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 that messages will never be delivered to a process in an order that violates causality. To do so, we define a correctness condition, *causal safety*, that implies causal delivery; express causal safety using refinement types; and prove that it holds of our implementation 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 few 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 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

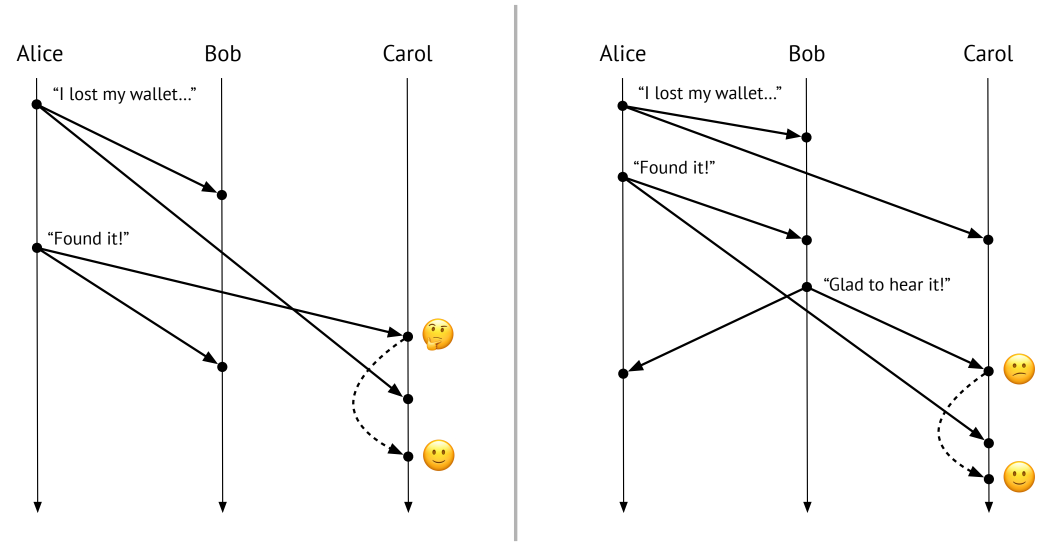

Fig. 1. Two examples of executions that violate causal delivery. The vertical direction represents time, where later is lower; the horizontal direction represents space. Solid arrows represent messages between processes. On the left, Carol sees Alice's messages in the opposite order of how they were 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.

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].[1]

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.

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

[1]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.

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 this theorem-proving capability of 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.* To our knowledge, it is the first distributed messaging protocol implementation of any kind to be verified with Liquid Haskell.[2] To carry out our proof, we first define a new correctness condition, *causal safety*, that implies causal delivery (Section 2). We then use Liquid Haskell to prove that our implementation satisfies causal safety (Section 4). Our approach leverages SMT automation in Liquid Haskell to verify domain properties of causal broadcast data structures and functions directly in the production implementation language, Haskell, as well as to check the proof about how those functions interact. We were able to simplify the structure of our proof by using a lemma that relates iteration to universal quantification and by defining several data types that exploit that relationship.

Our verified causal broadcast implementation is a Haskell library that can be used in a variety of applications, 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 library could be plugged together with such verified CRDT implementations to get an end-to-end correctness guarantee. Therefore our approach enables *modular* verification of higher-level properties for applications built on top of the causal broadcast layer.

[2]Previous work [Liu et al. 2020] used Liquid Haskell to verify the convergence of distributed data structures. Our work verifies a property of the underlying messaging layer, while remaining completely agnostic to the content of the messages; see Section 7 for further comparison.

[3]Totally-ordered delivery does not imply causal delivery in general, although Birman et al. [1991]'s extension of causal broadcast to atomic broadcast provides both.

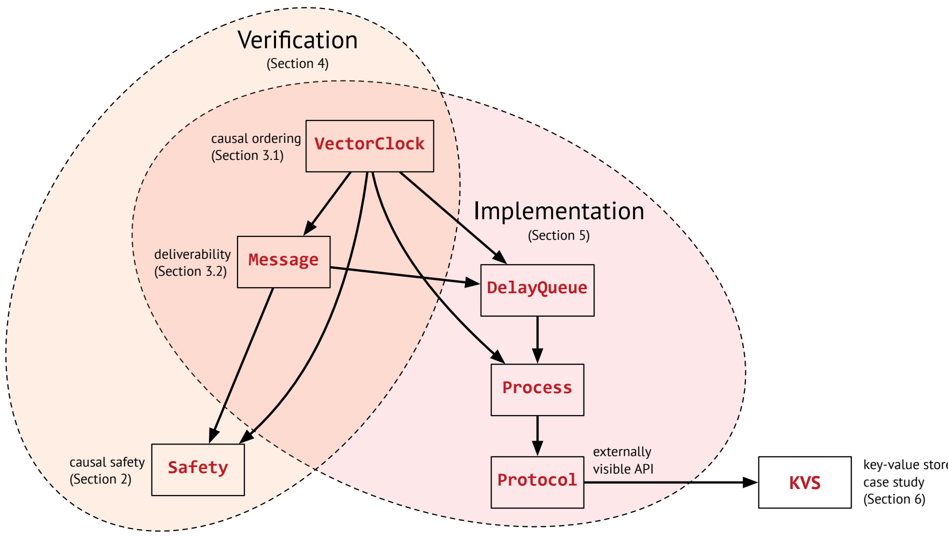

Fig. 2. Dependencies between the modules of our proof development and causal broadcast implementation. The `VectorClock` and `Message` modules are part of both the running implementation and of the proof of causal safety, connecting verification and implementation.

An 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 library with existing Haskell code. To demonstrate the usability of our library, we put it to work as the foundation of a causally-consistent key-value store and evaluate its performance (Section 6). We implemented the key-value store using widely-used, off-the-shelf Haskell libraries (e.g., `stm`, `servant`, `aeson`), and the verification imposes no run-time overhead.

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 causal safety 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 of our proof of causal safety (`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 SYSTEM MODEL AND CAUSAL SAFETY

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]

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 a 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 $broadcast(m)$ to denote the event that sends a message m to all processes, and $deliver_p(m)$ 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. In any execution, there can be at most one $deliver_p(m)$ event for a given message m and process p.

The *state* of a process p is the sequence of events that have occurred on p. The state of p *prior to* a given event e on p is the subsequence of events on p that precede e. For a message m and a process p with state s, we can define a predicate $\mathrm{delivered}(m, s)$ that holds if $deliver_p(m)$ is an event anywhere in s.

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, $e = broadcast(m)$ and $e' = deliver_p(m)$ 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 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.

[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.

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 $broadcast(m) \rightarrow broadcast(m')$. 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*; our definition is based on standard ones in the literature [Birman et al. 1991; Raynal et al. 1991]:

Definition 2 (Causal delivery). *An execution observes* causal delivery *if, given a process p with state s, for all messages m_1 and m_2 such that* $\mathsf{delivered}(m_1, s)$ *and* $\mathsf{delivered}(m_2, s)$*,*

$$m_1 \rightarrow m_2 \implies deliver_p(m_1) \rightarrow_p deliver_p(m_2).$$

The causal delivery property says that if message m_1 is sent before message m_2 in an execution, then any process delivering both m_1 and m_2 should deliver m_1 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, under causal delivery, m_1 and m_2 must be delivered in causal order even if they were sent by different processes. 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 deliverable that takes as arguments a message m and a process state s and returns true if m can be delivered at a process in state s and false otherwise. We say that an execution X is *guarded by* a given deliverable predicate if, for any message m and process p, for any $deliver_p(m)$ event in X, $\mathsf{deliverable}(m, s)$ holds, where s is the state of p prior to $deliver_p(m)$.

We introduce the notion of *causal safety* to express the behavior that we want from deliverable. To get causal delivery, we want $\mathsf{deliverable}(m, s)$ to return true if m can be delivered at a process in state s without causing a causality violation, and false otherwise. In particular, given two messages m_1 and m_2 where $m_1 \rightarrow m_2$, if the deliverable predicate deems m_2 to be deliverable at a process in state s, then m_1 should have already been delivered at p. We define causal safety as follows:

Definition 3 (Causal safety). *A predicate* deliverable *is* causally safe *if, for all messages m_1, m_2 and all process states s,*

$$\mathsf{deliverable}(m_2, s) \land m_1 \rightarrow m_2 \implies \mathsf{delivered}(m_1, s).$$

If we have a deliverable predicate that is causally safe, we can use it to ensure causal delivery of executions:

Theorem 1 (Causal safety implies causal delivery). *Given a causally safe predicate* deliverable *and an execution X, if X is guarded by* deliverable*, then X observes causal delivery.*

Proof. Consider any process p in X, where s is the state of p, and any two messages m_1, m_2 such that $m_1 \rightarrow m_2$ and $\mathsf{delivered}(m_1, s)$ and $\mathsf{delivered}(m_2, s)$. We need to show that $deliver_p(m_1) \rightarrow_p deliver_p(m_2)$. Since we already know (from $\mathsf{delivered}(m_1, s)$ and $\mathsf{delivered}(m_2, s)$) that $deliver_p(m_1)$

and $\mathit{deliver}_p(m_2)$ are both events on p, we only need to show that they happen in the specified order. Let s' be the state of p prior to $\mathit{deliver}_p(m_2)$. Since X is guarded by deliverable, deliverable(m_2, s') holds. Since deliverable is causally safe and $m_1 \to m_2$, then delivered(m_1, s') holds, meaning that m_1 has been delivered at p prior to $\mathit{deliver}_p(m_2)$. Therefore it must be the case that $\mathit{deliver}_p(m_1) \to_p \mathit{deliver}_p(m_2)$. □

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 deliverable predicate holds or does not hold for a given message *during* the run of the program. In the following sections, we will show how to implement a causally safe deliverable 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 $\to$; 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] Rather than needing to keep a complete history of events as described in Section 2, a process need only keep track of its own vector clock. 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.

[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.

- 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 at least as big as m_1's clock at every position.

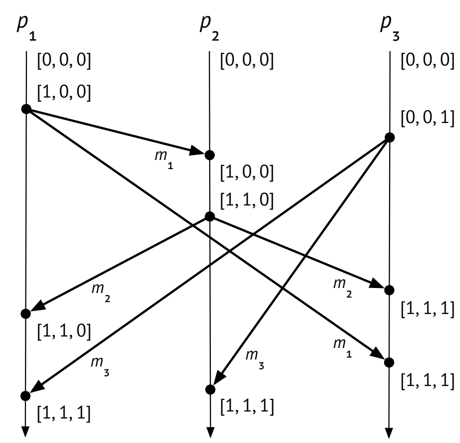

Fig. 3. An example execution using the vector clock protocol.

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', it can be shown that

$$m \rightarrow m' \iff VC(m) < VC(m').$$

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

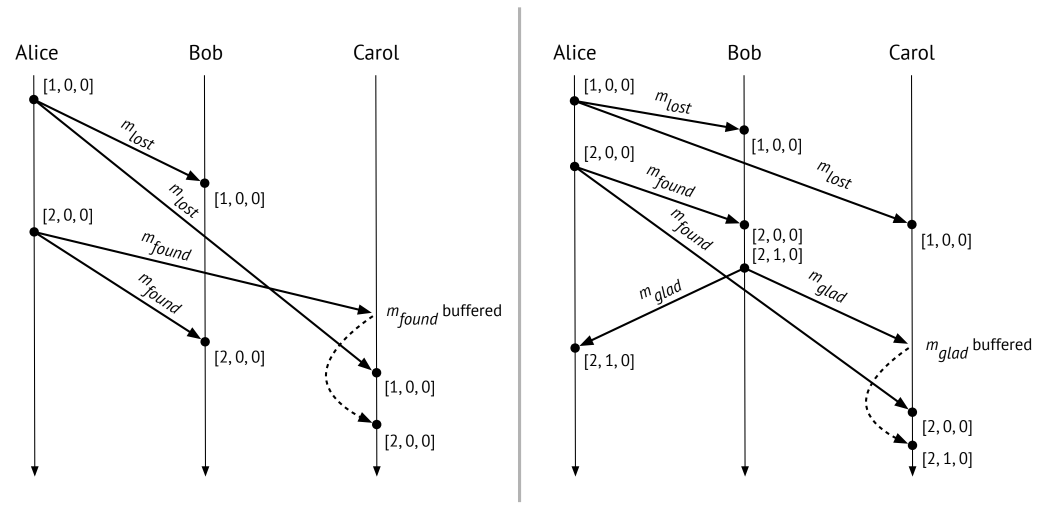

Fig. 4. The executions from Figure 1, annotated with vector clocks used by the causal broadcast protocol. On the left, Carol buffers message m_{found}, which has a vector clock of `[2,0,0]`, until she has received and delivered m_{lost}, which has a vector clock of `[1,0,0]`. On the right, Carol buffers message m_{glad}, which has a vector clock of `[2,1,0]`, until she has received and delivered m_{found}, which has a vector clock of `[2,0,0]`.

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 self-sent messages are sent and delivered locally 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 $k : 1..N$,*

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

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 is.)

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 $deliver_{p_i}(m') \rightarrow broadcast(m)$, so by transitivity of $\rightarrow$ we have $broadcast(m') \rightarrow broadcast(m)$. Therefore $m' \rightarrow m$, 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. Whenever a process receives a message, it buffers the message until it is deliverable according to Definition 4. Each process stores messages that need to be buffered in a process-local 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_{lost} is Alice's "I lost my wallet..." message, m_{found} is Alice's "Found it!" message, and m_{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_{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_{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, after Carol delivers m_{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 prove that our implementation is causally safe using Liquid Haskell's theorem-proving capabilities.

4.1 Background: 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 a 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 can define a type of even `Int`s using the refinement type `{ v:Int | v mod 2 == 0 }`, where `v mod 2 == 0` is the refinement predicate and `v:Int` binds the name `v` for values of type `Int` that appear in the refinement predicate. One could define an analogous refinement type for odd integers and then write a function for adding them:

```
type EvenInt = { v:Int | v mod 2 == 0 }
type OddInt = { v:Int | v mod 2 == 1 }

oddAdd :: OddInt -> OddInt -> EvenInt
oddAdd x y = x + y
```

The type `OddInt` of the arguments to `oddAdd` expresses the *precondition* that `x` and `y` will be odd, and the return type `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 `oddAdd` had been `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 `sumOdd` below expresses the extrinsic property that the sum of an odd and an even number is an odd number:

```
sumOdd :: x : OddInt -> y : EvenInt -> { _:Proof | (x + y) mod 2 == 1 }
sumOdd _ _ = ()
```

Here, `sumOdd` is a Haskell function that returns a *proof* that the sum of `x` and `y` is odd. (In Liquid Haskell, `Proof` is a type alias for Haskell's `()` (unit) type.) Because the proof of this particular property is easy for the SMT solver to carry out automatically, the body of the `sumOdd` function need not say anything but `()`. In general, however, programmers can specify arbitrary extrinsic properties in refinement types, including properties that refer to arbitrary Haskell functions via the notion of *reflection* [Vazou et al. 2017]. The programmer can then prove those extrinsic properties by writing Haskell programs that inhabit those refinement types, using Liquid Haskell's provided *proof combinators* — 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 `oddAdd` (which runs at execution time, as usual) and verification code like `sumOdd` (which is never run, but merely typechecked), 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. For instance, a programmer might begin with an implementation of `oddAdd` with the type `Int -> Int -> Int`, later refine it to `OddInt -> OddInt -> EvenInt`, even later prove the extrinsic property `sumOdd`, and still later use the proof returned by `sumOdd` as a premise to prove another, more interesting extrinsic property. 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.

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 `VectorClock` module an "uninterpreted" (that is, without a Haskell implementation) Liquid Haskell *measure* [Vazou et al. 2014] called `procCount`, of `Nat` type, and a type `ProcCount` based on it, as follows:

```
type Nat = { v : Int | 0 <= v }
measure procCount :: Nat
type ProcCount = { s : Nat | s == procCount }
```

We define a process identifier (and index into vector clocks) type `PID` as

```
type PID = Fin procCount
```

where `Fin V` is the type of natural numbers less than `V`, since our vector clocks are 0-indexed:

```
type Fin V = { k : Nat | k < V }
```

A vector clock index of `PID` type is therefore guaranteed to never index out of bounds.

We also define a vector clock type, `VC`, that makes use of `procCount`. We first define the type `Vec` of length-indexed vectors:[6]

```
type Vec a V = { v : [a] | len v == V }
```

and then define `VC` using `Vec` as follows:

```
data VC = VC (Vec Nat procCount)
```

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

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

```
data Message r = Message { mSender::PID, mSent::VC, mRaw::r }
```

`mSender` is the process identifier of the process that sent the message, and `mSent` is the vector clock attached to the message. The `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 `Message` module (shown in Figure 2) provides a predicate `deliverable` (shown in Listing 1) of type `Message r -> VC -> Bool` that determines the deliverability of a message at a process. The `VC` argument to `deliverable` represents the state of the process receiving the message. Although our implementation has a record type for processes, which has a `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 `deliverable` needs, so a `VC` is the only argument that needs to be passed.

We want to prove that our `deliverable` predicate has the causal safety property of Definition 3, where the `VC` argument to `deliverable` corresponds to the process state s in Definition 3.

Theorem 2. `deliverable` *is causally safe.*

[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. Here, `Vec` is parameterized with the Haskell type variable `a` and the Liquid Haskell expression variable `V`.

```
1  deliverable :: Message r -> VC -> Bool
2  deliverable m procVc = vcSize (mSent m) `iter` deliverableK m procVc
3
4  iter :: n : Nat -> (Fin n -> Bool) -> Bool
5  iter n f = listFoldr boolAnd True (listMap f (fin n))
6
7  deliverableK :: Message r -> VC -> PID -> Bool
8  deliverableK m procVc k
9      | k == mSender m = mSent m ! k == (procVc ! k) + 1
10     | otherwise      = mSent m ! k <= procVc ! k
```

Listing 1. The `deliverable` predicate and its helper functions. `deliverableK` takes as arguments a message `m`, a vector clock `procVC`, and a vector clock index `k`, and returns `True` or `False`. The `procVC` argument is the vector clock of the receiving process. `mSender m` is the `PID` of the sender of `m`, and `mSent m` is the vector clock attached to `m`. `deliverableK` implements the behavior specified in Definition 4: in the case where `k == mSender m` — that is, the case where `k` is the `PID` of the process that sent `m` — `deliverableK` checks that `m`'s vector clock is one greater than the receiving process's vector clock at index `k`, and otherwise, `deliverableK` checks that `m`'s vector clock is less than or equal to the process's vector clock at index `k`.

We can express the claim made by Theorem 2 as a refinement type in Liquid Haskell as follows:

```
causalSafety
    :: procVc : VC
    -> m1 : Message r
    -> m2 : Message r
    -> { _:Proof | deliverable m2 procVc }
    -> CausallyBeforeProp m1 m2
    -> DeliveredProp m1 procVc
```

The interesting parts of `causalSafety`'s type are the `{ _:Proof | deliverable m2 procVc }`, `CausallyBeforeProp m1 m2`, and `DeliveredProp m1 procVc` types. We first summarize their meanings briefly before explaining them in more detail in the rest of this section.

- `{ _:Proof | deliverable m2 procVc }` is the type of a Liquid Haskell *proof* that message `m2` is deliverable at a process with the vector clock `procVC`. `deliverable` is the executable Haskell function that we wish to prove the causal safety of. It appears in the type of `causalSafety` by means of Liquid Haskell's reflection mechanism [Vazou et al. 2017] that lets arbitrary (terminating) Haskell function calls appear in refinement predicates.
- `CausallyBeforeProp m1 m2` is the type of a proof that message `m1` causally precedes message `m2`.
- `DeliveredProp m1 procVc` is the type of a proof that message `m1` has been delivered at a process with the vector clock `procVc`.

4.3.1 Deliverability as a Refinement Type. Listing 1 shows our `deliverable` predicate, which implements the notion of deliverability from Definition 4. `deliverable` is implemented in terms of two short helper functions, `iter` and `deliverableK`. The `deliverableK` function checks the deliverability condition from Definition 4 (lines 9-10 of Listing 1) at a given vector clock index `k`. The `iter` function folds a predicate over a list of a specified length `n`. `deliverable` folds `deliverableK` over all the entries in those vector clocks, using `iter`. Because Liquid Haskell can determine that `deliverable` is terminating, we can refer to it in a refinement predicate in the type of `causalSafety`.

4.3.2 Causal Precedence as a Refinement Type. After the proof of deliverability, the next argument to `causalSafety` is of type `CausallyBeforeProp m1 m2`. `CausallyBeforeProp` is a type synonym for the type of a function that takes as its argument a `PID k` and returns a proof:

```
type CausallyBeforeProp M1 M2
    =   k : PID
    ->  { _:Proof | vcReadK (mSent M1) k <= vcReadK (mSent M2) k
                 &&          mSent M1     /=          mSent M2     }
```

The `CausallyBeforeProp` type exploits the isomorphism between the vector clock ordering and the happens-before relation discussed previously in Section 3.1: to check if a message `m1` causally precedes a message `m2`, we just need to check that, for all indices `k`, $VC(\texttt{m1})[\texttt{k}] \leq VC(\texttt{m1})[\texttt{k}]$ and $VC(\texttt{m1}) \neq VC(\texttt{m2})$. We implement this check in `CausallyBeforeProp` in terms of a `vcReadK` function (provided by the `VectorClock` model shown in Figure 2) that returns the value of a vector clock at a given index.

4.3.3 Deliveredness as a Refinement Type. Finally, `causalSafety` returns a value of type `DeliveredProp m1 procVc`. Like `CausallyBeforeProp` above, `DeliveredProp` is a Liquid Haskell type synonym for the type of a function that takes a `PID k` and returns a proof:

```
type DeliveredProp M P
    =   k : PID
    ->  { _:Proof | vcReadK (mSent M) k <= vcReadK P k }
```

To check that a message has been delivered at a process, `DeliveredProp` checks that the process vector clock is at least as large as the message's vector clock at all indices `k`. Since the vector clock protocol of Section 3.1 updates a process's vector clock to the pointwise maximum of the process vector clock and the message vector clock upon message delivery, we can know that this property holds if a message has been delivered.

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 2 by inhabiting the type of `causalSafety` with a program. Listing 2 shows our causal safety property and its 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 `causalSafety`; now we dive into the proof itself.

To carry out the proof, we first establish a relationship between the `deliverable` and `deliverableK` functions described in Section 4.3.1. In particular, we want to show that if `deliverable` holds for a given messsage and a given process with particular vector clocks, then `deliverableK` holds for all indices `k` into those respective vector clocks. In other words, if we define `DeliverableProp` to be the type of a function that takes a `k` (just like `CausallyBeforeProp` and `DeliveredProp` from the previous section) and returns a proof of `deliverableK` for that `k`, as follows:

```
type DeliverableProp M P
    =  k : PID
    -> { _:Proof | deliverableK M P k }
```

then we want to prove that, given a proof of `deliverable m p`, we can get a proof of `DeliverableProp m p`. This is straightforward to do in Liquid Haskell. In fact, we can prove a more general property that if `iter n p` holds for a given `n` and predicate `p`, then `p k` holds for all `k` less than `n`. Lines 1-7 of Listing 3 give this more general property, which we call `iterImpliesForall`,

```
1  assume vcAxiom
2      :: m1 : Message r -> m2 : Message r -> CausallyBeforeProp {m1} {m2}
3      -> { _:Proof | vcReadK (mSent m1) (mSender m2) < vcReadK (mSent m2) (mSender m2) }
4  vcAxiom _m1 _m2 _m1_before_m2 = ()
5
6  causalSafety
7      :: procVc : VC -> m1 : Message r -> m2 : Message r
8      -> { _:Proof | deliverable m2 procVc }
9      -> CausallyBeforeProp m1 m2
10     -> DeliveredProp m1 procVc
11 causalSafety procVc m1 m2 m2_deliverable_p m1_before_m2 k
12     | k /= mSender m2 = m1_before_m2 k
13                         ? (deliverableImpliesDeliverableProp procVc m2 m2_deliverable_p) k
14     | k == mSender m2 = m1_before_m2 k
15                         ? (deliverableImpliesDeliverableProp procVc m2 m2_deliverable_p) k
16                         ? vcAxiom m1 m2 m1_before_m2
17                         *** QED
```

Listing 2. Our causal safety property and its proof in Liquid Haskell.

```
1  iterImpliesForall
2      :: n : Nat -> p : (Fin n -> Bool)
3      -> { _:Proof | iter n p }
4      -> (k : Fin n -> { _:Proof | p k })
5  iterImpliesForall n p satisfied k
6      | k == n - 1 = ()
7      | k <  n - 1 = iterImpliesForall (n - 1) p satisfied k
8
9  deliverableImpliesDeliverableProp
10     ::  p: VC
11     ->  m : Message r
12     ->  { _:Proof | deliverable m p }
13     ->  DeliverableProp m p
14 deliverableImpliesDeliverableProp p m m_deliverable_p k =
15     iterImpliesForall (vcSize (mSent m)) (deliverableK m p) m_deliverable_p k
```

Listing 3. The `iterImpliesForall` property and its proof in Liquid Haskell, and a corollary, `deliverableImpliesDeliverableProp`, along with its proof.

and its proof in Liquid Haskell by induction on `n`. We can then state and prove on lines 9-15 of Listing 3 our desired property linking `deliverable` to **`DeliverableProp`**, in terms of `iterImpliesForall`.

Our causal safety proof depends on one assumed axiom, `vcAxiom`, shown on lines 1-4 of Listing 2. (The `assume` keyword in Liquid Haskell lets one state and use a property without providing a proof.) `vcAxiom` encodes a standard observation about the vector clock protocol of Section 3.1: if message m_1 causally precedes message m_2, then $VC(m_1)[k]$ will be *strictly less* than $VC(m_2)[k]$ where k is the index of m_2's *sender*. The intuition is that m_2's sender knows about m_2 at the time it sends m_2, but, since $m_1 \rightarrow m_2$, the sender of m_1 cannot know about m_2 at the time it sends m_1. Therefore m_2's vector clock will have an entry at m_2's sender's position that takes m_2 into account, but m_1's vector clock entry at the same position must be at least one less, because m_1's sender lacked knowledge of m_2.

Lines 6-10 of Listing 2 are the type of `causalSafety`, and the body of the function begins on line 11. As we saw in the previous section, `causalSafety`'s return type of **`DeliveredProp`** `m1 procVc` is the type of a function that takes an argument `k`, so `causalSafety` takes not five, but six arguments:

- the process vector clock `procVc`,
- the messages `m1` and `m2`;
- a premise that `m2` is deliverable at a process with vector clock `procVC`;
- a premise that `m1` causally precedes `m2`;
- and a vector clock index `k`, of type **`PID`**.

The proof has two cases: either `k` is the index of the vector clock entry corresponding to `m2`'s sender (lines 14-16 of Listing 2), or it is the index of some other vector clock entry (lines 12-13). 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 line 17 is not necessary to complete the proof, but included for aesthetic reasons.

We first consider the case in which `k` is *not* the sender of `m2` (lines 12-13 of Listing 2). Since `m1` $\rightarrow$ `m2`, we know that for all `k`, $VC(\texttt{m1})[\texttt{k}] \leq VC(\texttt{m2})[\texttt{k}]$. Furthermore, we know that `m2` is deliverable at a process with vector clock `procVc`, so we know from the second case of the definition of `deliverable` that $VC(\texttt{m2})[\texttt{k}] \leq \texttt{procVC}[\texttt{k}]$. Liquid Haskell puts these two facts together to conclude that $VC(\texttt{m1})[\texttt{k}] \leq \texttt{procVC}[\texttt{k}]$, as required by **`DeliveredProp`**.

Next we consider the case in which `k` *is* `m2`'s sender. Since `m1` $\rightarrow$ `m2`, we have from `vcAxiom` that $VC(\texttt{m1})[\texttt{k}] < VC(\texttt{m2})[\texttt{k}]$. Since `m2` is deliverable at a process with vector clock `procVc`, we know from the first case of the definition of `deliverable` that $VC(\texttt{m2})[\texttt{k}] = \texttt{procVC}[\texttt{k}] + 1$. Putting these facts together, Liquid Haskell can conclude that $VC(\texttt{m1})[\texttt{k}] < \texttt{procVC}[\texttt{k}] + 1$, and therefore that $VC(\texttt{m1})[\texttt{k}] \leq \texttt{procVC}[\texttt{k}]$, again as required by **`DeliveredProp`**.

In both cases of the proof, a call to `deliverableImpliesDeliverableProp` converts the provided proof of deliverability of `m2` to a function of type **`DeliverableProp`**, that is, a function that takes a particular vector clock index `k` and returns a proof that the deliverability condition holds for that particular index. Structuring our proof around functions that take an index `k` into a vector clock helps us precisely express in our proof the reason *why* Birman et al.'s causal broadcast protocol works, since deliverability is decided on the basis of comparing vector clocks at each index. It also means that the only place we need to use induction is in the proof of `iterImpliesForall`. This way of expressing `causalSafety` was inspired by Agda's dependent function types. However, the Liquid Haskell proof takes advantage of the underlying SMT solver's automated reasoning about linear arithmetic, making the proof shorter and less tedious than a corresponding Agda proof might be. For example, in the last step of the above proof, Liquid Haskell can automatically prove that $VC(\texttt{m1})[\texttt{k}] \leq \texttt{procVC}[\texttt{k}]$, given $VC(\texttt{m1})[\texttt{k}] < \texttt{procVC}[\texttt{k}] + 1$, without us having to provide any additional information to the solver. Finally, thanks to Liquid Haskell's refinement reflection, we can state and prove extrinsic properties directly about `deliverable`, which is a part of our running implementation, instead of having to separately write a logical specification for `deliverable`'s behavior and then relate it to the implementation.

5 FURTHER IMPLEMENTATION COMPONENTS

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 `deliverable`. In this section we give an overview of the remaining pieces of our implementation (provided by the **`DelayQueue`**, **`Process`**, and **`Protocol`** modules shown in Figure 2), including the delay queue where each process buffers messages that are not yet deliverable, additional state that processes

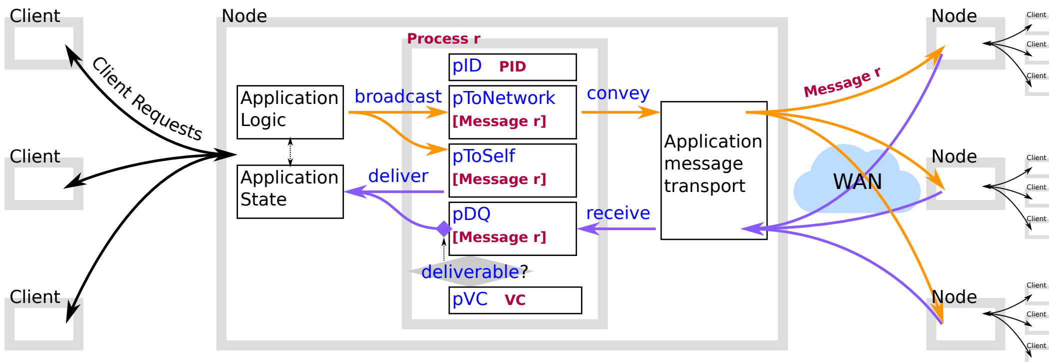

Fig. 5. Example architecture for a distributed application that uses our causal broadcast library.

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 4 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 4, 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 of `dqList` expresses the fact that processes never buffer messages sent from themselves, since a self-sent message is always immediately deliverable.

The `dqEnqueue` operation (lines 8-12 of Listing 4) takes a message m_{new} and inserts it into `dqList` *after* any existing messages $m_{existing}$ such that $VC(m_{existing}) \leq VC(m_{new})$, using helper functions `insertAfterBy` and `mSentLessEqual` (not shown in the listing). The delay queue is therefore

```
1  data DelayQueue [dqSize] r = DelayQueue
2      { dqSelf :: PID
3      , dqList :: [{ m : Message r | mSender m /= dqSelf }]
4      }
5
6  type DQ r PID = { dq : DelayQueue r | dqSelf dq == PID }
7
8  dqEnqueue
9      :: m:Message r
10     -> { xs : DelayQueue r | mSender m /= dqSelf xs }
11     -> { ys : DelayQueue r | dqSize xs + 1 == dqSize ys && dqSelf xs == dqSelf ys }
12 dqEnqueue m dq = dq{dqList=insertAfterBy mSentLessEqual m (dqList dq)}
13
14 dqDequeue
15     :: procVc : VC
16     -> xs : DelayQueue r
17     -> Maybe
18         ( { ys : DelayQueue r | dqSize xs - 1 == dqSize ys && dqSelf xs == dqSelf ys }
19         , { m : Message r | deliverable m procVc }
20     )
21 dqDequeue _ DelayQueue{dqList=[]} = Nothing
22 dqDequeue procVc dq@DelayQueue{dqList=x:xs}
23     | deliverable x procVc = Just (dq{dqList=xs}, x)
24     | otherwise = case dqDequeue procVc dq{dqList=xs} of
25         Just (DelayQueue{dqList=ys}, y) -> Just (dq{dqList=x:ys}, y)
26         Nothing -> Nothing
```

Listing 4. The **DelayQueue** data type, a convenience alias **DQ** r **PID**, and the dqEnqueue and dqDequeue operations on delay queues. dqEnqueue inserts messages into the delay queue such that the queue is sorted by messages' vector clocks. dqDequeue's type guarantees that the returned message is deliverable, and **DelayQueue**'s type prevents inserting messages sent from the current process into the 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.[7] The refinement type of dqEnqueue ensures that the queue grows by one when a message is enqueued, and that we do not enqueue self-sent messages.

The dqDequeue operation (lines 14-26 of Listing 4) takes the process vector clock `procVC` as an argument and tests each message in `dqList` in turn with the `deliverable` predicate from Listing 1. dqDequeue returns a pair of an updated delay queue and the first message `m` for which `deliverable m procVC` is satisfied, if there is one. If there are no deliverable messages in `dqList`, dqDequeue returns **Nothing**. The refinement type of dqDequeue ensures that the queue shrinks by one when a message is dequeued, and that the dequeued message is indeed deliverable.

5.2 Process State

Our **Process** module exports a **Process** data type, defined as follows:

```
data Process r = Process
    { pID :: PID
    , pVC :: VC
    , pDQ :: DQ r pID
```

[7] We considered using refinement types to express this sortedness invariant, but we had trouble doing so cleanly in Liquid Haskell, because `insertAfterBy` is recursive and the invariant needs to be preserved through a recursive call, resulting in undesirable code duplication.

```
  , pToSelf :: FIFO ({ m : Message r | pID == mSender m })
  , pToNetwork :: FIFO ({ m : Message r | pID == mSender m })
  }
```

In addition to its delay queue, a process must keep track of its own process ID, its own vector clock, and two additional process-local message buffers, `pToSelf` and `pToNetwork`. The `pToSelf` buffer contains self-sent messages, and the `pToNetwork` buffer contains outgoing messages.

We specify the type of a process's delay queue with the type synonym from line 6 of Listing 4, `DQ r pID`, which references the record field value `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 `pToSelf` and `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 `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: `pNew`, `broadcast`, `receive`, `deliver`, and `convey`. Listing 5 shows the implementations of the external API functions. We summarize their behavior as follows:

- The `pNew` function (lines 1-8 of Listing 5) creates and returns a fresh `Process` with the specified `PID` and `ProcCount`, an empty delay queue, and empty `pToSelf` and `pToNetwork` buffers.
- The `broadcast` function (lines 10-19) takes a raw message of type `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 `Message r`. It also pushes the wrapped message onto both the `pToSelf` and `pToNetwork` buffers.
- The `receive` function (lines 21-24) takes a message and adds it to the delay queue if it is from a different sender. `receive` ignores self-sent messages, since `broadcast` already handles them by adding them to the `pToSelf` buffer.
- The `deliver` function (lines 26-41) pops the first message from the `pToSelf` buffer, if any, or dequeues the first deliverable message from the delay queue, if any, and otherwise returns `Nothing`. If a message `m` is returned, `deliver` implements the delivery side of the vector clock protocol of Section 3.1 by updating the process vector clock to the pointwise maximum of VC(`m`) and the current process vector clock. If a message from another process is returned, it is verified to be deliverable.
- The `convey` function (lines 43-47) 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 6 simulates the client behavior from Figure 4 (right) to demonstrate how our protocol implementation reorders messages. Lines 1-3 of Listing 6 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 `broadcast` followed by `convey`, 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

```
pNew :: PID -> ProcCount -> Process r
pNew pid pCount = Process
    { pID = pid
    , pVC = vcNew pCount
    , pDQ = dqNew pid
    , pToSelf = fNew
    , pToNetwork = fNew
    }

broadcast :: r -> Process r -> Process r
broadcast r p
    = let
        vc = vcTick (pID p) (pVC p)
        m = Message{mSender=pID p, mSent=vc, mRaw=r}
    in p
        { pVC = vc
        , pToNetwork = fPush (pToNetwork p) m
        , pToSelf = fPush (pToSelf p) m
        }

receive :: Message r -> Process r -> Process r
receive m p
    | mSender m == pID p = p -- Messages sent by this process are ignored.
    | otherwise = p{pDQ=dqEnqueue m (pDQ p)}

deliver
    :: p : Process r
    -> ( Process r
       , Maybe ({ m:Message r | mSender m /= pID p => deliverable m (pVC p)})
       )
deliver p = case fPop (pToSelf p) of
    Just (m, toSelf) ->
        ( p{pToSelf=toSelf, pVC=vcCombine (pVC p) (mSent m)}
        , Just m
        )
    Nothing -> case dqDequeue (pVC p) (pDQ p) of
        Just (dq, m) ->
            ( p{pDQ=dq, pVC=vcCombine (pVC p) (mSent m)}
            , Just m
            )
        Nothing -> (p, Nothing)

convey :: Process r -> (Process r, [Message r])
convey p =
    ( p{pToNetwork=fNew}
    , fList (pToNetwork p)
    )
```

Listing 5. Implementation of the external API provided by our causal broadcast implementation. The vcNew operation returns a new **VC** initialized with zeroes, and dqNew creates a new delay queue. vcTick increments the specified index in a **VC**, and vcCombine takes the pointwise maximum of **VC**s. Operations on the **FIFO** data type (fNew, fPush, fPop, fList) have the expected semantics.

```
1  alice <- newIORef (pNew 0 3 :: Process String)
2  bob   <- newIORef (pNew 1 3 :: Process String)
3  carol <- newIORef (pNew 2 3 :: Process String)
4
5  -- Alice sends 'lost' and their VC increments to [1,0,0].
6  modifyIORef alice $ broadcast "I lost my wallet..."
7  aliceBcastLost <- atomicModifyIORef alice convey
8
9  -- Alice sends 'found' and their VC increments to [2,0,0].
10 modifyIORef alice $ broadcast "Found it!"
11 aliceBcastFound <- atomicModifyIORef alice convey
12
13 -- Bob receives both 'lost' and 'found' and delivers them in causal order,
14 -- updating their VC to [2,0,0].
15 modifyIORef bob $ \p -> foldr receive p (aliceBcastLost ++ aliceBcastFound)
16 Just (Message{mRaw="I lost my wallet..."}) <- atomicModifyIORef bob deliver
17 Just (Message{mRaw="Found it!"}) <- atomicModifyIORef bob deliver
18
19 -- Carol receives 'lost' and delivers it, updating their VC to [1,0,0].
20 modifyIORef carol $ \p -> foldr receive p aliceBcastLost
21 Just (Message{mRaw="I lost my wallet..."}) <- atomicModifyIORef carol deliver
22
23 -- Bob sends 'glad' and their VC increments to [2,1,0].
24 modifyIORef bob $ broadcast "Glad to hear it!"
25 bobBcastGlad <- atomicModifyIORef bob convey
26
27 -- Carol receives 'glad' and delays it because it depends on 'found'.
28 modifyIORef carol $ \p -> foldr receive p bobBcastGlad
29 Nothing <- atomicModifyIORef carol deliver
30
31 -- Carol receives 'found' and delivers it, updating their VC to [2,0,0].
32 modifyIORef carol $ \p -> foldr receive p aliceBcastFound
33 Just (Message{mRaw="Found it!"}) <- atomicModifyIORef carol deliver
34
35 -- Carol delivers 'glad', updating their VC to [2,1,0].
36 Just (Message{mRaw="Glad to hear it!"}) <- atomicModifyIORef carol deliver
```

Listing 6. An example execution that observes causal delivery.

(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 `broadcast` and `receive` functions are both ways to inject message data into a local `Process r`, for different purposes, and the `convey` and `deliver` functions are both ways to extract message data from a `Process r`. The `broadcast` and `convey` functions both have the purpose of sending information outward to the causal broadcast cluster, and the `receive` and `deliver` functions both have the purpose of bringing information inward from the cluster.

6 CASE STUDY: A DISTRIBUTED IN-MEMORY KEY-VALUE STORE

Using our causal broadcast API of Section 5, we implemented a geo-distributed, replicated in-memory key-value store (KVS) application. Our application uses the architectural pattern depicted by Figure 5. The KVS is implemented by an HTTP service running on each node that fields requests from clients,[8] and simultaneously broadcasts state updates to a cluster of other nodes, 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].

6.1 Design and Implementation

We implemented the KVS HTTP service 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 `DELETE`, the endpoint generates a value of type `KvCommand` to update the causal broadcast cluster, where `KVCommand` is defined as follows:

```
type Key = String
type Val = Aeson.Value
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:

```
type NodeState = Process KvCommand
type KvState = Map Key Val
```

The `PUT` and `DELETE` endpoints call `broadcast` on the `NodeState` and the `KvCommand` value, waking up two background threads waiting on `stm` references. The *send mail* background thread uses `convey` 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 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 `KvCommand`s 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.

6.2 Deployment and Evaluation

We deployed a three-node causal broadcast cluster, geo-distributed across three AWS regions (one node each in *us-west-1*, *us-west-2*, and *us-east-1*) and 30 client nodes with ten clients assigned to each AWS region. All the nodes were AWS EC2 `t3.micro` instances with 2 vCPUs at 2.5 GHz and 1 GiB of memory. The average inter-region latencies were 75ms between us-east-1 and us-west-2, 61ms between us-east-1 and us-west-1, and 20ms between us-west-1 and us-west-2. Each of the three nodes in the cluster ran an instance of our KVS application compiled with GHC 8.10.2.

[8]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].

We conducted a simple experiment in which each of the 30 clients made 10,000 `curl` requests to the KVS replica in their same region (for a total of 300,000 client requests), uniformly distributed over `GET`, `PUT`, and `DELETE` requests. We ensured that there were key collisions by drawing keys from among the lowercase ASCII characters. For `PUT` requests, we used randomly generated JSON data for values. Using this experimental setup, we sought empirical answers to two questions:

- *How fast does the KVS process requests?* Two-thirds of the 300,000 requests generated by clients were `PUT` and `DELETE` requests, and each results in a broadcast to the cluster, generating an additional 400,000 requests exchanged among nodes. The KVS replicas handled all requests in about 5 minutes with about 0.05 CPU utilization, reflecting that the application was network-bound rather than CPU-bound. As a static verification approach, Liquid Haskell itself imposes no running time overhead compared to vanilla Haskell.
- *How often are messages queued for later delivery?* The delay queues at each replica were usually empty. About 5000 message deliveries occurred when there was one other message in the delay queue, and another 1000 deliveries occurred with more than one delayed message. That is, about 1 in 120 messages was received before its causal dependencies were delivered, motivating the need for causal broadcast. A larger or more widely geo-distributed cluster might result in more messages being received out of causal order.

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, which considerably simplifies integration of the verified code with unverified application code, with no need for an extraction step. 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 [Ahamad 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 extract and deploy their verified-convergent CRDTs atop our verified-safe causal broadcast protocol to get an "end-to-end" convergence 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 than Gomes et al.'s CRDTs; 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, 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 hypothesize that our library would lessen the need for such ad hoc mechanisms and help simplify the implementation and verification of CRDTs.

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 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 CONCLUSION AND FUTURE WORK

Causal message broadcast is a widely used building block of distributed applications, motivating the need for practically usable verified implementations. We have presented a verified executable causal broadcast library implemented using Liquid Haskell, which enables proofs to be carried out directly in Haskell, with no need for subsequent transformation or extraction steps. We define a correctness condition, causal safety, that ensures that every execution of our protocol observes causal delivery of messages. We mechanically verify the causal safety of our implementation. The resulting verified library is of immediate use in real distributed systems written in Haskell. We evaluate its utility with a case study application of a distributed key-value store.

We acknowledge the gap between what our safety property ensures, and what one might like to claim is true of a running causal broadcast 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 that is in scope for this work pertains only to the behavior of `deliverable`, an internal aspect of our implementation. In future work, we are also interested in verifying the correctness of the external API we provide to applications, and in verifying application-level properties, such as causal consistency of our key-value store and convergence of CRDTs that depend on causal broadcast.

REFERENCES

Arup Acharya and B.R. Badrinath. 1992. Recording distributed snapshots based on causal order of message delivery. *Inform. Process. Lett.* 44, 6 (1992), 317 – 321. https://doi.org/10.1016/0020-0190(92)90107-7

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

Sridhar Alagar and S. Venkatesan. 1994. An optimal algorithm for distributed snapshots with causal message ordering. *Inform. Process. Lett.* 50, 6 (1994), 311 – 316. https://doi.org/10.1016/0020-0190(94)00055-7

Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

K. Birman and T. Joseph. 1987a. Exploiting Virtual Synchrony in Distributed Systems. *SIGOPS Oper. Syst. Rev.* 21, 5 (Nov. 1987), 123–138. https://doi.org/10.1145/37499.37515

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

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

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

Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On Verifying Causal Consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 626–638. https://doi.org/10.1145/3009837.3009888

K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66.

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133933

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Ajay D Kshemkalyani, Michel Raynal, and Mukesh Singhal. 1995. An introduction to snapshot algorithms in distributed computing. *Distributed systems engineering* 2, 4 (1995), 224.

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

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., USA.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-Value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 357–370. https://doi.org/10.1145/2837614.2837622

Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 216 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428284

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.

1145/2043556.2043593

P. Mahajan, L. Alvisi, and M. Dahlin. 2011. *Consistency, Availability, Convergence.* Technical Report TR-11-22. Computer Science Department, University of Texas at Austin.

Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms.* North-Holland, 215–226.

Kartik Nagar and Suresh Jagannathan. 2019. Automated parameterized verification of CRDTs. In *International Conference on Computer Aided Verification.* Springer, 459–477.

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP'08).* Springer-Verlag, Berlin, Heidelberg, 230–266.

Jon Postel. 1981. *Transmission Control Protocol.* STD 7. RFC Editor. http://www.rfc-editor.org/rfc/rfc793.txt http://www.rfc-editor.org/rfc/rfc793.txt.

Michel Raynal, André Schiper, and Sam Toueg. 1991. The causal ordering abstraction and a simple way to implement it. *Inform. Process. Lett.* 39, 6 (1991), 343–350. https://doi.org/10.1016/0020-0190(91)90008-6

Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel Distrib. Comput.* 71, 3 (March 2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08).* Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

J. Rushby, S. Owre, and N. Shankar. 1998. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (1998), 709–720. https://doi.org/10.1109/32.713327

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, Berlin, Heidelberg, 219–232.

N. Schiper, V. Rahli, R. Van Renesse, M. Bickford, and R. L. Constable. 2014. Developing Correctly Replicated Databases Using Formal Tools. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 395–406. https://doi.org/10.1109/DSN.2014.45

Frank B Schmuck. 1988. *The use of efficient broadcast protocols in asynchronous distributed systems.* Ph.D. Dissertation.

Reinhard Schwarz and Friedemann Mattern. 1994. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing* 7, 3 (1994), 149–174.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11).* Springer-Verlag, Berlin, Heidelberg, 386–400.

KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).* Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

Robbert van Renesse. 1993. Causal Controversy at Le Mont St.-Michel. *SIGOPS Oper. Syst. Rev.* 27, 2 (April 1993), 44–53. https://doi.org/10.1145/155848.155857

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018).* Association for Computing Machinery, New York, NY, USA, 132–144. https://doi.org/10.1145/3242744.3242756

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14).* Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1, Article 19 (June 2016), 34 pages. https://doi.org/10.1145/2926965

Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. 2008. The Isabelle Framework. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–38.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732