contribution of this paper are:

- A Zero-space mechanism to enforce causal ordering by topological arrangement.
- A formal model to describe message passing in tree networks, described through an operational semantics.
- A sketch of the proof that causality is preserved.

Outline We present an informal view of Tree-based causality in section 2, formalize it in section 3, and provide a sketch for a correctness proof in section 4. We conclude and discuss further work in section 5.

2 Tree-Based Causality

In this section we describe what we mean by causality, by causal message delivery, and the TCP/IP delivery, and explain informally how a Tree-based topology ensures causal message delivery.

Preliminaries We say that a message **causes** another message, if one of the following three rules applies:

1. If an actor (or process) receives a message and later on sends another message, then the first message is a cause of the second message.
2. If an actor (or process) sends a message and later on sends another message, then the first message is a cause of the second message.
3. Causality is transitive.

Causal message delivery mandates that if two messages are in a causal relationship and have the same recipient, then the causing message will be delivered before the other one. In Lamport's terminology, [16], if two messages are in a causal relationship, the message that *logically precedes* some other messages is to be delivered *before* the message it is related to. We expect that the actors are placed on nodes in a *tree*. The tree topology does not restrict which actors may communicate with each other – in fact, actor references can form an arbitrary graph. But it means that messages to actors in different nodes must travel via the node which is the most common ancestor of the nodes holding the two actors. In this work we are not concerned with how the tree is built, nor with how the actors are placed on the nodes – an algorithm to arrange nodes in a tree topology with a networking service is presented in [8].

We assume that communication across nodes is consistent with **TCP/IP**: Packages (or messages) sent across one network connection arrive in the order they were sent – *i.e.* there is no overtaking of messages between adjacent nodes.

Example Consider three actors (or processes): A *Customer*, a *Shop* and a *Bank*: The *Customer* intends to buy an item from the *Shop* by electronic cash. She makes sure that the balance on the bank account is high enough by sending a "credit" message to the *Bank*, and then informs the *Shop*, by sending it a "buy" message. Upon receipt of the "buy" message, the *Shop* sends a "debit" message to the *Bank*. Obviously, it is crucial that the *Bank* receives the "credit" message before it receives the "debit" message.

Causal message delivery guarantees exactly that! Namely, "credit" causes "buy" (rule 2), "buy" causes "debit" (rule 1), and thus "credit" causes "debit" (rule 3). Since "credit" and "debit" have the same recipient, causality guarantees that "credit" will arrive first! On the other hand, even though "credit" causes "buy" there is no guarantee as to whether "credit" will arrive at the *Bank* before "buy" arrives at the *Shop*.

In this paper we argue that in a distributed system, a tree topology of the nodes on which the actors are running implicitly guarantees causal message delivery.

To continue with our example, consider a system consisting of four nodes, $\iota 1$, $\iota 2$, $\iota 3$, and $\iota 4$, arranged in a tree as shown in figure 1, and where *Customer*, *Shop* and *Bank*, are scheduled on nodes $\iota 1$, $\iota 3$, and $\iota 4$ respectively.

In figure 1 we show four snapshots from one possible scenario. In the first snapshot, *Customer* sends "credit" to the *Bank*: Since $\iota 1$ has no direct connection to $\iota 3$, the message is first sent to $\iota 2$. In the second snapshot, $\iota 2$ forwards "credit" to $\iota 3$, and *Customer* sends "buy" through $\iota 2$. Note that, a scenario where ι_2 first sends "buy" and then "credit" is impossible, because ι_2 will forward the messages in the order they arrive, and because TCP/IP guarantees that the message do not overtake each other on the wire. In the third snapshot, "buy" has been forwarded from $\iota 2$ to $\iota 3$, and from $\iota 3$ to $\iota 4$. In the last snapshot, *Shop* has received "buy", and as a result it sends "debit" to the *Bank*. In this scenario, causal message delivery is respected!

But there are more possible scenarios on the same topology: Other messages may be sent interleaved with the messages from our example, and "buy" may be sent only *after* "credit" has been received. Nevertheless, even though the timing of a message may vary, its route is always the same. In this topology, the route of "credit" is a prefix of the rote of "buy", and because "credit" starts its journey before "buy", we have the guarantee, by TCP-IP, that "buy" cannot overtake "credit". Therefore, "credit" will arrive at the *Bank* before "buy" arrives at the *Shop*. And because "buy" caused "debit", we also have that "credit" will arrive at the *Bank* before "debit".

But will causal message delivery be guaranteed on other topologies? In figure 2 we see two further topologies. We can see that causal message delivery for our example is guaranteed for these topologies too, but for slightly different reasons:

In the left topology in figure 2, "buy" may arrive at the *Shop* before "credit" arrives at the *Bank*, even though "credit" causes "buy" – remember that causal message delivery is agnostic about delivery on different nodes. Nevertheless, "debit"'s route has "credit"'s route as a suffix, and as "debit"

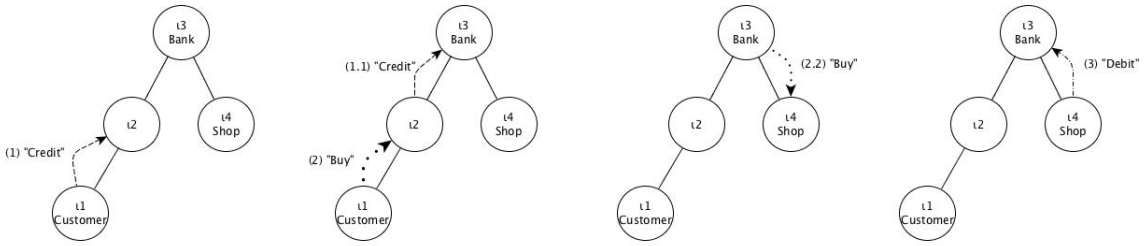

Figure 1. One possible scenario on one tree topology

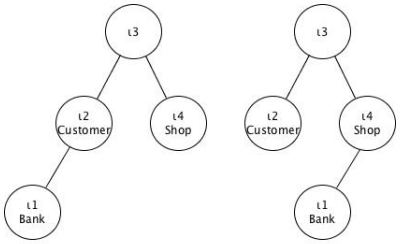

Figure 2. Two further topologies

will start after "credit", it cannot overtake "credit". Thus, "debit" will be delivered after "credit" has been delivered.

In the right topology in figure 2, we also have that "buy" may arrive at the *Shop* before "credit" arrives at the *Bank*. However, "credit" is guaranteed to arrive at $\iota 4$ before "buy". Therefore, "debit" will be placed on $\iota 1$'s queue after "credit", and will arrive at *Bank* after "credit".

So far, we considered three different topologies with four nodes, and in these topologies, for three actors and three messages, causal message delivery is guaranteed. We used the argument that the route taken by the "causing" message is included in some way in the route of the combined messages it "caused".

But does this argument apply over any topologies with any number of nodes, any number of actors, and any number of messages involved in a causal relationship, and any number of messages unrelated to the causal chain?

The answer is yes. However, even though at an intuitive level the guarantee that a tree topology guarantees causal message delivery seemed convincing, it became less so when considering more cases. For this reason, we developed a formal model and a proof, which we will present in the next section.

3 Formal Model

In this section we develop a formal model, expressed as an operational semantics for sending and receiving messages in a Tree-based network. The runtime configuration $\mathcal{T}, \Sigma$ does not reflect the causal relationship of messages. This can only be deduced from the "history" ghost state Δ. The core of our argument is the "arrives before" relation between messages, $<$, which is defined so that if $m < m'$ can be observed in the current snapshot $\mathcal{T}, \Sigma$, then m will arrive ι before m' arrives at ι, where ι is m's destination. Interestingly, the relation is not transitive, and even allows in some cases that $m < m'$ as well as $m' < m$. We prove that causality, which can be observed in Δ, implies the arrives before relation. We were surprised that we were able to prove our result without needing to reflect time – or even logical time – in the model.

3.1 Runtime Entities and Operational Semantics

Figure 3 presents the runtime entities and shows the operational semantics describing the behavior of a Tree-based distributed system.

Runtime entities Both node and message identifiers, referred to as ι and μ, are (uniquely) picked out of the set of natural numbers. A message m consists of an ID and its origin node. At any point in time during execution of the system a message is contained in exactly one message queue until it was received. That is, message delivery is immediate. We model a message queue q as an unbounded sequence of messages. The tree topology $\mathcal{T}$ is modeled as a mapping from node identifier to node identifier. Every node within the tree holds a pointer to its parent. The root node points to itself. Every node n holds separate in- and out-queues for each of its children and its parent node. In-queues contain messages being received from the node related to that neighbor, and out-queues contain messages that need to be delivered to the next hop towards the destination of a message, respectively. We separated sender from receiver queues in order to model unbounded point-to-point delay, resulting in ordered but arbitrarily delayed distributed network semantics. The state of every node, i.e. the contents of its message queues, can be observed in Σ. Furthermore, events are defined as either sending or receiving a message m at some node ι, and are recorded as shadow state in the history Δ. Paths are sequences of node identifiers.

Note that we are not modeling actors and their state as part of the system, since sending messages between actors on a single distributed participant is causal [10] and therefore we are only interested in inter-node communication. Auxiliary helper functions to mutate runtime entities such as queues and the history to allow for a succinct presentation of the

operational semantics are briefly discussed in section A and shown in figure 9 and 11.

Notation In this paper, we make use of some additional notation for convenience.

- We use set operations on the domains of mappings:
  - $x \in \text{map} \iff x \in \text{dom}(\text{map})$
- We use set operations on sequences:
  - $e \in \text{seq} \iff seq = \_ \cdot e \cdot \_$
- We use set operations to express the existence of queues and messages:
  - $q \in \Sigma \iff \exists \iota, \iota'.\text{In}(\Sigma, \iota, \iota\prime) = q \vee \text{Out}(\Sigma, \iota, \iota') = q$
  - $m \in \Sigma \iff \exists q \in \Sigma.m \in \Sigma$
  - $m \in n \iff \exists \iota.[m \in n(\iota)\downarrow_1 \vee m \in n(\iota)\downarrow_2]$
  - $m \in q \iff \exists q' \in \Sigma.q' = \_ \cdot m \cdot \_ \wedge q = q'$
- We use index operations to examine sequences:
  - Seq[k] is the k^{th} item in Seq

Operational semantics In a distributed system, execution progresses by sending and receiving messages. For any message to reach a node from its origin within a tree network, a routing and forwarding scheme is required. Our operational semantics consist of four rules, discussed in more detail in the following.

M-Send Sending a new message. The newly created message at node ι_1 with a previously unknown identifier μ is pushed to the out-queue towards its destination node ι_2. Sending this message is recorded in Δ.

M-Route Routing a message to the next node. A message m is at the top of the in-queue at node ι_1 and was forwarded from node ι_2. However, ι_1 is not the final destination of m. Consequently, the message is dispatched from that in-queue to an out-queue towards a node closer to Dest(m).

M-Forward Forwarding a message to the next node. The message m has been newly created or routed at node ι_1 and was pushed to the out-queue for ι_2. Messages on out-queues are always delivered to the next node on the path from ι_1 to the destination of m. Note that both routing and forwarding are not recorded within the history Δ.

M-Receive Receiving a message. The message m meant for node ι_1 has reached the head of some in-queue at ι_1. Therefore, execution for delivering m is finished. The message is consumed by popping it from the respective queue and its receipt is recorded in Δ.

To illustrate introduced runtime entities, we refer to the example application discussed in section 2, figure 1. The initial configuration, before forwarding "*credit*" (m_0) looks like the following:

$$\mathcal{T} = [\iota_1 \to \iota_2, \iota_2 \to \iota_3, \iota_3 \to \iota_3, \iota_4 \to \iota_3]$$
$$n_1 = [\iota_2 \to (\epsilon, m_0 \cdot \epsilon)]$$
$$n_2 = [\iota_1 \to (\epsilon, \epsilon), \iota_3 \to (\epsilon, \epsilon))]$$
$$n_3 = [\iota_2 \to (\epsilon, \epsilon), \iota_4 \to (\epsilon, \epsilon)]$$
$$n_4 = [\iota_3 \to (\epsilon, \epsilon)]$$
$$\Sigma = [\iota_1 \to n_1, \iota_2 \to n_2, \iota_3 \to n_3, \iota_4 \to n_4]$$
$$\Delta = \text{Snd}(m_0, \iota_1)$$

Assuming the messages "buy" and "debit" are given the identifiers m_1 and m_2, a (possible) configuration right before the *Shop* asks the *Bank* for money is:

$$\mathcal{T} = [\iota_1 \to \iota_2, \iota_2 \to \iota_3, \iota_3 \to \iota_3, \iota_4 \to \iota_3]$$
$$n_1 = [\iota_2 \to (\epsilon, \epsilon)]$$
$$n_2 = [\iota_1 \to (\epsilon, \epsilon), \iota_3 \to (\epsilon, \epsilon))]$$
$$n_3 = [\iota_2 \to (\epsilon, \epsilon), \iota_4 \to (\epsilon, \epsilon)]$$
$$n_4 = [\iota_3 \to (\epsilon, m_2 \cdot \epsilon)]$$
$$\Sigma = [\iota_1 \to n_1, \iota_2 \to n_2, \iota_3 \to n_3, \iota_4 \to n_4]$$
$$\Delta = \text{Snd}(m_0, \iota_1) \cdot \text{Snd}(m_1, \iota_1) \cdot \text{Rec}(m_0) \cdot \text{Rec}(m_1) \cdot \text{Snd}(m_2, \iota_4)$$

We can observe for this instance of the applications execution that both m_0 and m_1 were already received before m_2 was sent at ι_4. Moreover m_0 was received before m_1 was. Consequently, it is impossible for m_2 to arrive at ι_3 after the next configuration that would follow from the application of the operational semantics. That is, causal message delivery is preserved. Note that the delay between sending messages can be arbitrary. Message m_0 could also have been received before m_1 was even sent. In fact, for this particular example scenario, there are *two* possible instances of Δ:

$$\Delta_1 = \text{Snd}(m_0, \iota_1) \cdot \text{Rec}(m_0) \cdot \text{Snd}(m_1, \iota_1) \cdot \text{Rec}(m_1) \cdot \text{Snd}(m_2, \iota_4) \cdot \text{Rec}(m_2)$$
$$\Delta_2 = \text{Snd}(m_0, \iota_1) \cdot \text{Snd}(m_1, \iota_1) \cdot \text{Rec}(m_0) \cdot \text{Rec}(m_1) \cdot \text{Snd}(m_2, \iota_4) \cdot \text{Rec}(m_2)$$

The only difference between the two histories is the point in time at which m_0 is received, as m_0 is sent asynchronously and there is no upper bound on when to send m_1 after m_0 has been put on the out-queue towards ι_1. However, the order of $\text{Snd}(m_0, \iota_1)$ and $\text{Snd}(m_1, \iota_1)$ cannot be different, as m_0 *causes* m_1. The same applies to the order of receiving m_1 and sending m_2. This leaves us with only Δ_1 and Δ_2 as possible histories for the scenario illustrated in figure 1.

We now introduce several definitions to form the basis of our formal argument. We show that the order of messages delivery in our example application is causal in *any* well formed tree, state and configuration and can be observed in any possible recording of Δ.

3.2 Well-Formed Trees, Paths and Next Hops

In this section we introduce several definitions we will be using throughout the discussion of the formal system and proofs. At first, we discuss the basics of children and descendants as well as our definition of paths and next hops in tree networks.

$$
\begin{array}{llll}
\iota & \in & \text{NodeId} & ::= \mathbb{N} \\
\mu & \in & \text{MessageId} & ::= \mathbb{N} \\
m & \in & \text{Message} & ::= \text{MessageId} \times \text{NodeId} \\
q & \in & \text{Queue} & ::= \text{Message}* \\
\mathcal{T} & \in & \text{Tree} & ::= \text{NodeId} \rightarrow \text{NodeId} \\
n & \in & \text{Node} & ::= \text{NodeId} \rightarrow \text{Queue} \times \text{Queue} \\
\Sigma & \in & \text{State} & ::= \text{NodeId} \rightarrow \text{Node} \\
e & \in & \text{Event} & ::= \text{Snd}(m, \iota) \\
 & & & \mid \text{Rec}(m) \\
\Delta & \in & \text{History} & ::= \text{Event}* \\
p & \in & \text{Path} & ::= \iota \cdot p
\end{array}
$$

M-Send

$$\frac{\text{Snd}((\mu, \_), \_) \notin \Delta}{\mathcal{T} \vdash \Delta, \Sigma \rightsquigarrow \Delta \cdot \text{Snd}((\mu, \iota_2), \iota_1), \Sigma[\iota_1 \mapsto \text{PushOut}(\Sigma, \mathcal{T}, \iota_1, (\mu, \iota_2))]}$$

M-Route

$$\frac{\text{TopIn}(\Sigma, \iota_1, \iota_2) = m}{\mathcal{T} \vdash \Delta, \Sigma \rightsquigarrow \Delta, \Sigma[\iota_1 \mapsto \text{PopIn}(\Sigma, \iota_1, \iota_2)], [\iota_2 \mapsto \text{PushOut}(\Sigma, \mathcal{T}, \iota_1, m)]}$$

M-Forward

$$\frac{\text{TopOut}(\Sigma, \iota_1, \iota_2) = m}{\mathcal{T} \vdash \Delta, \Sigma \rightsquigarrow \Delta, \Sigma[\iota_1 \mapsto \text{PopOut}(\Sigma, \iota_1, \iota_2), \iota_2 \mapsto \text{PushIn}(\Sigma, \iota_1, \iota_2, m)]}$$

M-Receive

$$\frac{\text{TopIn}(\Sigma, \iota_1, \iota_2) = (\mu, \iota_1)}{\mathcal{T} \vdash \Delta, \Sigma \rightsquigarrow \Delta \cdot \text{Rec}((\mu, \iota_1)), \Sigma[\iota_1 \mapsto \text{PopIn}(\Sigma, \iota_1, \iota_2)]}$$

Figure 3. Runtime entities and operational semantics

Definition 3.1. (Children and descendants). Given $cfg = (\mathcal{T}, \Sigma)$, we define:

- The children of ι in $\mathcal{T}$ is the set of nodes which are in the range of $\mathcal{T}(\iota)$, except ι itself:
  - $\text{children}(\iota, \mathcal{T}) = \{\iota' | \mathcal{T}(\iota') = \iota \wedge \iota' \neq \iota\}$
- The descendants of a node ι in $\mathcal{T}$ are described as the transitive closure of the child set:
  - $\text{descendants}(\iota, \mathcal{T}) = \{\iota' | \exists \iota''. \iota'' \in \text{children}(\iota, \mathcal{T}) \wedge \iota' \in \text{descendants}(\iota'', \mathcal{T})\} \cup \text{children}(\iota, \mathcal{T})$

Figure 4 shows how to compute the next hop towards a node ι_2 from ι_1. The next hop is trivial to decide if ι_2 is in the child set of ι_1. Similarly, if ι_2 is the root node, the next hop is the parent of ι_1. In the general case, the next node can be decided by determining the next hop from $\iota_1$to the parent of ι_2.

The concatenation of next hops from ι_1 to ι_2 is called the *path* from ι_1 to ι_2, as defined in figure 5. As mentioned in section 2, path inclusion is one of the important pieces to solve the puzzle why message delivery in tree networks is causal. Formalized in figure 6, throughout the discussion of this paper, we refer to it as a path p is *totally included* in a path p' if p is the suffix of the non-empty sequence p'.

3.3 Well-Formed Configuration

Definition 3.2. (A well-formed tree). We say that a tree $\mathcal{T}$ is well-formed, formally $\models \mathcal{T}$, if there are no cycles and if there is exactly one root node:

- $\forall \iota \in \text{dom}(\mathcal{T}). \iota \notin \text{descendants}(\iota, \mathcal{T})$
- $\exists! \iota \in \text{dom}(\mathcal{T}). \mathcal{T}(\iota) = \iota$

$$\text{next} ::= \text{NodeId} \times \text{NodeId} \times \mathcal{T} \rightarrow \text{NodeId}$$

$$\text{next}(\iota_1, \iota_2, \mathcal{T}) = \begin{cases} \iota_2 & \text{if} \quad \iota_2 \in \text{children}(\iota_1, \mathcal{T}) \\ \mathcal{T}(\iota_1) & \text{if} \quad \iota_2 = \mathcal{T}(\iota_2) \\ \text{next}(\iota_1, \mathcal{T}(\iota_2), \mathcal{T}) & \text{otherwise} \end{cases}$$

Figure 4. Next hop towards a destination node

$$\text{path} ::= \text{NodeId} \times \text{NodeId} \times \mathcal{T} \rightarrow \text{NodeId}*$$

$$\text{path}(\iota_1, \iota_2, \mathcal{T}) = \begin{cases} \iota_1 \cdot \text{path}(\text{next}(\iota_1, \iota_2, \mathcal{T}), \iota_2, \mathcal{T}) & \text{if } \iota_1 \neq \iota_2 \\ \iota_1 & \text{if } \iota_1 = \iota_2 \end{cases}$$

Figure 5. Paths in a tree

$$\frac{\bar{\iota} \neq \epsilon}{p \sqsubset \bar{\iota} \cdot p} \quad \text{P-INCL}$$

Figure 6. Path inclusion

Definition 3.3. (A well-formed state). We say that the state Σ is well formed, formally $\models \Sigma$, if a message only ever appears in exactly one message queue and if the destination of every message is a valid node in Σ:

- $\forall m \in \Sigma. \exists! q \in \Sigma. m \in q$

$\prec_\Delta \subseteq$ Event × Event

$e \prec_\Delta e'$
iff
$\Delta = \_ \cdot e \cdot \_ \cdot e' \cdot \_$

$\prec_\Delta \subseteq$ Message × Message

$m \prec_\Delta m'$
iff
$\mathrm{Rec}(m) \prec_\Delta \mathrm{Snd}(m', \mathrm{Dest}(m))$
$\vee$
$\exists \iota.\, \mathrm{Snd}(m, \iota) \prec_\Delta \mathrm{Snd}(m', \iota)$

$\ll_\Delta \subseteq$ Message × Message

$m \ll_\Delta m'$
iff
$m \prec_\Delta m'$
$\vee$
$\exists m''.\ (\ m \ll_\Delta m'' \wedge m'' \prec_\Delta m'\ )$

Figure 7. Causally dependent events and messages

$\mathcal{T}, \Delta, \Sigma \vdash m < m'$ iff

1. $\exists q \in \Sigma.\quad q = \_ \cdot m \cdot \_ \cdot m' \cdot \_$
$\vee$
2. $\exists \iota, \iota_1, \iota_2 \in \Sigma.[m \in \mathrm{Out}(\Sigma, \iota, \iota_1) \wedge m' \in \mathrm{In}(\Sigma, \iota, \iota_2)]$
$\vee$
3. $\exists \iota_1, \iota_2 \in \Sigma\, [m \in \Sigma(\iota_1) \wedge m' \in \Sigma(\iota_2) \wedge \iota_1 \neq \iota_2 \wedge \mathrm{path}(\iota_1, m, \mathcal{T}) \sqsubset \mathrm{path}(\iota_2, \mathrm{Dest}(m), \mathcal{T})]$
$\vee$
4. $\mathrm{Dest}(m) \neq \mathrm{Dest}(m')\ \wedge\ [\ \mathrm{Rec}(m) \in \Delta \vee \mathrm{Rec}(m') \in \Delta\ ]$
$\vee$
5. $\mathrm{Dest}(m) = \mathrm{Dest}(m')\ \wedge\ [\ \mathrm{Rec}(m') \in \Delta\ \rightarrow\ \mathrm{Rec}(m) \prec_\Delta \mathrm{Rec}(m')\ ]$

Figure 8. Message arrives at its destination before another message

- $\forall \mu, \iota\, [(\mu, \iota) \in \Sigma \rightarrow \iota \in \mathrm{dom}(\Sigma)]$

Definition 3.4. (A well-formed state for a given tree). We say that the state Σ is well formed with respect to $\mathcal{T}$, formally $\mathcal{T} \models \Sigma$, if all queues point to valid nodes, and if the state only contains valid nodes. All messages reside in a queue which is on a node along the route a message has to travel in order to reach its destination (figures 4 and 5). That is, formally:

- $\mathrm{dom}(\Sigma) = \mathrm{dom}(\mathcal{T})$
- $\forall \iota, \iota' \in \Sigma.[\Sigma(\iota, \iota')$ is defined
  $\rightarrow \iota' \in (\mathrm{children}(\iota, \mathcal{T}) \cup \mathrm{children}(\iota', \mathcal{T}))]$
- $\forall \iota_1, \iota_2 \in \Sigma. \forall m[m \in \mathrm{Out}(\Sigma, \iota_1, \iota_2)$
  $\rightarrow \mathrm{next}(\iota_1, \mathrm{Dest}(m), \mathcal{T}) = \iota_2]$
- $\forall \iota_1, \iota_2 \in \Sigma. \forall m[m \in \mathrm{In}(\Sigma, \iota_1, \iota_2)$
  $\rightarrow \mathrm{next}(\iota_2, \mathrm{Dest}(m), \mathcal{T}) = \iota_1]$

Definition 3.5. (A well formed history). We say that the history Δ is well formed, formally $\models \Delta$, if a message is exactly sent and received once:

- $\forall \mu\, [\#\{k | \Delta[k] = \mathrm{Rec}(\mu, \_)\} \leq 1 \wedge$
  $\#\{k | \Delta[k] = \mathrm{Snd}(\mu, \_), \_)\} \leq 1]$

Causally dependent messages. We now introduce a formal counterpart, the relation $\prec_\Delta$, to express that m caused m'. With $\prec_\Delta$ of figure 7 we apply an order on the events in Δ and combine all three relations to give a transitive version of $\prec_\Delta$, namely $\ll_\Delta$ (figure 7). Hence, a message m *causes* another message m' if any of the three cases apply:

- m was received prior to m' being sent from the node where m was received.
- m and m' have been sent from the same node, in that order. That is, the sending of m *preceded* the sending of m'.
- There exists a message m'' such that $\prec_\Delta$ can be established transtively using m''.

Definition 3.6. (A well formed state and path for a given history). We say that the state Σ is well formed for a given history Δ, formally $\Delta \models \Sigma$, if for every message in the state there exists a Snd event recorded in the history, and, if a message was received, the corresponding Snd event must have been observed in Δ earlier with respect to $\prec_\Delta$. Also, messages can still be in transit:

- $\forall m\, [m \in \Sigma \rightarrow \exists \iota'\, \mathrm{Snd}(m, \iota') \in \Delta]$
- $\forall m \in \Sigma\, [\mathrm{Rec}(m) \in \Delta \rightarrow \mathrm{Snd}(m, \_) \prec_\Delta \mathrm{Rec}(m)]$
- $\forall m \in \Sigma\, [\mathrm{Snd}(m, \_) \in \Delta \wedge \mathrm{Rec}(m) \notin \Delta]$

The path of a message is well formed, formally $\mathcal{T}, \Delta, \Sigma \models m$, if it does not take any unnecessary detours towards its destination, with respect to path inclusion (figure 6 and 5):

- $\forall \iota, \iota'.\, [m \in \Sigma(\iota) \wedge \mathrm{Snd}(m, \iota') \in \Delta$
  $\rightarrow \mathrm{path}(\iota, \mathrm{Dest}(m), \mathcal{T}) \sqsubset \mathrm{path}(\iota', \mathrm{Dest}(m), \mathcal{T})]$

Definition 3.7. (A well-formed configuration). We say that a configuration $\mathrm{cfg} = (\mathcal{T}, \Delta, \Sigma)$ is well formed, formally WF(cfg), if the properties described in definitions 3.2 to 3.6 hold:

1. $\models \mathcal{T}$
2. $\models \Sigma$
3. $\models \Delta$
4. $\mathcal{T} \models \Sigma$

5. $\Delta \models \Sigma$
6. $\forall m \in \Sigma . \mathcal{T}, \Delta, \Sigma \models m$

3.4 Cause and Arrival

We now define what it means for a message to be a cause of another message (figure 7), and what it means for a message to arrive before another one (figure 8).

We define the relation $\ll_\Delta$ in figure 7, where $m \ll_\Delta m'$ means that message m is a cause of message m'. The definition reflects the informal description of causality given at the beginning of section 2. It is easy to see that $\ll_\Delta$ is transitive and that $m \not\ll_\Delta m$ never holds. In a Δ_0 describing the example from section 2, we would have that $Credit \ll_{\Delta_0} Buy \ll_{\Delta_0} Debit$.

We then define a relation to express when a message m is guaranteed to "*arrive at its destination before*" another message m', formally $m < m'$. The relation depends on the tree topology $\mathcal{T}$, the history Δ and state Σ. As described in in figure 8, the relation $\mathcal{T}, \Delta, \Sigma \vdash m < m'$ holds if any of the following cases apply:

1. m and m' are in the same queue, and m precedes m' in that queue.
2. The two messages are on the same node, but m already resides on an out-queue towards the *destination* of m', whereas m' is still in an in-queue from some child node.
3. The two messages are on two different nodes, but the path of m is fully included in the path of m' towards their, potentially individual, destinations.
4. The two messages have different destinations, and one of them has already been received.
5. The two messages have the same destination, and if m' has been received, then m was received before it.

To be precise, $m_1 < m_2$ means that m_1 is guaranteed to arrive at its destination *before* m_2 will arrive at m_1's destination. Therefore, the relation is applicable even if the two messages do not have the same destination, even if the second message will never visit the first message's destination, even if they should not share any part of the route. For example, assuming the topology of figure 2, and that the "Credit" message was in the out-queue of ι_2, and the "Buy" message was at the in-queue of ι_4. Then we would have that "Credit" and "Buy" have different destinations, and their routes have nothing in common – nevertheless, we have that $Credit < Buy$. Interestingly, we here also have that $Buy < Credit$.

Notice that while $m < m$ is never possible, the relation $<$ is neither symmetric, nor antisymmetric, nor transitive. To see an example for the latter, consider the topology on figure 2, and messages *msg_A*, *msg_B*, and *msg_C*, where *msg_A* has destination $\iota 3$ and is on the out-queue of node $\iota 2$ towards $\iota 3$, while *msg_B* has destination $\iota 1$ and is on the out-queue of node $\iota 2$ towards $\iota 1$, while *msg_C* has destination $\iota 2$ and is on the out-queue of node $\iota 4$ towards $\iota 3$. Then, we have that *msg_A* $<$ *msg_B* and *msg_B* $<$ *msg_C*, but we do not have that *msg_A* $<$ *msg_C*.

We require that causality implies an earlier arrival – *c.f.,* the definition of consistent configurations in the next paragraph.

Consistent configurations We say that a configuration $(\mathcal{T}, \Delta, \Sigma)$ is *consistent* if it is well-formed (definition 3.7), and for any causally related messages m and m' m is guaranteed to arrive at its destination *before* m' arrives at the destination of the former.

Definition 3.8.

$\text{Consistent}(\mathcal{T}, \Delta, \Sigma)$ iff
- $\text{WF}(\mathcal{T}, \Delta, \Sigma) \wedge$
- $\forall m, m' . [m \ll_\Delta m' \rightarrow \mathcal{T}, \Delta, \Sigma \vdash m < m']$

4 Consistency Implies Causality

Theorem 4.1 expresses the key property of our system: It guarantees that indeed in a tree topology messages are delivered in causal order. Namely, in a consistent runtime configuration, if after a number of execution steps, a message m is a cause of another message m', and m' has already been delivered, and the two messages have the same destination, then m' will also have been delivered.

Theorem 4.1 (Message delivery is causal)**.**

$$\begin{array}{ll} Consistent(\mathcal{T}, \Delta, \Sigma) & \wedge \\ \mathcal{T} \vdash \Delta, \Sigma \overset{*}{\rightsquigarrow} \Delta', \Sigma' & \wedge \\ m \ll_{\Delta'} m' & \wedge \\ Dest(m) = Dest(m') & \wedge \\ Rec(m) \in \Delta' & \\ \rightarrow & \\ Rec(m) \prec_{\Delta'} Rec(m') & \end{array}$$

Proof sketch Theorem 4.1 follows directly from lemma 4.2, which guarnantees that in a consistent configuration, messages which have the same destination will be delivered in causal order, and from lemma 4.6, which guatantees that execution preserves consistency. Both lemmas are stated below. In a companion technical report in preparation we have longer proofs – here we give proof sketches.

Lemma 4.2. *Consistency implies Causality*

$$\begin{array}{ll} Consistent(\mathcal{T}, \Delta, \Sigma) & \wedge \\ Dest(m) = Dest(m') & \wedge \\ m \ll_\Delta m' & \wedge \\ Rec(m) \in \Delta & \\ \rightarrow & \\ Rec(m) \prec_\Delta Rec(m') & \end{array}$$

Proof. By application of definition $\text{Consistent}(\mathcal{T}, \Delta, \Sigma)$, and because $m \ll_\Delta m'$, we know that $\mathcal{T}, \Delta, \Sigma \vdash m < m'$. Because $\text{Rec}(m) \in \Delta$, only case 5 of the definition of $\mathcal{T}, \Delta, \Sigma \vdash m < m'$ is applicable, and this gives that $\text{Rec}(m) \prec_\Delta \text{Rec}(m')$

□

We will now prove that execution preserves consistency of configurations. Lemma 4.3 guarantees that execution starting from a well-formed configuration leads to another well-formed configuration. Lemma 4.4 guarantees that execution starting from a well-formed configuration preserves the ordering of messages.

Lemma 4.3 (Execution preserves well-formedness)**.**

$$WF(\mathcal{T}, \Delta, \Sigma) \ \wedge \ \mathcal{T} \vdash \Delta, \Sigma \rightsquigarrow \Delta', \Sigma' \ \rightarrow \ WF(\mathcal{T}, \Delta', \Sigma')$$

Proof. By case analysis over $\mathcal{T} \vdash \Delta, \Sigma \rightsquigarrow \Delta', \Sigma'$ and application of the definition of $\mathrm{WF}(\mathcal{T}, \Delta, \Sigma)$ □

Lemma 4.4 (Message order is preserved)**.**

$$WF(\mathcal{T}, \Delta, \Sigma) \ \wedge \ \mathcal{T} \vdash \Delta, \Sigma \rightsquigarrow \Delta', \Sigma' \ \wedge \ \mathcal{T}, \Delta, \Sigma \vdash m < m' \ \rightarrow \ \mathcal{T}, \Delta', \Sigma' \vdash m < m'$$

Proof. By case analysis over $\mathcal{T} \vdash \Delta, \Sigma \rightsquigarrow \Delta', \Sigma'$. □

Lemma 4.5 characterises the causal dependencies which can be derived from $\Delta \cdot e$ but not from Δ.

Lemma 4.5 (Extending causal dependencies)**.**

$$\begin{array}{ll} \neg(m \ll_{\Delta} m') & \wedge \\ m \ll_{\Delta'} m' & \wedge \\ \Delta' = \Delta \cdot e & \wedge \\ \rightarrow & \\ [e = Snd(m', \iota) \vee e = Rec(m')] & \wedge \\ [m <_{\Delta} m' \vee \exists m''.m \ll_{\Delta} m'' \wedge m'' <_{\Delta'} m'] & \end{array}$$

Lemma 4.6 guarantees that any number of execution steps preserves consistency.

Lemma 4.6 (Mulitple step execution preserves consistency)**.**

$$Consistent(\mathcal{T}, \Delta, \Sigma) \ \wedge \ \mathcal{T} \vdash \Delta, \Sigma \overset{*}{\rightsquigarrow} \Delta', \Sigma' \ \rightarrow \ Consistent(\mathcal{T}, \Delta', \Sigma')$$

Proof. By induction on the number of steps in $\overset{*}{\rightsquigarrow}$. For 0 steps it is trivial, for 1 step by application of lemmas 4.3, 4.4, and 4.5, and for more than 1 step by straightforward induction. □

5 Conclusion and Future Work

We have shown that distributed systems arranged in a tree topology are causal by construction. Specifically, we:

- Provided a mechanism to enforce causal ordering without the need for augmenting messages or storing meta data,
- Proposed a formal model to capture message passing in tree networks,
- Outlined proof sketch that causality is ensured.

In recent years we have worked on the development of an actor-based programming language called *Pony* [8, 10, 11, 28]. In the current distribution, Pony is concurrent, but our aim is to develop it so as to provide *transparent* distributed programming: so that the notion of distribution is not exposed to the programmer. The result of this paper is central to the development of Distributed Pony: arranging the nodes into a tree topology,

In further work we want to study more dynamic distributed systems, which support actor migration, the addition (and potentially exchange) of nodes at runtime, as well as distributed snapshotting. A preliminary outlook on such mechanisms appeared in [8].

We believe that causal message delivery simplifies programming considerably, and thus our result is of wider importance. For instance, causal message delivery (along with types) has been leveraged for fully concurrent Garbage Collection [21].

On the other hand, enforcing causality by a topological arrangement of nodes is, at least for some applications, a case of trading space for time [24]. Therefore, we plan to also investigate the negative effects and runtime/space complexity tree topologies might have on messaging and routing performance, what possible optimizations are and how partial trees can be used to explicitly specify causal relationships to reduce the routing overhead on totally unrelated messages.

A significant open question is the handling of failures. What will recovery look like in a tree network? What protocols can be put in place to allow for topology changes? How does the ability to enforce causality without the ability to determine causal relations (as in our system), affect failure recovery? We plan to draw inspiration from distributed snapshots [4, 17].

A Auxiliary Functions

$$\begin{array}{lcl} \mathrm{Dest} & ::= & \mathrm{Message} \rightarrow \mathrm{NodeId} \\ \mathrm{In} & ::= & \mathrm{State} \times \mathrm{NodeId} \times \mathrm{NodeId} \rightarrow \mathrm{Message} \\ \mathrm{Out} & ::= & \mathrm{State} \times \mathrm{NodeId} \times \mathrm{NodeId} \rightarrow \mathrm{Message} \\ \mathrm{path} & ::= & \mathrm{NodeId} \times \mathrm{Message} \times \mathcal{T} \rightarrow \mathrm{NodeId}* \\ \mathrm{next} & ::= & \mathrm{NodeId} \times \mathrm{Message} \times \mathcal{T} \rightarrow \mathrm{NodeId} \end{array}$$

$$\begin{array}{lcl} \mathrm{Dest}(m) & = & m \downarrow_2 \\ \mathrm{In}(\Sigma, \iota_1, \iota_2) & = & \Sigma(\iota_1)(\iota_2) \downarrow_1 \\ \mathrm{Out}(\Sigma, \iota_1, \iota_2) & = & \Sigma(\iota_1)(\iota_2) \downarrow_2 \\ \mathrm{path}(\iota, m, \mathcal{T}) & = & \mathrm{path}(\iota, \mathrm{Dest}(m), \mathcal{T}) \\ \mathrm{next}(\iota, m, \mathcal{T}) & = & \mathrm{next}(\iota, \mathrm{Dest}(m), \mathcal{T}) \end{array}$$

Figure 9. Shorthands

$$\begin{array}{lcl} \mathrm{TopIn} & ::= & \mathrm{State} \times \mathrm{NodeId} \times \mathrm{NodeId} \rightarrow \mathrm{Message} \\ \mathrm{TopOut} & ::= & \mathrm{State} \times \mathrm{NodeId} \times \mathrm{NodeId} \rightarrow \mathrm{Message} \\ \mathrm{PushIn} & ::= & \mathrm{State} \times \mathrm{NodeId} \times \mathrm{NodeId} \times \mathrm{Message} \rightarrow \mathrm{Node} \\ \mathrm{PushOut} & ::= & \mathrm{State} \times \mathrm{Tree} \times \mathrm{NodeId} \times \mathrm{Message} \rightarrow \mathrm{Node} \\ \mathrm{PopIn} & ::= & \mathrm{State} \times \mathrm{NodeId} \times \mathrm{NodeId} \rightarrow \mathrm{Node} \\ \mathrm{PopOut} & ::= & \mathrm{State} \times \mathrm{NodeId} \times \mathrm{NodeId} \rightarrow \mathrm{Node} \end{array}$$

Figure 10. Auxiliary functions - declarations

$$
\begin{array}{lcll}
\text{TopIn}(\Sigma, \iota_1, \iota_2) & = & m & \text{if } \text{In}(\Sigma, \iota_1, \iota_2) = m \cdot \_ \\
\text{TopOut}(\Sigma, \iota_1, \iota_2) & = & m & \text{if } \text{Out}(\Sigma, \iota_1, \iota_2) = m \cdot \_ \\
\text{PushIn}(\Sigma, \iota_1, \iota_2, m) & = & \Sigma(\iota_1)[\iota_2 \mapsto \text{In}(\Sigma, \iota_1, \iota_2) \cdot m, \text{Out}(\Sigma, \iota_1, \iota_2)] & \\
\text{PushOut}(\Sigma, \mathcal{T}, \iota, m) & = & \Sigma(\iota)[\iota' \rightarrow \text{In}(\Sigma, \iota, \iota'), \text{Out}(\Sigma, \iota, \iota') \cdot m] & \\
 & & \text{where } \iota' = \text{next}(\iota, m, \mathcal{T}) & \\
\text{PopIn}(\Sigma, \iota_1, \iota_2) & = & \Sigma(\iota_1)[\iota_2 \mapsto (q, \text{Out}(\Sigma, \iota_1, \iota_2)] & \\
 & & \text{where } \text{In}(\Sigma, \iota_1, \iota_2) = m \cdot q & \\
\text{PopOut}(\Sigma, \iota_1, \iota_2) & = & \Sigma(\iota_1)[\iota_2 \mapsto (\text{In}(\Sigma, \iota_1, \iota_2), q] & \\
 & & \text{where } \text{Out}(\Sigma, \iota_1, \iota_2) = m \cdot q &
\end{array}
$$

Figure 11. Auxiliary functions - definitions

References

[1] M. Ahamad, P. W. Hutto, and R. John. 1991. Implementing and programming causal distributed shared memory. In *[1991] Proceedings. 11th International Conference on Distributed Computing Systems*. 274–281. https://doi.org/10.1109/ICDCS.1991.148677

[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (01 Mar 1995), 37–49. https://doi.org/10.1007/BF01784241

[3] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. 2008. *Interval Tree Clocks*.

[4] Özalp Babaoğlu and Keith Marzullo. 1993. Distributed Systems (2Nd Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, Chapter Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, 55–96. http://dl.acm.org/citation.cfm?id=302430.302434

[5] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2012. The Potential Dangers of Causal Consistency and an Explicit Solution. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, Article 22, 7 pages. https://doi.org/10.1145/2391229.2391251

[6] Kenneth Birman, André Schiper, and Pat Stephenson. 1991. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.* 9, 3 (Aug. 1991), 272–314. https://doi.org/10.1145/128738.128742

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

[8] Sebastian Blessing. 2013. A String of Ponies - Transparent Distributed Programming with Actors. (2013). https://www.doc.ic.ac.uk/~scb12/distinguished-projects/2013/s.blessing.pdf

[9] Bernadette Charron-Bost. 1991. Concerning the Size of Logical Clocks in Distributed Systems. *Inf. Process. Lett.* 39, 1 (July 1991), 11–16. https://doi.org/10.1016/0020-0190(91)90055-M

[10] Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully Concurrent Garbage Collection of Actors on Many-core Machines. *SIGPLAN Not.* 48, 10 (Oct. 2013), 553–570. https://doi.org/10.1145/2544173.2509557

[11] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2824815.2824816

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[13] C. J. Fidge. February, 1988. Timestamps In Message-Passing Systems That Preserve The Partial Ordering. (February, 1988).

[14] Alexey Gotsman, Hongseok Yang, Marek Zawirski, and Sebastian Burckhardt. 2014. Replicated Data Types: Specification, Verification, Optimality. https://www.microsoft.com/en-us/research/publication/replicated-data-types-specification-verification-optimality/

[15] M. P. Herlihy and J. M. Wing. 1987. Axioms for Concurrent Objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/41625.41627

[16] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[17] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

[18] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. 1983. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Trans. Softw. Eng.* 9, 3 (May 1983), 240–247. https://doi.org/10.1109/TSE.1983.236733

[19] Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. 2010. Dotted Version Vectors: Logical Clocks for Optimistic Replication. *CoRR* abs/1011.5808 (2010). http://arxiv.org/abs/1011.5808

[20] M. Raynal and M. Singhal. 1996. Logical time: capturing causality in distributed systems. *Computer* 29, 2 (Feb 1996), 49–56. https://doi.org/10.1109/2.485846

[21] S. Clebsch, J.P. Franco, S. Drossopoulou, A. M. Yang, T. Wrigstad, and J. Vitek. 2017. Orca: GC and Type System Co-Design for Actor Languages. In *OOPSLA*. https://doi.org/10.1145/3133896

[22] André Schiper, Jorge Eggli, and Alain Sandoz. 1989. A New Algorithm to Implement Causal Ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*. Springer-Verlag, London, UK, UK, 219–232. http://dl.acm.org/citation.cfm?id=645946.675010

[23] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distrib. Comput.* 7, 3 (March 1994), 149–174. https://doi.org/10.1007/BF02277859

[24] Mark Stamp. Accessed July 2017. Once Upon a Time-Memory Tradeoff. (Accessed July 2017). http://www.cs.sjsu.edu/~stamp/RUA/TMTO.pdf

[25] Doug Terry. 2011. *Replicated Data Consistency Explained Through Baseball*. Technical Report. https://www.microsoft.com/en-us/research/publication/replicated-data-consistency-explained-through-baseball/

[26] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS '94)*. IEEE Computer Society, Washington, DC, USA, 140–149. http://dl.acm.org/citation.cfm?id=645792.668302

[27] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 172–182. https://doi.org/10.1145/224056.224070

[28] The Pony Core Development Team. 2013 - today. Pony. (2013 - today). http://www.ponylang.org

[29] Robbert van Renesse. 1993. Causal Controversy at Le Mont St.-Michel. *Operating Systems Review* 27, 2 (1993), 44–53. https://doi.org/10.1145/155848.155857