Verified Causal Broadcast with Liquid Haskell

PATRICK REDMOND, University of California, Santa Cruz, USA
GAN SHEN, University of California, Santa Cruz, USA
NIKI VAZOU, IMDEA, Spain
LINDSEY KUPER, University of California, Santa Cruz, USA

Protocols to ensure that messages are delivered in causal order are a ubiquitous building block of distributed systems. For instance, key-value stores can use causally ordered message delivery to ensure causal consistency — a sweet spot in the availability/consistency trade-off space — and replicated data structures rely on the existence of an underlying causally-ordered messaging layer to ensure that geo-distributed replicas eventually converge to the same state. A causal delivery protocol ensures that when a message is delivered to a process, any causally preceding messages sent to the same process have already been delivered to it. While causal message delivery protocols are widely used in distributed systems, verification of the correctness of those protocols is less common, much less machine-checked proofs about executable implementations.

We implemented a standard causal broadcast protocol in Haskell and used the Liquid Haskell solver-aided verification system to express and mechanically prove the causal safety of our implementation: messages will never be delivered to a process in an order that violates causality. We express this safety property using refinement types and prove that it holds using Liquid Haskell’s theorem-proving facilities, resulting in the first machine-checked proof of correctness of an executable causal broadcast implementation. Our proof is simple and intuitive, consisting of only a dozen lines of Liquid Haskell code, and gives insight into why the protocol works. We then put our verified causal broadcast implementation to work as the foundation of a causally-consistent distributed key-value store implemented in Haskell.

1 INTRODUCTION

Causal message delivery [Birman and Joseph 1987a; Birman et al. 1991; Birman and Joseph 1987b,c; Schiper et al. 1989] is a fundamental communication abstraction for distributed computations in which processes communicate by sending and receiving messages. One of the challenges of implementing distributed systems is the asynchrony of message delivery; messages arriving at the recipient in an unexpected order can cause confusion and bugs. A causal delivery protocol can help by ensuring that, when a message \( m \) is delivered to a process \( p \), any message sent “before” \( m \) (in the sense of Lamport’s "happens-before"; see Section 2) will have already been delivered to \( p \). When a mechanism for causal message delivery is available, it simplifies the implementation of many important distributed algorithms, such as replicated data stores that must maintain causal consistency [Ahamad et al. 1995; Lloyd et al. 2011], conflict-free replicated data types [Shapiro et al. 2011b], distributed snapshot protocols [Acharya and Badrinath 1992; Alagar and Venkatesan 1994], and applications that "involve human interaction and consist of large numbers of communication endpoints" [van Renesse 1993].

A particularly useful special case of causal delivery is causal broadcast, in which each message is sent to all processes in the system. An underlying causal broadcast protocol enables a straightforward implementation strategy for a causally consistent replicated data store — one of the strongest consistency models available for applications that must maximize availability and tolerate network partitions [Mahajan et al. 2011]. Conflict-free replicated data types (CRDTs) implemented in the operation-based style [Gomes et al. 2017; Shapiro et al. 2011a,b] also assume the existence of an underlying causal broadcast layer to deliver updates to replicas [Shapiro et al. 2011b, §2.4].

What can go wrong in the absence of causal broadcast? Consider a scenario in which Alice, Bob, and Carol are exchanging group text messages. Suppose Alice sends the message "I lost my wallet..." to the group. A little while later, Alice finds her missing wallet between her couch cushions and follows up with a "Found it!” message to the group. Alice has a reasonable expectation that Bob
and Carol will see the messages in the order that she sent them; otherwise, they might be rather confused by the message contents, as Carol is in Figure 1 (left) when she sees the “Found it!” message before seeing the original “I lost my wallet...” message. Alice’s expectation of first-in first-out (FIFO) delivery — in which all messages from a given sender to a given recipient are delivered at the recipient in the order they were sent — is an aspect of causal message ordering, and is enforced by standard networking protocols such as TCP [Postel 1981]. If an application is running on top of such a FIFO-enforcing networking protocol (as is typical), then there is no need for any additional mechanism to enforce FIFO delivery at a higher level of the stack.

Unfortunately, FIFO delivery is not enough to eliminate all violations of causality. Consider an execution like that in Figure 1 (right), which is a standard example from the literature [Lesani et al. 2016; Lloyd et al. 2011]. Here, Bob sees Alice’s messages about her lost and found wallet in the order they were sent, and then responds with a friendly message of his own. Unfortunately, Carol sees Bob’s “Glad to hear it!” message only after having seen Alice’s initial “I lost my wallet...”, and not the follow-up message, so from Carol’s perspective, Bob is being rude. The problem here is not any lack of FIFO delivery; indeed, Alice’s messages are both seen by Bob and Carol in the order they were sent. Rather, the issue is that Bob’s “Glad to hear it!” response causally depends on Alice’s second message of “Found it!”, yet Carol sees “Glad to hear it!” before seeing the messages on which it depends. What is needed is a mechanism that will ensure that, for every message that is applied at a process, all of the messages on which it causally depends — comprising its causal history — are applied at that process first, regardless of who sent them.

Fig. 1. Two examples of executions that violate causal delivery. The vertical direction represents time, where later is lower; the horizontal direction represents space. Solid arrows represent messages between processes. On the left, Carol sees Alice’s messages in the opposite order of how they were sent. On the right, Carol sees Bob’s message before sees Alice’s second message. The dashed arrows in both executions depict how a causal delivery mechanism (Section 5) might delay the received messages in a buffer and deliver them later on, once doing so would not violate causal ordering.

TCP’s FIFO ordering guarantee applies so long as the messages in question are sent in the same TCP session. For cross-session guarantees, additional mechanisms are necessary.
In a setting like Alice, Bob, and Carol’s group text — where all messages are broadcast messages, that is, sent to all participants — a causal broadcast protocol is what is called for. The key idea of the protocol is to buffer messages at the receiving end until all causally preceding broadcast messages have been applied. The dashed arrows in Figure 1 represent the behavior of such a buffering mechanism. A typical implementation strategy is to have the sender of a message augment the message with metadata (for instance, a vector clock; see Section 3.1) that summarizes that message’s causal history in a way that can be efficiently checked on the receiver’s end to determine whether the message needs to be buffered or can be applied immediately to the receiver’s state. Although such mechanisms are well-known in the distributed systems literature [Birman and Joseph 1987a; Birman et al. 1991; Birman and Joseph 1987b], their implementation is “generally very delicate and error prone” [Bouajjani et al. 2017], motivating the need for machine-verified implementations of causal delivery mechanisms that are usable in real, running code.

The main contribution of this paper is a verified causal broadcast protocol implementation in Liquid Haskell. Liquid Haskell is an extension to the Haskell programming language that adds support for refinement types [Rushby et al. 1998; Xi and Pfenning 1998], which let programmers specify logical predicates that restrict, or refine, the set of values described by a type. Beyond giving more precise types to individual functions, Liquid Haskell’s reflection [Vazou et al. 2018, 2017] facility lets programmers use refinement types to specify “extrinsic” properties (see Section 4.1) that can relate multiple functions, and then prove those properties by writing Haskell programs to inhabit the specified types. We use Liquid Haskell to prove that in our causal broadcast implementation, messages cannot be delivered to a process in an order that violates causality, ruling out violations of causal delivery like those in Figure 1. Our implementation and proof development are, to our knowledge, the first machine-checked proof of correctness of an executable causal broadcast implementation, as well as the first distributed messaging protocol implementation of any kind to be verified with Liquid Haskell.2

Our verified causal broadcast implementation is useful in a variety of settings, including key-value stores, CRDTs, distributed snapshot algorithms, and peer-to-peer applications, and can be extended into a totally-ordered broadcast protocol that also preserves the causal order of messages [Birman et al. 1991].3 While previous work has mechanically verified the correctness of particular applications of causal ordering in distributed systems (such as causally consistent distributed key-value stores [Lesani et al. 2016]), our approach of factoring the causal broadcast protocol out into its own standalone, verified component means that it can be reused in each of these contexts. There is a need for such a standalone component: for instance, recent work on mechanized verification of CRDT convergence [Gomes et al. 2017] assumes the existence of a correct causal broadcast mechanism for its convergence result to hold. Our separately-verified causal broadcast implementation could be plugged together with CRDT implementations such as Gomes et al.’s to get an end-to-end correctness guarantee. Therefore our approach enables modular verification of higher-level consistency or convergence properties for applications built on top of the causal broadcast layer.

A great advantage of Liquid Haskell as a verification platform is that it results in immediately executable Haskell code, with no extraction step necessary, as with proof assistants such as Coq [Bertot and Castran 2010] or Isabelle [Wenzel et al. 2008] — making it easy to integrate our verified causal broadcast implementation with existing Haskell code. To demonstrate the usability

2While previous work [Liu et al. 2020] used Liquid Haskell to verify the convergence of various replicated data structures (see Section 7), our work verifies a property of the underlying messaging layer, while remaining completely agnostic to the content of the messages.

3Totally-ordered delivery does not imply causal delivery in general, although Birman et al. [1991]’s extension of causal broadcast to atomic broadcast provides both.
of our verified code, we put it to work as the foundation of a causally-consistent key-value store application (Section 6).

Figure 2 shows the module dependencies (and conceptual dependencies) between the parts of our proof development and our implementation, with references to the sections of the paper relevant to each. After formally defining causal delivery and the causal safety correctness property that we want our implementation to satisfy in Section 2, we describe the causal broadcast protocol we implemented (which is due to Birman et al. [1991], and not a new contribution of our work) in Section 3. Section 4 introduces Liquid Haskell and explains how we use it to express and prove the causal safety of our implementation, including a walk-through of the entire machine-checked proof. The VectorClock and Message modules are part of both the running implementation and our proof of causal safety, connecting verification and implementation. Section 5 then gives an overview of the remaining pieces of our Haskell implementation (DelayQueue, Process, and Protocol), including the external API presented to clients of the protocol, and Section 6 describes the key-value store (KVS) that we implemented on top of our protocol as a case study. Our proof development and implementation are available at [submitted as anonymized supplementary material].

2 BACKGROUND AND SYSTEM MODEL

We model a distributed system as a finite set of \( N \) processes (or nodes) \( p_i, i : 1..N \). Processes communicate with other processes by sending and receiving messages. In our setting, all messages are broadcast messages, meaning that they are sent to all processes in the system, including the sender itself.\(^4\)

\(^4\)For simplicity, we omit the messages that processes send to themselves from our example executions in Figures 1, 3, and 4. We assume that these self-sent messages are sent and delivered in one atomic step on the sender’s process.
We distinguish between message receipt and message delivery: nodes can receive messages at any time and in any order, and they may further choose to deliver a received message, causing that message to take effect at the node receiving it. Importantly, although nodes cannot control the order in which they receive messages, they can control the order in which they deliver those messages. Imagine a “mail clerk” on each node that intercepts incoming messages and chooses whether, and when, to deliver each one (by handing it off to the above application layer and recording that it has been delivered). Our task will be to ensure that the mail clerk delivers the messages in an order consistent with causality, regardless of the order in which they were received — implementing the behavior illustrated by the dashed arrows in Figure 1.

Each process consists of a totally ordered sequence of events. (Processes are single-threaded; a multi-threaded process could be modeled with multiple processes.) We denote the total order of events on process $p$ (the process order) with $\rightarrow_p$. For our discussion of causal delivery, we need to consider two kinds of events: broadcast events and deliver events. We will use $\text{broadcast}(m)$ to denote the event that sends a message $m$ to all processes, and $\text{deliver}_p(m)$ to denote the event that delivers $m$ on process $p$. Although a broadcast message has $N$ recipients (and therefore may be implemented as $N$ individual unicast messages under the hood), we nevertheless treat the sending of the message as a single event on the sender’s process. It is necessary to distinguish the process on which a deliver event takes place because each delivery of a broadcast is a distinct event.

Our network model is asynchronous, meaning that sent messages can take arbitrarily long to be received. Furthermore, for our safety result we need not assume that sent messages are eventually received (although such an assumption would be necessary for liveness; see Section 5 for a discussion).

Lamport’s happens-before relation [Lamport 1978] establishes an irreflexive partial order on the events in an execution of a distributed system:

**Definition 1 (Happens-Before ($\rightarrow$) [LAMPORT 1978]).** Given events $e$ and $e'$, we say that $e$ happens before $e'$, written $e \rightarrow e'$, if:

- $e$ and $e'$ occur on the same process $p$ with $e \rightarrow_p e'$; or
- $e$ is a message broadcast event and $e'$ is its corresponding deliver event, that is, $e = \text{broadcast}(m)$ and $e' = \text{deliver}_p(m)$ for a given message $m$ and some process $p$; or
- $e \rightarrow e''$ and $e'' \rightarrow e'$ for some event $e''$.

The first part of Definition 1 says that events happening on the same process are ordered by the happens-before relation. (In fact, such events are totally ordered by the happens-before relation.) For example, in Figure 1, Alice’s broadcast of “I lost my wallet...” happens before her broadcast of “Found it!”. The second part of the definition says that the broadcast of a given message happens before any delivery of that message, and the third part of the definition makes the relation transitive.

The $\rightarrow$ relation defines a partial order on events, but based on it, we can define a partial order on messages: we say that $m \rightarrow m'$ if $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$. We use the notation $\rightarrow$ for both relations, relying on context to disambiguate the meaning.

An execution of a distributed system consists of the set of all events on all processes, together with the process order relation $\rightarrow_p$ over events in each process $p$ and the happens-before relation $\rightarrow$ over all events. The happens-before relation captures the potential causality of events in an execution: for any two events $e$ and $e'$, if $e \rightarrow e'$, then $e$ may have caused $e'$, but we can be certain that $e'$ did not cause $e$. To avoid anomalous executions like those in Figure 1, we want to ensure that processes deliver messages in an order consistent with the happens-before partial order. This property is known as causal delivery.
Definition 2 (Causal delivery [Birman et al. 1991]). An execution observes causal delivery if, for all messages \( m \) and \( m' \), for all processes \( p \) delivering both \( m \) and \( m' \),
\[
m \rightarrow m' \implies \text{deliver}_p(m) \rightarrow \text{deliver}_p(m').
\]

The causal delivery property says that if message \( m \) causally precedes message \( m' \) in an execution, then any process delivering both \( m \) and \( m' \) should deliver \( m \) first. For example, in Figure 1 (left), the “I lost my wallet...” message causally precedes the “Found it!” message, because Alice broadcasts both messages with “I lost my wallet...” first, and so Bob and Carol would each need to deliver “I lost my wallet...” first for the execution to observe causal delivery. Furthermore, there is no requirement in the definition of causal delivery that \( m \) and \( m' \) must be broadcast by the same process. For example, in Figure 1 (right), Alice’s “Found it!” message causally precedes Bob’s “Glad to hear it!” message, and therefore Carol, who delivers both messages, must deliver Alice’s message first for the execution to observe causal delivery.

In the next section, we will discuss the algorithm run by our notional “mail clerk” that decides when to deliver received messages. For now, consider a predicate \( d \) that takes as arguments a message \( m \) and a process \( p \), returning true if \( m \) can be delivered at \( p \) without causing a causality violation, and false otherwise. (If \( m \) has already been delivered at \( p \), it should no longer be deliverable, and \( d(m, p) \) should return false.)

We are concerned with proving the safety side of causal delivery: if a process \( p \) receives two messages \( m \) and \( m' \), and \( m \) causally precedes \( m' \), then \( p \) should not deliver \( m' \) first. We can state this safety property as follows:

Definition 3 (Causal safety). A predicate \( d : (\text{Message} \times \text{Process}) \rightarrow \text{Bool} \) is causally safe if, for all messages \( m, m' \) and all processes \( p \),
\[
d(m, p) \land m \rightarrow m' \implies \neg d(m', p).
\]

Unlike Definition 2 above, which defines a property of executions (which can be thought of as traces generated by running a program and inspected after the fact), the causal safety property of Definition 3 has to do with whether the \( d \) predicate holds or does not hold for a given message at run time. In the following sections, we will show how to implement a causally safe predicate, and how we used Liquid Haskell to specify the causal safety property and prove that our Haskell implementation satisfies it.

3 Causal Broadcast Protocol

The causal broadcast protocol that we implemented is due to Birman et al. [1991]. The protocol is based on a type of logical clock well-known in the distributed systems literature, called a vector clock [Fidge 1988; Mattern 1989; Schmuck 1988]. Like other logical clocks, vector clocks do not track physical time (which would be problematic in distributed computations that lack a global physical clock), but instead track only the order of events in an execution.

Vector clocks provide a lightweight way for each process to keep track of how many messages it has seen and from whom they were sent, and for senders to transmit information about the causal dependencies of each message along with the message. The key idea of the protocol is that, when a process receives a message, it can compare that message’s accompanying vector clock to its own vector clock and determine whether the message can be delivered immediately (if all its causal dependencies have already been delivered) or if it needs to be buffered. In Section 3.1, we review vector clocks and how they precisely represent the happens-before relation \( \rightarrow \); readers already
familiar with vector clocks may skip ahead. In Section 3.2 we explain how the causal broadcast protocol works, and in particular, how it lets us implement a causally safe predicate.

### 3.1 Vector Clock Protocol

In this section we review the vector clock protocol as it is used in Birman et al.’s causal broadcast protocol. Vector clocks have numerous applications in distributed computing beyond the implementation of causal broadcast, and were independently invented by several authors [Fidge 1988; Mattern 1989; Schmuck 1988]; for readers interested in more detail, Schwarz and Mattern [1994] survey the main results.

A vector clock for a process $p$, which we denote $VC(p)$, is a vector of length $N$ (the number of processes in the system), indexed by process identifiers $i : 1..N$, where each entry is a natural number. At the beginning of execution, every process’s vector clock is initialized to zeroes; for instance, in a system with three processes, each process’s vector clock is initialized to $[0, 0, 0]$.

We will use vector clocks to track which broadcast messages each process “knows about”.$^5$ The vector clock protocol is as follows:

- When a process $p_i$ broadcasts a message $m$, $p_i$ increments its own position in its vector clock, $VC(p_i)[i]$, by 1.
- Each message broadcast by a process $p$ carries as metadata the value of $p$’s vector clock $VC(p)$ that was current at the time the message was broadcast (including the increment of the sender’s position). We denote the vector clock carried by a message $m$ with $VC(m)$.
- When a process $p$ delivers a message $m$, $p$ updates its own vector clock $VC(p)$ to the pointwise maximum of $VC(m)$ and $VC(p)$ by taking the maximum of the two integers at each index: for $k: 1..n$, we update $VC(p)[k]$ to $\max(VC(m)[k], VC(p)[k])$.

Figure 3 illustrates an example execution of three processes running the vector clock protocol. Initially, every process has a vector clock of $[0, 0, 0]$, with entries corresponding to processes $p_1$, $p_2$, and $p_3$ respectively. As each process broadcasts and delivers messages, it updates its vector clock according to the protocol. For example, when process $p_1$ broadcasts $m_1$, it increments its own position in its clock immediately before broadcasting the message, and $m_1$ carries the incremented clock $[1,0,0]$ as metadata. When $p_2$ delivers $m_1$, it updates its own vector clock to the pointwise maximum of its current value (still $[0,0,0]$ at this point) and $m_1$’s clock, resulting in $[1,0,0]$. $p_2$ then broadcasts $m_2$ with a vector clock of $[1,1,0]$, which $p_1$ delivers, updating its clock to the pointwise maximum of its current value ($[1,0,0]$ at this point) and $m_2$’s clock of $[1,1,0]$. On the other hand, when $p_3$ delivers $m_1$, there is no need for it to update its clock because its current clock value of $[1,1,0]$ is already greater than or equal to $m_1$’s clock at every position.

---

$^5$Some applications require vector clocks to track both send and receive events, or events internal to a process; for our purposes, we need to track broadcast events only.
We can define an ordering on vector clocks (of the same length) as follows: for two vector clocks \( VC_1 \) and \( VC_2 \) indexed by \( i : 1..n \),

- \( VC_1 \leq VC_2 \) if \( \forall i. VC_1[i] \leq VC_2[i] \), and
- \( VC_1 < VC_2 \) if \( VC_1 \leq VC_2 \) and \( \exists i. VC_1[i] < VC_2[i] \).

This ordering is not total: for example, in Figure 3, \( m_1 \) carries a vector clock of \([1, 0, 0]\) while \( m_3 \) carries a vector clock of \([0, 0, 1]\), and neither is less than the other. Correspondingly, \( m_1 \) and \( m_3 \) are causally independent (or concurrent): neither message has a causal dependency on the other. \( m_2 \), on the other hand, causally depends on \( m_1 \), and carries a vector clock of \([1, 1, 0]\), which is less than \( m_1 \)'s vector clock of \([1, 0, 0]\) according to our ordering. In fact, vector clocks precisely characterize the causal partial ordering [Fidge 1988; Mattern 1989]: for all messages \( m, m' \),

\[
m \rightarrow m' \iff VC(m) < VC(m').
\]

This powerful two-way implication means that we can boil down the problem of reasoning about causal relationships between messages to the problem of comparing fixed-length vectors of integers — a task that Liquid Haskell and its underlying SMT solver happen to be well suited for.

### 3.2 Deliverability

By itself, the vector clock protocol from the previous section does not enforce causal delivery of messages. Indeed, the execution in Figure 3 violates causal delivery: under causal delivery, process \( p_3 \) would not deliver \( m_1 \) before \( m_2 \). However, the vector clock metadata attached to each message can be used to enforce causal delivery of broadcast messages. This is possible because the vector clock attached to a message can be thought of as a summary of the causal history of that message: for example, in Figure 3, \( m_2 \)'s vector clock of \([1, 1, 0]\) expresses that one message from \( p_1 \) (represented by the \( 1 \) in the first entry of the vector) causally precedes \( m_2 \). Furthermore, each process’s vector clock tracks how many messages it has delivered from each process in the system (including itself, since messages are sent and delivered in one atomic step).

We can therefore ensure that an execution that is running the vector clock protocol observes causal broadcast by having the recipient of each broadcast message compare the message’s attached vector clock with its own vector clock, as follows:

**Definition 4 (Deliverability [Birman et al. 1991]).** A message \( m \) broadcast by a process \( p_i \) is deliverable at a process \( p_j \neq p_i \) if, for \( k : 1..n \),

\[
\begin{align*}
VC(m)[k] &= VC(p_j)[k] + 1 & \text{if } k = i, \text{ and} \\
VC(m)[k] &\leq VC(p_j)[k] & \text{otherwise.}
\end{align*}
\]

The first part of Definition 4 says that \( m \) is the recipient \( p_j \)'s next expected message from the sender, \( p_i \). The number of messages from \( p_i \) that \( p_j \) has already delivered will appear in \( VC(p_j) \) at index \( i \), so \( VC(m)[i] \) should be exactly one greater than \( VC(p_j)[i] \). If \( VC(m)[i] \) is more than one greater than \( VC(p_j)[i] \), it means that there is at least one other message \( m' \) from \( p_i \) that causally precedes \( m \) and that \( p_j \) has not yet delivered, and so \( p_j \) should not deliver \( m \) while \( m' \) remains undelivered. (The case where \( VC(m)[i] \leq VC(p_j)[i] \) cannot occur, because \( p_i \) is always at least as up to date on its own sent messages as \( p_j \).)

The second part of Definition 4 says that \( m' \)'s causal history does not include any messages sent by processes other than \( p_i \) that \( p_j \) has not yet delivered. If \( m' \)'s vector clock is greater than \( p_j \)'s vector clock in any position \( k \neq i \), then it means that, before sending \( m \), process \( p_i \) must have delivered some message \( m' \) from \( p_k \) that has not yet been delivered at \( p_j \). By Definition 1, we have \( broadcast(m') \rightarrow deliver_{p_i}(m') \), and because \( m' \) was delivered at \( p_i \) before \( m \) was broadcast
by \( p_i \), we have \( \text{deliver}_{p_i}(m') \rightarrow \text{broadcast}(m) \), so by transitivity of \( \rightarrow \) we have \( \text{broadcast}(m') \rightarrow \text{broadcast}(m) \). Therefore \( m' \) causally precedes \( m \), and so \( p_j \) should not deliver \( m \) while \( m' \) remains undelivered.

Combining the vector clock protocol from the previous section with the deliverability property of Definition 4 gives us Birman et al.’s causal broadcast protocol. The key idea is to buffer each message \( m \) received at a process \( p \) until \( m \) is deliverable at \( p \) according to Definition 4. Each process stores messages that need to be buffered in a queue, the delay queue (see Section 5). Whenever a process delivers a message and updates its own vector clock, it can check its delay queue for buffered messages and deliver any messages that have become deliverable (which may in turn make other buffered messages deliverable).

To illustrate how the protocol works, Figure 4 shows the two problematic executions we saw previously in Figure 1, but now with the causal broadcast protocol in place to prevent messages being delivered out of causal order. Each process keeps a vector clock with three entries corresponding to Alice, Bob, and Carol respectively. Suppose that \( m_{\text{lost}} \) is Alice’s “I lost my wallet...” message, \( m_{\text{found}} \) is Alice’s “Found it!” message, and \( m_{\text{glad}} \) is Bob’s “Glad to hear it!” message.

In Figure 4 (left), Bob receives Alice’s messages in the order she broadcasted them, and so he can deliver them immediately. For example, when Bob receives \( m_{\text{lost}} \), his own vector clock is \([0, 0, 0]\), and the vector clock on the message is \([1, 0, 0]\). The message is deliverable at Bob’s process because it is one greater than Bob’s own vector clock in the sender’s (Alice’s) position, and less than or equal to Bob’s vector clock in the other positions, so Bob delivers it immediately after receiving it. Carol, on the other hand, receives \( m_{\text{found}} \) first. This message has a vector clock of \([2, 0, 0]\), so it is not immediately deliverable at Carol’s process because Carol’s vector clock is \([0, 0, 0]\), and so the entry of 2 at the sender’s index is too large, indicating that the message is “from the future” and needs to be buffered in Carol’s delay queue for later delivery. \( m_{\text{found}} \) can be delivered later on, once Carol receives and delivers \( m_{\text{lost}} \).
In Figure 4 (right), Bob delivers two messages from Alice and then broadcasts \( m_{glad} \). \( m_{glad} \) has a vector clock of \([2,1,0]\), indicating that it has two messages sent by Alice in its causal history. When Carol receives \( m_{glad} \), her own vector clock is only \([1,0,0]\), indicating that she has only delivered one of those messages from Alice so far, so Carol must buffer \( m_{glad} \) in her delay queue until she receives and delivers \( m_{found} \), the missing message from Alice, increasing her own vector clock to \([2,0,0]\). Now \( m_{glad} \) is deliverable at Carol’s process, and Carol can deliver it, increasing her own vector clock to \([2,1,0]\).

4 EXPRESSING AND PROVING CAUSAL SAFETY IN LIQUID HASKELL

While the examples in the previous section may help build intuition for why Definition 4’s deliverability condition is the right one to ensure causal delivery, we still need to prove that our running implementation of Section 3’s protocol is correct. In this section, after giving a brief introduction to refinement types and Liquid Haskell and introducing some of the key data types of our implementation, we show how we express the causal safety property of Definition 3 as a refinement type in Liquid Haskell, and how we proved that property using Liquid Haskell’s theorem-proving capabilities.

4.1 Refinement Types and Liquid Haskell

Refinement types [Rushby et al. 1998; Xi and Pfenning 1998] let programmers specify types augmented with logical predicates, called refinement predicates, that restrict the set of values that can inhabit the type. Depending on the expressivity of the predicate language, programmers can specify rich properties using refinement types, sometimes at the expense of decidability of type checking. Liquid Haskell avoids that problem by restricting refinement predicates to an SMT-decidable logic [Rondon et al. 2008; Vazou et al. 2014]. For example, in Liquid Haskell we could define the type of even integers by refining the Haskell type \( \text{Int} \) using the refinement type \( \{ v : \text{Int} \mid v \mod 2 == 0 \} \), where \( v \mod 2 == 0 \) is the refinement predicate and \( v : \text{Int} \) binds the name \( v \) for values of type \( \text{Int} \) that appear in the refinement predicate. One could define an analogous refinement type for odd integers, and then write a Liquid Haskell function for adding them:

\[
\begin{align*}
\text{type EvenInt} &= \{ v : \text{Int} \mid v \mod 2 == 0 \} \\
\text{type OddInt} &= \{ v : \text{Int} \mid v \mod 2 == 1 \} \\
\text{oddAdd} : : \text{OddInt} &\rightarrow \text{OddInt} \rightarrow \text{EvenInt} \\
\text{oddAdd} \ x \ y &= x + y
\end{align*}
\]

The type \( \text{OddInt} \) of the arguments to \( \text{oddAdd} \) expresses the precondition that \( x \) and \( y \) will be odd, and the return type \( \text{EvenInt} \) expresses the postcondition that \( x + y \) will evaluate to an even number. Liquid Haskell automatically proves that such postconditions hold by generating verification conditions that are checked at compile time by the underlying SMT solver, Z3 [de Moura and Bjørner 2008]. If the solver finds a verification condition to be invalid, typechecking fails. If the return type of \( \text{oddAdd} \) had been \( \text{OddInt} \), for instance, the above code would fail to typecheck.

Aside from preconditions and postconditions of individual functions, Liquid Haskell makes it possible to verify extrinsic properties that are not specific to any particular function’s definition. For example, the type of \( \text{sumOdd} \) below expresses the extrinsic property that the sum of an odd and an even number is an odd number:

\[
\begin{align*}
\text{sumOdd} : : x : \text{OddInt} &\rightarrow y : \text{EvenInt} \rightarrow \{ (x + y) \mod 2 == 1 \} \\
\text{sumOdd} ~ ~ = ~ ~ ()
\end{align*}
\]
Here, \texttt{sumOdd} is a Haskell function that returns a \textit{proof} that the sum of \texttt{x} and \texttt{y} is odd. (The return type \{ (\texttt{x + y}) \mod 2 == 1 \} is short for \{ \texttt{v:Proof | (x + y) \mod 2 == 1} \}, where \texttt{Proof} is a type alias for Haskell’s \texttt{(())} (unit type). The \texttt{v:Proof} part can be omitted because there is no need to name the result.) Because the proof of this particular property is easy for the SMT solver to carry out automatically, the body of the \texttt{sumOdd} function need not say anything but \texttt{()}. In general, however, programmers can specify arbitrary extrinsic properties in refinement types, including properties that refer to arbitrary Haskell functions via the notion of \textit{reflection} [Vazou et al. 2017]. The programmer can then prove those extrinsic properties by writing Haskell programs that inhabit those refinement types — with the help of the underlying SMT solver to simplify the construction of these proofs-as-programs [Vazou et al. 2018, 2017].

Liquid Haskell thus occupies a unique position at the intersection of SMT-based program verifiers such as Dafny [Leino 2010], and theorem provers that leverage the Curry-Howard correspondence such as Coq [Bertot and Castran 2010] and Agda [Norell 2008]. A Liquid Haskell program can consist of both application code like \texttt{oddAdd} (which runs at execution time, as usual) and verification code like \texttt{sumOdd} (which only “runs” at compile time), but, pleasantly, both are just Haskell programs, albeit annotated with refinement types. Being based on Haskell enables programmers to gradually port code from vanilla Haskell to Liquid Haskell, adding richer specifications to code as they go. Furthermore, verified Liquid Haskell libraries can be used directly in arbitrary Haskell programs, letting programmers take advantage of formally-verified components from unverified code written in an industrial-strength, general-purpose language. (We discuss our experiences — both positive and negative — with this “gradual verification” approach later on in Section 8.)

4.2 The PID, VC, and Message Types

Recall from Section 2 that we model a distributed system as a finite set of \(N\) processes, where \(N\) is an arbitrary natural number. We want our proof of causal safety to be agnostic to \(N\), yet at run time we need to know what \(N\) is specifically, because \(N\) determines the length of vector clocks (and hence what constitutes a valid index into a vector clock). We accomplish this in Liquid Haskell by defining in our \texttt{VectorClock} module an “uninterpreted” Liquid Haskell \textit{measure} [Vazou et al. 2014] called \texttt{procCount}, of \texttt{Nat} type, and a type \texttt{ProcCount} based on it, as follows:

\begin{verbatim}
  type Nat = { v : Int | 0 <= v }
  measure procCount :: Nat
  type ProcCount = { s : Nat | s == procCount }
\end{verbatim}

We define a process identifier (and index into vector clocks) type, \texttt{PID} as

\begin{verbatim}
  type PID = Fin procCount
\end{verbatim}

where \texttt{Fin V} is the type of natural numbers less than \texttt{V}, since vector clocks are 0-indexed in our implementation:

\begin{verbatim}
  type Fin V = { k : Nat | k < V }
\end{verbatim}

A vector clock index of \texttt{PID} type is therefore guaranteed to never index out of bounds.

We also define a vector clock type, \texttt{VC}, that makes use of \texttt{procCount}. We first define the type \texttt{Vec} of length-indexed vectors:

\begin{verbatim}
  type Vec a V = { v : [a] | len v == V }
\end{verbatim}

and then define \texttt{VC} using \texttt{Vec} as follows:

\begin{verbatim}
  data VC = VC (Vec Nat procCount)
\end{verbatim}

Whenever we launch a new process (see Section 6), we initialize that process’s vector clock by calling a function \texttt{vcNew} of type \texttt{ProcCount -> VC} that creates a new \texttt{VC} of the appropriate length
and initializes it with zeroes. The actual number passed to \texttt{vcNew} is specified at run time, via user input.

Finally, the type we use for messages throughout our implementation is a record with three fields:

\begin{verbatim}
data Message r = Message { mSender::PID, mSent::VC, mRaw::r }
\end{verbatim}

\texttt{mSender} is the process identifier of the process that sent the message, and \texttt{mSent} is the vector clock attached to the message. The \texttt{r} type parameter is the type of the actual content of the message (which might be, say, a string of serialized JSON), necessary for the implementation, but irrelevant for verification purposes.

### 4.3 Causal Safety as a Refinement Type

In our implementation, the \texttt{Message} module provides a predicate \texttt{deliverable} of type \texttt{Message r \rightarrow VC \rightarrow Bool} that determines the deliverability of a message at a process. The \texttt{VC} argument to \texttt{deliverable} represents the state of the process receiving the message. Although our implementation has a record type for processes, which has a \texttt{VC} field along with other process state (as we will see in Section 5), the vector clock is the only part of the process state that \texttt{deliverable} needs, so a \texttt{VC} is the only argument that needs to be passed.

We want to prove that our \texttt{deliverable} predicate has the causal safety property of Definition 3, where the \texttt{VC} argument to \texttt{deliverable} corresponds to the process \( p \) in Definition 3.

\begin{theorem}
\texttt{deliverable} is causally safe.
\end{theorem}

We can express the claim made by Theorem 1 as a refinement type in Liquid Haskell as follows:

\begin{verbatim}
safety :: procVc : VC
  -> m1 : Message r
  -> m2 : Message r
  -> DeliverableProp m1 procVc
  -> CausallyBeforeProp m1 m2
  -> Not (DeliverableProp m2 procVc)
\end{verbatim}

The interesting parts of \texttt{safety}'s type are the \texttt{DeliverableProp} and \texttt{CausallyBeforeProp} types. We first summarize their meaning briefly before explaining them in more detail below:

- \texttt{DeliverableProp \( m \) procVc} is the type of a \textit{proof} that message \( m \) is deliverable at a process with the vector clock \texttt{procVC}.
- \texttt{CausallyBeforeProp \( m1 \) \( m2 \)} is the type of a \textit{proof} that message \( m1 \) causally precedes \( m2 \).

#### 4.3.1 DeliverableProp

We now go into more detail about \texttt{DeliverableProp}, which is a Liquid Haskell type synonym parameterized with expression variables\(^6\) \texttt{M} and \texttt{P}, defined as follows:

\begin{verbatim}
  type DeliverableProp M P = k:PID \rightarrow { _::Proof | deliverableK M P k }
\end{verbatim}

Therefore \texttt{DeliverableProp M P} is really the type of a \textit{function} that takes a \texttt{PID}, \( k \), and returns a proof that the predicate \texttt{deliverableK M P k} evaluates to \texttt{True}. As mentioned before, a \texttt{PID} is a process identifier, but more importantly for our purposes, it is also an index into a vector clock. \texttt{deliverableK} is an ordinary, executable Haskell function and a part of our running implementation, provided by the \texttt{Message} module shown in Figure 2. It appears in the \texttt{DeliverableProp} type by means of Liquid Haskell’s \textit{reflection} mechanism [Vazou et al. 2017] that lets arbitrary Haskell

\(^6\)In Liquid Haskell, type synonyms can be parameterized either with ordinary Haskell type variables or with Liquid Haskell expression variables; in the latter case, the parameter starts with a capital letter.
function calls appear in refinement types. The \texttt{deliverableK} function implements the deliverability condition we saw in Definition 4, but for a particular index \( k \):

\begin{verbatim}
deliverableK :: Message r -> VC -> PID -> Bool

deliverableK m procVc k
  | k == mSender m = mSent m ! k == (procVc ! k) + 1
  | otherwise = mSent m ! k <= procVc ! k
\end{verbatim}

\texttt{deliverableK} takes as arguments a message \( m \), a vector clock \texttt{procVc}, and a vector clock index \( k \), and returns \texttt{True} or \texttt{False}. The \texttt{procVc} argument is the vector clock of the receiving process. \texttt{mSender m} is the \texttt{PID} of the sender of \( m \), and \texttt{mSent m} is the vector clock attached to \( m \). \texttt{deliverableK} implements the behavior specified in Definition 4: in the case where \( k == mSender m \) — that is, the case where \( k \) is the \texttt{PID} of the process that sent \( m \) — \texttt{deliverableK} checks that \( m \)'s vector clock is one greater than the receiving process's vector clock at index \( k \), and otherwise, \texttt{deliverableK} checks that \( m \)'s vector clock is less than or equal to the process's vector clock at index \( k \).

In our running implementation, the \texttt{deliverable} predicate calls \texttt{deliverableK} for each valid index \( k \) and returns \texttt{True} if all the calls to \texttt{deliverableK} return \texttt{True}. On the verification side, the equivalent of iteration over values of \( k \) is expressed as universal quantification: the type \( k:\texttt{PID} -> \{
\_:\texttt{Proof} | \texttt{deliverableK M P k} \}\) says that, for all \( k \), a call to \texttt{deliverableK M P k} will return \texttt{True}. Therefore, if we have a value \( v \) of type \texttt{DeliverableProp M P}, we can find \texttt{deliverableK M P k} for any \( k \) by applying \( v \) to \( k \).

The \texttt{Not} in the return type of \texttt{safety} is the Liquid Haskell type synonym

\begin{verbatim}
type Not t = t -> \{ \_:Proof | False \}
\end{verbatim}

So, \texttt{safety}’s return type \( \{ \texttt{Not (DeliverableProp M2 procVc)} \} \) is the type of a function that, when provided an argument of type \texttt{DeliverableProp M2 procVc} — that is, when provided a proof that message \( m2 \) is deliverable at a process with the vector clock \texttt{procVc} — will return a proof of \texttt{False}.

4.3.2 \textbf{CausallyBeforeProp}. A similar quantification takes place in \texttt{CausallyBeforeProp}, which is also a standard Liquid Haskell type synonym:

\begin{verbatim}
type CausallyBeforeProp M1 M2 = k:\texttt{PID} -> \{ \_:Proof | causallyBeforeK M1 M2 k \}
\end{verbatim}

\texttt{CausallyBeforeProp M1 M2} is the type of a \textit{function} that takes as its argument a \texttt{PID} \( k \) and returns a proof that the predicate \texttt{causallyBeforeK M1 M2 k} evaluates to \texttt{True}. Just as with \texttt{DeliverableProp}, the \( k \) argument is an index into a vector clock, and like \texttt{deliverableK}, \texttt{causallyBeforeK} is an ordinary, executable Haskell function that is reflected into the refinement logic. \texttt{causallyBeforeK} takes as arguments two messages \( m1 \) and \( m2 \) and an index \( k \), and checks that \( m1 \) causally precedes \( m2 \) at the index \( k \) by leveraging the relationship between causal ordering and vector clocks described earlier in Section 3.1. In particular, \texttt{causallyBeforeK m1 m2 k} should return \texttt{True} if \( VC(m1)[k] \leq VC(m1)[k] \) and \( VC(m1) \neq VC(m2) \), and \texttt{False} otherwise.

\texttt{causallyBeforeK} can therefore be implemented in terms of helper functions \texttt{vcLessK} and \texttt{vcLessEqualK}, provided by the \texttt{VectorClock} module shown in Figure 2. The implementation of all three functions is as follows:

\begin{verbatim}
causallyBeforeK :: Message r -> Message r -> PID -> Bool
causallyBeforeK m1 m2 k = vcLessK (mSent m1) (mSent m2) k

vcLessK :: VC -> VC -> PID -> Bool
vcLessK a b k = vcLessEqualK a b && a /= b
\end{verbatim}
vcLessEqualK :: VC -> VC -> PID -> Bool
vcLessEqualK a b k = a ! k <= b ! k

Therefore CausallyBeforeProp M1 M2 amounts to a proposition that message M1 causally precedes message M2, because it is the type of a function that takes any PID k and returns a proof that causallyBeforeK M1 M2 k returns True.

4.4 Proving Causal Safety in Liquid Haskell

Now that we have seen how our specification of the causal safety property connects to our executable implementation, in this section we show how we actually carry out the proof of Theorem 1 by inhabiting the type of safety with a program. Listing 1 shows our causal safety property and its complete proof in Liquid Haskell. Liquid Haskell checks this proof in under half a second. We have already discussed the property being proved, expressed by the type of safety; now we dive into the proof itself.

Our causal safety proof depends on one assumed axiom, processOrderAxiom, shown on lines 1-5 of Listing 1. (The assume keyword in Liquid Haskell lets one state and use a property without providing a proof.) processOrderAxiom says that every message sent by a given process has a distinct value at the sender’s index of its vector clock. In particular, given messages m1 and m2 that were sent by the same process pk (expressed by the precondition in line 3 of Listing 1), processOrderAxiom returns a proof that VC(m1)[k] ≠ VC(m2)[k]. This follows from the fact that events on a single process have a total order.7

Lines 8-9 of Listing 1 are the type of safety, and the body of the function begins on line 10. The return type of safety is Not (DeliverableProp m2 procVc), which, as we saw in the previous section, is a synonym for DeliverableProp m2 procVc -> { _:Proof | False }. So safety takes not five, but six arguments:

- the process vector clock _procVc (used only in the type, not the proof itself, so ignored by prepending an underscore);
- the messages m1 and m2;
- a premise that m1 is deliverable at a process with vector clock procVC, named m1_d_p;
- a premise that m1 causally precedes m2, named m1_before_m2;
- and a premise (from which we need to prove False) that m2 is deliverable at a process with vector clock procVC, named m2_d_p.

Let us consider how we can combine the claims represented by these premises to derive a contradiction and prove False, to establish that m2 is, in fact, not deliverable at a process with vector clock procVc. The proof has two cases: either m1 and m2 came from the same sender (lines 11-16 of Listing 1), or they came from different senders (lines 17-22). Each line of the proof consists of a use of one of the above premises (or of our assumed axiom), chained together by the ? operator that Liquid Haskell provides. x ? p returns x, but encodes the information contained in p to the SMT logic and gives it to the SMT solver to help with reasoning. The *** QED on lines 16 and 22 is not necessary to complete the proof, but included for aesthetic reasons.

We consider the same-sender case first:

- The argument m1_d_p is of type DeliverableProp m1 procVc, so it is a function that, when applied to a vector clock index k, returns a proof that deliverableK m1 procVc k is true. On line 13 of the proof we apply this function, m1_d_p, to a particular index, _mSender m1, to which we need to prove False that m2 is deliverable at a process with vector clock procVC, named m2_d_p.

7In fact, since in the causal safety proof m1 causally precedes m2 and they were both sent by p_k, it is the case that VC(m1)[k] < VC(m2)[k], but just the fact that they are distinct is all Liquid Haskell needs to know to carry out the proof.
assume processOrderAxiom

::: m1 : Message r -> m2 : Message r
-> { :::Proof | mSender m1 == mSender m2 }
-> { :::Proof | vcReadK (mSent m1) (mSender m1) != vcReadK (mSent m2) (mSender m2) }
processOrderAxiom _m1 _m2 _proof = ()

safety

::: procVc : VC -> m1 : Message r -> m2 : Message r
-> DeliverableProp m1 p -> CausallyBeforeProp m1 m2 -> Not (DeliverableProp m2 p)
safety _procVc m1 m2 m1_d_p m1_before_m2 m2_d_p
| mSender m1 == mSender m2
= ()
? m1_d_p (mSender m1)
? m2_d_p (mSender m2)
? processOrderAxiom m1 m2 ()
*** QED
| otherwise
= ()
? m1_before_m2 (mSender m1)
? m1_d_p (mSender m1)
? m2_d_p (mSender m1)
*** QED

Listing 1. Our causal safety property and its proof in Liquid Haskell.

we know \( m_1 \) is deliverable, then we know that the entry at that index must be exactly one greater than the sender’s entry in \( \text{procVc} \), so \( \text{VC}(m_1)[k_{\text{sender}}] = \text{procVc}[k_{\text{sender}}] + 1 \).

- Likewise, the argument \( m_2_d_p \) is of type \( \text{DeliverableProp} \) \( m_2 \) \( \text{procVc} \), so it is a function that, when applied to a \( k \), returns a proof that \( \text{deliverableK} \) \( m_2 \) \( \text{procVc} \) \( k \) is true. On line 14 of the proof we apply \( m_2_d_p \) to \( \text{mSender} \) \( m_2 \), which must be the same index \( k_{\text{sender}} \) we applied \( m_1_d_p \) to, since \( m_1 \) and \( m_2 \) came from the same sender. Since \( k_{\text{sender}} \) is the sender’s position in \( m_2 \)’s vector clock and \( m_2 \) is deliverable, we know that the entry there, too, must be exactly one greater than the sender’s entry in \( p \)’s vector clock, so \( \text{VC}(m_2)[k_{\text{sender}}] = \text{procVc}[k_{\text{sender}}] + 1 \).

- Therefore \( \text{VC}(m_1)[k_{\text{sender}}] = \text{VC}(m_2)[k_{\text{sender}}] \). However, on line 15 of the proof we apply \( \text{processOrderAxiom} \), which says that the sender’s vector clock entry in messages from the same sender must be distinct (due to events on a process being totally ordered). So we have established a contradiction, and Liquid Haskell is able to prove \( \text{False} \), as required.

Next, on lines 17-22 of Listing 1, we consider the case in which \( m_1 \) and \( m_2 \) came from different senders:

- The argument \( m_1-before_m2 \) is of type \( \text{CausallyBefore} \) \( m_1 \) \( m_2 \), so it is a function that, when applied to a vector clock index \( k \), returns a proof that \( \text{causallyBeforeK} \) \( m_1 \) \( m_2 \) \( k \) is true. On line 19 of the proof we apply this function, \( m_1-before_m2 \), to the index \( \text{mSender} \) \( m_1 \). Let us call this index \( k \). Because of the causal order of \( m_1 \) and \( m_2 \), we have that \( \text{VC}(m_1)[k] \leq \text{VC}(m_2)[k] \).

- Next, on line 20 of the proof we apply the function \( m_1_d_p \) to the same index, \( k \). Since \( m_1 \) is deliverable, we have that \( \text{VC}(m_1)[k] = \text{procVc}[k] + 1 \).

- On line 21 of the proof we apply the function \( m_2_d_p \) to the same index, \( k \). Since \( m_2 \) is not from the same sender as \( m_1 \), and \( m_2 \) is deliverable, we have that \( \text{VC}(m_2)[k] \leq \text{procVc}[k] \).
Putting these facts together, we have:

\[ VC(m_1)[k] \leq VC(m_2)[k] \leq procVc[k] < procVc[k] + 1 = VC(m_1)[k] \]

which is a contradiction because \( VC(m_1)[k] \) cannot be less than itself, so Liquid Haskell is able to prove \textbf{False} in this case as well, completing the proof.

Structuring our safety property in such a way that its arguments are \textit{functions} that take a particular index \( k \) into a vector clock helps us precisely express in our proof the reason why Birman et al.’s causal broadcast algorithm works. Since deliverability is decided on the basis of comparing vector clocks at each index, to prove that \( m_2 \) is not deliverable, there has to be at least one particular index for which the claim that \( m_2 \) is deliverable at that index would lead to a contradiction, and the proof tells us exactly which index it is: that of \( m_1 \)’s sender (which is also \( m_2 \)’s sender in the same-sender case)! This way of stating the safety property was inspired by Agda’s dependent function types (see Section 8 for further discussion on this point).

5 DELAY QUEUE, PROCESS STATE, AND EXTERNAL API

Our causal safety result establishes the correctness of a key piece of our causal broadcast implementation — in particular, the correctness of the notion of deliverability implemented by \texttt{deliverable}. In this section we give an overview of the remaining pieces of our implementation (provided by the \texttt{DelayQueue}, \texttt{Process}, and \texttt{Protocol} modules shown in Figure 2), including the delay queue where each process buffers messages that are not yet deliverable, additional state that processes keep, and the external API that applications may call to broadcast messages. We use refinement types to precisely specify the interfaces of many of these components.

Figure 5 shows an example architecture of an application using our causal broadcast implementation. A collection of (potentially geo-distributed) peer nodes, which we call the causal broadcast cluster, each run the causal broadcast protocol along with their application code (for instance, a key-value store or a group chat application). Clients of the application communicate their requests to the nodes; one or more clients may communicate with each node. The application instance on a node generates messages and broadcasts them to other nodes, and delivers messages received from other nodes.

The causal safety of our implementation means that messages are never delivered to the application layer in an order that violates causality. However, this safety property would also be true
of an implementation in which messages are never delivered at all! Therefore we use this section
to informally argue that our implementation is not only safe, but live, meaning that messages will
not languish forever in the delay queue. As mentioned in Section 2, for our safety result we need
not make any assumption of reliable message receipt, but liveness rests on the assumption of a
reliable message transport layer, that is, one in which sent messages are eventually received —
albeit in arbitrary order and with arbitrarily long latency.

5.1 Delay Queue

When a process receives a message that is not yet deliverable according to the deliverable
predicate, it must buffer that message in a process-local delay queue for later delivery once the causally
preceding messages have been delivered. Listing 2 shows the DelayQueue type and the definitions
of the dqEnqueue and dqDequeue operations it supports, provided by the DelayQueue module
shown in Figure 2. The DelayQueue record type, defined on lines 1-4 of Listing 2, has a dqSelf field for the PID of the process to which it belongs, and a dqList field for the buffered messages themselves. The refinement type specification for dqDequeue guarantees that the returned message is deliverable, and the refinement on DelayQueue prevents inserting messages sent from the current process into the delay queue.
sorted in ascending order by the vector clock of the message, with concurrent messages in ascending order of receipt, although our implementation does not rely on this behavior.\footnote{We considered using refinement types to express this sortedness invariant, but we had trouble doing so cleanly in Liquid Haskell, because \texttt{insertAfterBy} is recursive and the invariant needs to be preserved through a recursive call, resulting in undesirable code duplication.} The refinement type postcondition of \texttt{dqEnqueue} ensures that the queue grows by one when a message is enqueued, and that we do not enqueue self-sent messages.

The \texttt{dqDequeue} operation (lines 14-26 of Listing 2) takes the process vector clock \texttt{procVC} as an argument and tests each message in \texttt{dqList} in turn with the \texttt{deliverable} predicate that we describe at the end of Section 4.3.1. \texttt{dqDequeue} returns a pair of an updated delay queue and the first message \texttt{m} for which \texttt{deliverable m procVC} is satisfied, if there is one. If there are no deliverable messages in \texttt{dqList}, \texttt{dqDequeue} returns \texttt{Nothing}. The refinement type postcondition of \texttt{dqDequeue} ensures that the queue shrinks by one when a message is dequeued, and that the message we dequeued is indeed deliverable.

5.2 Process State

Our \texttt{Process} module exports a \texttt{Process} data type, defined as follows:

```
data Process r = Process
    { pID :: PID,
      pVC :: VC,
      pDQ :: DQ r pID,
      pToSelf :: FIFO {{ m : Message r | pID == mSender m }},
      pToNetwork :: FIFO {{ m : Message r | pID == mSender m }}
    }
```

In addition to the delay queue discussed in the previous section, a process must keep track of its own process ID, its own \texttt{VC}, and two additional process-local message buffers, \texttt{pToSelf} and \texttt{pToNetwork}. The \texttt{pToSelf} buffer contains self-sent messages, and the \texttt{pToNetwork} buffer contains outgoing messages.

We specify the type of a process’s delay queue with the type synonym from line 6 of Listing 2, \texttt{DQ r pID}, which references the record field value \texttt{pID} defined just above to express the invariant that messages a process sends to itself cannot be inserted into its own delay queue. Similarly, the refinement types of \texttt{pToSelf} and \texttt{pToNetwork} are constrained to have only messages sent by this process, to provide a place to temporarily store them before delivering to itself and broadcasting them to the network, respectively. The \texttt{FIFO} type (not shown) is implemented straightforwardly using a list and provides standard operations.

5.3 External API

Our causal broadcast implementation exposes a purely functional API that provides five functions: \texttt{pNew}, \texttt{send}, \texttt{receive}, \texttt{deliver}, and \texttt{drainBroadcasts}. Listing 3 shows the implementations of the external API functions. We summarize their behavior as follows:

- The \texttt{pNew} function (lines 1-8 of Listing 3) creates and returns a fresh \texttt{Process} with the specified \texttt{PID} and \texttt{ProcCount}, an empty delay queue, and empty \texttt{pToSelf} and \texttt{pToNetwork} buffers.
- The \texttt{send} function (lines 10-19) takes a raw message of type \texttt{r}. It implements the sending side of the vector clock protocol of Section 3.1 by incrementing its own position in the process vector clock by 1 and wrapping the raw message up with the updated vector clock and sender ID in an envelope \texttt{Message r}. It also pushes the wrapped message onto both the \texttt{pToSelf} and \texttt{pToNetwork} buffers.
verified causal broadcast with liquid haskell

\[
p_{\text{New}} :: \text{PID} \rightarrow \text{ProcCount} \rightarrow \text{Process}\ r
\]

\[
p_{\text{New}}\ \text{pid}\ p_{\text{Count}} = \text{Process}
\]

\[
\begin{align*}
&\text{\{ pID = pid} \\
&\text{, pVC = vcNew pCount} \\
&\text{, pDQ = dqNew pid} \\
&\text{, pToSelf = fNew} \\
&\text{, pToNetwork = fNew}\}
\end{align*}
\]

\[
\)\]

\[
\text{send} :: r \rightarrow \text{Process}\ r \rightarrow \text{Process}\ r
\]

\[
\begin{align*}
&\text{send}\ r\ p \\
&\quad = \text{let} \\
&\quad \quad \text{vc} = \text{vcTick (pID p) (pVC p)} \\
&\quad \quad \text{m} = \text{Message(mSender=pID p, mSent=vc, mRaw=r)} \\
&\quad \quad \text{in} \\
&\quad \quad \text{p}\ \\
&\quad \quad \text{, pToNetwork = fPush (pToNetwork p) m} \\
&\quad \quad \text{, pToSelf = fPush (pToSelf p) m}
\end{align*}
\]

\[
\)\]

\[
\text{receive} :: \text{Message}\ r \rightarrow \text{Process}\ r \rightarrow \text{Process}\ r
\]

\[
\begin{align*}
&\text{receive}\ m\ p \\
&\quad | \text{mSender m == pID p = p -- Messages sent by this process are ignored.} \\
&\quad | \text{otherwise = p(pDQ=dqEnqueue m (pDQ p))}
\end{align*}
\]

\[
\)\]

\[
\text{deliver} :: \text{Process}\ r \rightarrow (\text{Process}\ r, \text{Maybe}\ (\text{Message}\ r))
\]

\[
\begin{align*}
&\text{deliver}\ p = \text{case fPop (pToSelf p) of} \\
&\quad \text{Just (m, toSelf)} \rightarrow \\
&\quad \quad \text{( p[toSelf=toSelf, pVC=vcCombine (pVC p) (mSent m)}} \\
&\quad \quad \quad , \text{Just m)} \\
&\quad \text{Nothing} \rightarrow \text{case dqDequeue (pVC p) (pDQ p) of} \\
&\quad \quad \text{Just (dq, m)} \rightarrow \\
&\quad \quad \quad \text{( p[pDQ=dq, pVC=vcCombine (pVC p) (mSent m)}} \\
&\quad \quad \quad , \text{Just m)} \\
&\quad \text{Nothing} \rightarrow (p, \text{Nothing})
\end{align*}
\]

\[
\)\]

\[
\text{drainBroadcasts} :: \text{Process}\ r \rightarrow (\text{Process}\ r, [\text{Message}\ r])
\]

\[
\begin{align*}
&\text{drainBroadcasts}\ p = \\
&\quad \text{( p[pToNetwork=fNew} \\
&\quad \quad , \text{fList (pToNetwork p)}
\end{align*}
\]

Listing 3. Implementation of the external API provided by our causal broadcast implementation. The \textit{vcNew} operation returns a new \textit{VC} initialized with zeroes, and \textit{dqNew} creates a new delay queue. \textit{vcTick} increments the specified index in a \textit{VC}, and \textit{vcCombine} takes the pointwise maximum of \textit{VCs}. Operations on the \textit{FIFO} data type (\textit{fNew}, \textit{fPush}, \textit{fPop}, \textit{fList}) have the expected semantics.

- The \textit{receive} function (lines 21-24) takes a message and adds it to the delay queue if it is from a different sender. \textit{receive} ignores self-sent messages, since \textit{send} already handles them by adding them to the \textit{pToSelf} buffer.
- The \textit{deliver} function (lines 26-37) pops the first message from the \textit{pToSelf} buffer, if any, or dequeues the first deliverable message from the delay queue, if any, and otherwise returns \textit{Nothing}. If a message \textit{m} is returned, \textit{deliver} implements the delivery side of the vector clock.
Listing 4. Carol delays a message (line 29) from Bob that is causally dependent on a message from Alice. After Carol delivers the causal dependency (line 33), the delayed message is delivered (line 36).

protocol of Section 3.1 by updating the process vector clock to the pointwise maximum of $VC(m)$ and the current process vector clock.

- The `drainBroadcasts` function (lines 39-43) empties and returns the contents of the `pToNetwork` buffer for the application or transport layer to broadcast to the other processes.

5.4 API Example

We illustrate the use of the API with an example. Recall the confusion about Alice’s lost wallet from Figure 1, which was cleared up with the help of causal broadcast in Figure 4. We now revisit the “lost wallet” example by implementing it in terms of our API.

Listing 4 simulates the client behavior from Figure 4 (right) to demonstrate how our protocol implementation reorders messages. Lines 1-3 of Listing 4 launch a new `Process` for each of Alice, Bob, and Carol by calling `pNew`. On lines 6-7, Alice’s process broadcasts “I lost my wallet...” to Bob and Carol by calling `send` followed by `drainBroadcasts`, at which point a transport layer
carries the message to Bob’s and Carol’s processes. A bit later, Alice finds her wallet and broadcasts "Found it!" in the same way (lines 10-11). Bob receives Alice’s broadcasts in one foldr step (line 15), delivers them (lines 16-17), and broadcasts a kind response (lines 24-25). However, Carol receives only Alice’s first message (line 20) before Bob’s missive (line 28) and so Bob’s message is delayed (line 29). Only after receipt and delivery of "Found it!" (lines 32-33) is Carol able to deliver the buffered message from Bob (line 36).

Consider the utility and purposes of the four primary API functions in light of this example: The send and receive functions are both ways to inject message data into a local Process r, for different purposes, and the drainBroadcasts and deliver functions are both ways to extract message data from a Process r. The send and drainBroadcasts functions both have the purpose of conveying information outward to the causal broadcast cluster, and the receive and deliver functions both have the purpose of conveying information inward from the cluster.

6 CASE STUDY: A CAUSALLY CONSISTENT KEY-VALUE STORE

Using our causal broadcast API of Section 5, we implemented a replicated key-value store application. Our application uses the architectural pattern depicted by Figure 5. The key-value store service is provided by an HTTP server that fields requests from clients, and simultaneously broadcasts state updates to a cluster of other key-value store instances, each of which support additional clients. Our application demonstrates that it is not difficult to integrate our causal broadcast framework with an application and provide appropriate message transport to realize the benefits of causal broadcast for our application, namely, causal consistency of replicated data [Ahamad et al. 1995; Lloyd et al. 2011].

The HTTP service is implemented with the Haskell packages servant, in which endpoints are specified by types, stm to express multithreaded access to state, and aeson to provide JSON (de)serialization. Clients may request to PUT a value at a key, DELETE a key-value pair specified by key, or GET the value corresponding to a specified key. Servers may POST a message to each other, and do so to implement broadcast using direct HTTP requests. When a client requests to modify application state via PUT or a DELETE, the endpoint generates a value of type KvCommand to update the causal broadcast cluster, where KvCommand is defined as follows:

```haskell
type Key = String

-- Define a value type

type Val = Aeson.Value

-- Define a data type

data KvCommand = KvPut Key Val | KvDel Key
```

Each server runs a Process where the type of raw messages is KvCommand. We define the following type aliases:

```haskell

-- Define a node state type

type NodeState = Process KvCommand

type KvState = Map Key Val
```

The PUT and DELETE endpoints call send on the NodeState and the KvCommand value, waking up two background threads waiting on stm references. The send mail background thread uses drainBroadcasts to POST the Message KvCommand value to the other nodes in the causal broadcast cluster, and the read mail thread uses deliver to apply the Message KvCommand to the local KvState. The POST endpoint calls receive to inject Message KvCommand values from other nodes.

---

9 Since our Val type is Aeson.Value, the key-value store can actually be considered a JSON document store, which is a common type of NoSQL database found in industry.

10 To keep the client/server interface simple, we adopt a "sticky sessions" model, in which a given client will only ever talk to a given server. In a setting where clients can communicate with more than one server, clients would need to participate in the propagation of causal metadata generated by the servers [Lloyd et al. 2011].
into the NodeState, which also triggers the read mail background thread to wake up and attempt deliver. Finally, the GET endpoint performs Map lookup on the KvState to answer clients’ queries.

Since the messages received via the POST endpoint are from other nodes, deliver will return Nothing in cases where the causal dependencies of the message are not satisfied. Therefore all nodes (and hence all clients of those nodes) observe the effects of causally-related KvCommands in the same (causal) order, giving us causal consistency. By adding indexing of values inserted with KvCommand and a richer query set to take advantage of the indexing, our key-value store could be easily extended to provide real utility in a production setting. We deployed our key-value store running on top of our verified causal broadcast protocol implementation on 8 nodes and ran some simple experiments to confirm that data was being broadcast as expected.

7 RELATED WORK

Machine-checked correctness proofs of executable distributed protocol implementations. Much work on specification and verification of distributed systems has focused on specifying and verifying properties of models using tools such as TLA+ [Lamport 2002], rather than of executable implementations. The state of the art for machine-checked correctness proofs of executable distributed protocol implementations includes Verdi [Wilcox et al. 2015], IronFleet [Hawblitzel et al. 2015], ShadowDB [Schiper et al. 2014], and Chapar [Lesani et al. 2016].

Verdi [Wilcox et al. 2015] is a Coq framework for implementing distributed systems; verified executable OCaml implementations can be extracted from Coq. IronFleet [Hawblitzel et al. 2015] uses the Dafny verification language, which compiles both to verification conditions checked by an SMT solver and to executable code. Both Verdi and IronFleet have been used to verify safety properties (in particular, linearizability) of distributed consensus protocol implementations (Raft and Multi-Paxos, respectively) and of strongly-consistent key-value store implementations, and IronFleet additionally considers liveness properties. The ShadowDB project [Schiper et al. 2014] uses a language called EventML that inverts the extraction workflow used in a proof assistant like Coq or Isabelle: instead of first carrying out a proof in a proof assistant and then extracting an executable implementation, the programmer writes code in EventML, which compiles both to a logical specification and to executable code that is automatically guaranteed to satisfy the specification, and correctness properties of the logical specification can then be proved using the Nuprl proof assistant. Schiper et al. [2014] used this workflow to verify the correctness of a Paxos-based atomic broadcast protocol. None of these projects looked at causal broadcast or causal message ordering in particular.

Chapar [Lesani et al. 2016] presented a technique and Coq-based framework for mechanically verifying the causal consistency of distributed key-value store (KVS) implementations, with executable OCaml KVSes extracted from Coq. Lesani et al.’s verification approach effectively bakes a notion of causal message delivery into an “abstract causal operational semantics”, which specifies how a causally consistent KVS should behave. They then used the Chapar framework to check that a KVS implementation satisfies that specification. The executable KVSes extracted from Chapar can safely run on top of a messaging layer that does not provide causal delivery. Chapar is specific to the KVS use case, whereas our verified causal broadcast implementation factors out causal message delivery into a separate layer, agnostic to the content of messages, that can be used as a standalone component in a variety of applications rather than just KVSes. Our Liquid Haskell implementation is also immediately executable Haskell, with no need for an extraction step, which considerably simplifies integration of the extracted code with code in the target language. While we did not attempt to verify the causal consistency of our case-study KVS, the fact that it is built on top of a verified causal broadcast protocol would likely simplify the verification process compared to the Chapar approach. We intend to pursue KVS verification in future work.
Mechanized reasoning about causal consistency. One of the most important applications of causal broadcast is keeping distributed replicas of data causally consistent [Ahmad et al. 1995; Lloyd et al. 2011] across a number of nodes. Out of dozens of possible data consistency policies [Viotti and Vukolić 2016], causal consistency represents an appealing “sweet spot” in the consistency/availability trade-off space, letting replica states diverge when necessary to preserve availability while still ensuring that causal dependencies between operations are respected. Various SMT-powered verification tools [Gotsman et al. 2016; Sivaramakrishnan et al. 2015] enable automatically verifying that a given application invariant or operation contract holds under a given consistency policy, including causal consistency. Rather than verifying that causal consistency itself is satisfied, these approaches assume that the underlying data store provides a given consistency guarantee, and then prove that application-level invariants are satisfied.

Causal broadcast for CRDT convergence. Conflict-free replicated data types (CRDTs) [Shapiro et al. 2011a,b] are data structures designed for replication. Their operations must satisfy certain mathematical properties that can be leveraged to ensure strong convergence [Shapiro et al. 2011b], meaning that replicas are guaranteed to have equivalent state if they have received and applied the same unordered set of updates. While some CRDTs ask little of the underlying messaging layer to ensure convergence (for instance, for an integer counter that can be incremented and decremented, all updates commute, so order of message delivery is irrelevant), many CRDTs that implement container types, such as Roh et al.’s Replicated Growable Array (RGA) [2011] or Shapiro et al.’s two-phase set [2011b], rely on causal delivery to ensure that, for example, a message that updates or deletes an element of a set will not be delivered before the message that inserts that element. (Nagar and Jagannathan [2019] recently established that causal consistency, and hence causal message delivery, suffices to ensure convergence of many standard CRDTs from the literature, although it is insufficient for some more sophisticated CRDTs and unnecessary for the simplest ones.)

Gomes et al. [2017] use the Isabelle/HOL proof assistant [Wenzel et al. 2008] to implement and verify the strong convergence of several CRDTs, including RGA. To carry out the proof, they bake in causal delivery as an underlying assumption, modeled by the “network axioms” in their Isabelle proof development. Therefore, for strong convergence to hold for an actual deployed implementation of Gomes et al.’s CRDTs, the deployment environment must provide causal delivery. Our work implements just such an environment, with its safety verified by Liquid Haskell. Thus our work is complementary to Gomes et al.’s: one could deploy their verified-convergent CRDTs atop our verified-safe causal broadcast protocol to get an “end-to-end” guarantee on top of a weaker network model that offers no causal delivery guarantee itself.

Liu et al. [2020] use Liquid Haskell to verify the convergence of several operation-based CRDT implementations. Their work differed from Gomes et al.’s in that it did not assume causal delivery, and therefore required less of the deployment environment; on the other hand, it took a more strenuous implementation and verification effort, requiring on the order of thousands of lines of Liquid Haskell proofs for the more sophisticated CRDTs. In fact, Liu et al.’s verified two-phase map implementation included a “pending buffer” for updates that arrived out of order, and a collection of ad hoc, data-structure-specific rules to determine which updates should be buffered and which should be immediately applied. These mechanisms resemble the delay queue and the deliverable predicate from our causal broadcast implementation, but are specific to a particular application-level data structure and use an ad hoc delivery policy, rather than operating at the messaging layer and using the more general principle of causal delivery. We suspect that our separately-verified causal broadcast implementation would obviate the need for such mechanisms and simplify the implementation and verification of CRDTs, and we plan to investigate this in future work.
**Other applications of causal delivery.** Aside from causally consistent data stores and convergent CRDTs, causal delivery is useful for applications that must detect whether a *stable property* [Chandy and Lamport 1985] holds of a distributed system. A stable property is a property that, once becoming true, remains true for the rest of an execution; examples of stable property detection include deadlock detection and termination detection. Causal delivery can simplify the implementation of such algorithms [van Renesse 1993]. Not unrelatedly, some snapshot algorithms for recording the global state of a distributed system [Acharya and Badrinath 1992; Alagar and Venkatesan 1994] rely on causal message delivery, which simplifies their implementation compared to snapshot algorithms for systems that lack causal delivery support [Kshemkalyani et al. 1995].

**Foundational work on causal delivery.** We implemented and mechanically verified the causal broadcast protocol proposed by Birman et al. [1991]. The notion of causal delivery and a protocol for causal broadcast was originally proposed by Birman and Joseph [1987b], although this earlier design required messages to include a copy of every causally preceding message, necessitating a garbage-collection mechanism to clean up extra message metadata. Schiper et al. [1989] proposed a more general protocol that ensures causal delivery of point-to-point messages in addition to broadcast messages. All these papers give relatively informal proofs or proof sketches to aid intuition about the correctness of their protocols. Verifying the correctness of an implementation of causal delivery for point-to-point messages would be an interesting direction for future work, although such protocols are less often used, in part because their message metadata overhead is necessarily higher than that of causal broadcast.

## 8 DISCUSSION

As we carried out this project, the statement of the safety property we wanted to prove went through many iterations. In particular, we only arrived at a version of the property that used the `DeliverableProp` and `CausallyBeforeProp` types from Section 4.3 after we stepped back from Liquid Haskell, carried out the proof in Agda [Norell 2008] instead, then ported it back to Liquid Haskell. Agda’s dependent function types suggested to us a natural and useful way to encode the property we wanted to prove. However, Agda lacks Liquid Haskell’s reflection mechanism. When we ported our Agda proof back to Liquid Haskell, we realized that, thanks to reflection, the return types of `DeliverableProp` and `CausallyBeforeProp` could refer directly to `deliverableK` and `causallyBeforeK` respectively, letting us straightforwardly state and prove a property of our running implementation. Furthermore, the Liquid Haskell proof takes advantage of SMT automation, thus it is shorter.\(^1\)

A selling point of Liquid Haskell is that it promises a way to do “gradual verification” of existing Haskell code, letting programmers add refinement types and prove extrinsic properties about existing code a little at a time, and with each step resulting in immediately executable code that can be used directly from arbitrary Haskell programs. We began this project with such a gradual verification workflow in mind, and so we started by implementing the causal broadcast protocol in plain Haskell, intending to add refinement types later without having to change the implementation much. What we found instead was that we had to make invasive changes to our code to make it amenable to verification, mostly involving removing dependencies on third-party libraries that interacted poorly with Liquid Haskell. For example, we arrived at the current external API design only after attempting a different design and then discarding it because it involved calls to functions from a third-party library that could not be reflected into the refinement logic. On the other hand, in our eagerness to use Liquid Haskell to the fullest, we attempted to reflect many functions

---

\(^1\)Our Agda proof can be found in the submitted supplementary material.
that turned out not to be necessary to state and prove the safety property we ultimately arrived at. We are reminded of the wisdom of Dijkstra [1990]'s urging "not to try to prove the correctness of existing programs, but to design correctness proof and program together."

Finally, while we have contributed a machine-checked proof of causal safety of an executable causal broadcast implementation, we want to acknowledge the gap between what our safety property ensures, and what we would like to claim is true of our running implementation. As Section 5 notes, we (like nearly all efforts to verify implementations of distributed protocols) address only safety, not liveness. Furthermore, the safety property we prove is about the internal behavior of an aspect of our implementation; we cannot guarantee that the external API we provide is free of bugs, and given that bugs often occur at interfaces or shim layers between verified and unverified components [Fonseca et al. 2017], it would be irresponsible to make such a claim. Narrowing the size of this gap between verification and implementation will be an important goal for future work.

REFERENCES


Verified Causal Broadcast with Liquid Haskell

2737981


