Chapar: Certified Causally Consistent Distributed Key-Value Stores

Mohsen Lesani Christian J. Bell Adam Chlipala
Massachusetts Institute of Technology, USA
{lesani, cjbell, adamc}@mit.edu

Artifact
Consistent * Complete * Well Documented * Easy to Reuse *
POPL * AEC * Evaluated

Abstract

Today's Internet services are often expected to stay available and render high responsiveness even in the face of site crashes and network partitions. Theoretical results state that causal consistency is one of the strongest consistency guarantees that is possible under these requirements, and many practical systems provide causally consistent key-value stores. In this paper, we present a framework called Chapar for modular verification of causal consistency for replicated key-value store implementations and their client programs. Specifically, we formulate separate correctness conditions for key-value store implementations and for their clients. The interface between the two is a novel operational semantics for causal consistency. We have verified the causal consistency of two key-value store implementations from the literature using a novel proof technique. We have also implemented a simple automatic model checker for the correctness of client programs. The two independently verified results for the implementations and clients can be composed to conclude the correctness of any of the programs when executed with any of the implementations. We have developed and checked our framework in Coq, extracted it to OCaml, and built executable stores.

Categories and Subject Descriptors C.2.2 [*Computer Communication Networks*]: Network Protocols—Verification; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness Proofs

General Terms Algorithms, Reliability, Verification

Keywords causal consistency, theorem proving, verification

1. Introduction

Modern Internet servers rely crucially on distributed algorithms for performance scaling and availability. Services should stay *available* even in the face of site crashes or network partitions. In addition, most services are expected to exhibit high responsiveness [21]. Hence, modern data stores are *replicated* across continents. During the downtime of a replica, other replicas can keep the service available, and the locality of replicas enhances responsiveness.

Program 1 (p_1): Uploading a photo and posting a status

```
0 →                                                    Alice
    $put(Pic, ☺)$;                           ▷ uploads a new photo
    $put(Post, ☁)$                           ▷ announces it to her friends
1 →                                                    Bob
    $post \leftarrow get(Post)$;             ▷ checks Alice's post
    $photo \leftarrow get(Pic)$;             ▷ then loads her photo
    $assert(post = ☁ \Rightarrow photo \neq \bot)$
```

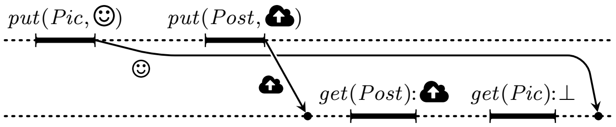

Figure 1. Inconsistent trace of Photo-Upload example

On the flip side, maintaining strong consistency across replicas [30] can limit parallelism [35] and availability. When availability is a must, the CAP theorem [19] formulates a fundamental trade-off between strong consistency and partition tolerance, and PACELC [3] formulates a trade-off between strong consistency and latency [5]. In reaction to these constraints, modern storage systems including Amazon's Dynamo [17], Facebook's Cassandra [27], Yahoo's PNUTS [16], LinkedIn's Voldemort [1], and memcached [2] have adopted relaxed notions of consistency that are collectively called *eventual consistency* [48]. The main guarantee that eventually consistent stores provide is that if clients stop issuing updates, then the replicas will converge to the same state. Researchers [13, 44, 46] have proposed eventually consistent algorithms for common datatypes like registers, counters, and finite sets. Recent work [12, 14, 54] has formalized and verified the eventual-consistency condition for these algorithms.

Weaker consistency is a double-edged sword. It can lead to more efficient and fault-tolerant algorithms, but at the same time it exposes clients to less consistent data. Programming with weak consistency is challenging and error-prone. As an example, consider Program 1, which shows two client routines (0 for Alice and 1 for Bob) running concurrently. An execution of the program with an eventually consistent store is shown in Figure 1. Alice uploads a photo of herself ☺ and then posts a message that she has uploaded a photo ☁. Bob reads Alice's post announcing the upload. He attempts to see the photo but only sees the default value. The message containing the photo arrives late. The post is issued after the photo is uploaded in Alice's node. We call this a *node-order dependency* from the post to the upload. If Bob can see the

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837622

Program 2 (p_2): The lost and found ring

```
0 →                                                    Alice
    $put(Alice, ❓)$;                       ▷ "I've lost my ring"
    $put(Alice, ☺)$                                ▷ "Found it!"
1 →                                                      Bob
    $post \leftarrow get(Alice)$;
    if $post = ☺$ then
        $put(Bob, 👍)$                      ▷ "Glad to hear it!"
2 →                                                    Carol
    $post \leftarrow get(Bob)$;
    $post' \leftarrow get(Alice)$;
    $assert\ (post = 👍 \Rightarrow post' \neq ❓)$
```

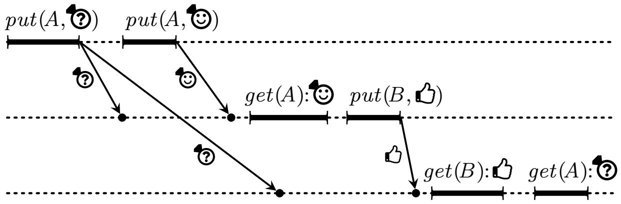

Figure 2. Inconsistent trace of Lost-Ring example

post, he is expected not to miss the photo. Bob's code includes an assertion formalizing his expectation that the presence of a post implies the presence of a photo. Unfortunately, some natural implementations of a key-value store will *not* respect such properties. For instance, eventual consistency does not guarantee this invariant in *intermediate* execution states, just quiescent states.

Program 2 and Figure 2 depict another scenario [31], where Alice posts that she has lost her ring ❓, but then she finds it and posts that all is well ☺. Bob sees Alice's second post. We say that there is a *gets-from* dependency from Alice's second put to Bob's get operation. Bob then responds with "Glad to hear it!" 👍. Bob's put operation is node-order dependent on his get operation. Thus, Bob's put operation is *transitively* dependent on Alice's second put operation. Carol is an observer of this exchange. If she sees Bob's post, she should not miss Alice's second post. However, a store that does not respect dependencies may allow for an inconsistent execution, where Carol could mistakenly think that Bob is glad to hear that Alice has lost her ring.

Therefore, stronger notions of consistency that can still be provided in the face of partitions are desirable. *Causal consistency* [4, 9, 28, 42] is shown [34] to be one of the strongest consistency notions compatible with high availability. Thus, many pioneering systems such as ISIS [9], causal memory [4], lazy replication [26], Bayou [37, 47], and PRACTI [7], plus recent systems such as COPS and Eiger [31, 32] and Bolt-On [6], provide causal consistency. In addition to social-network applications, many others, including the familiar example of electronic mail, can benefit from causal consistency.

Causal consistency ensures that replicas respect the *causal dependencies* between the operations. In other words, if an update is visible at a replica, all the updates that it is dependent on should be also visible at that replica. For example, in the scenario above, Alice's post is dependent on her upload of the photo. Thus, the update for the post should be applied to Bob's replica only when the update for the photo is already applied. Therefore, when the post is retrieved from the replica, the photo is already available in the replica.

Concurrent and distributed algorithms are challenging to design and understand, and distributed-system bugs [20, 52] are notoriously hard to find and reproduce. Further, the integrity of the data store and the consistency of the data that clients observe are dependent on the correctness of the causally consistent algorithm that coordinates between the replicas. Therefore, *precise specification* of causal consistency and *verification techniques* that check the compliance of key-value store implementations with the specification enhance the reliability of applications that are built using these data stores. In addition, causal consistency provides weaker guarantees than serializability to clients. Thus, the client programs are exposed to less consistent data and are prone to more bugs. Therefore, *automatic checkers* are a useful aid for client programmers to verify that their programs preserve their application invariants if executed with causally consistent stores. Although there have been recent efforts on verification of eventual consistency, to the best of our knowledge, our work presented here is the first to address causal consistency.

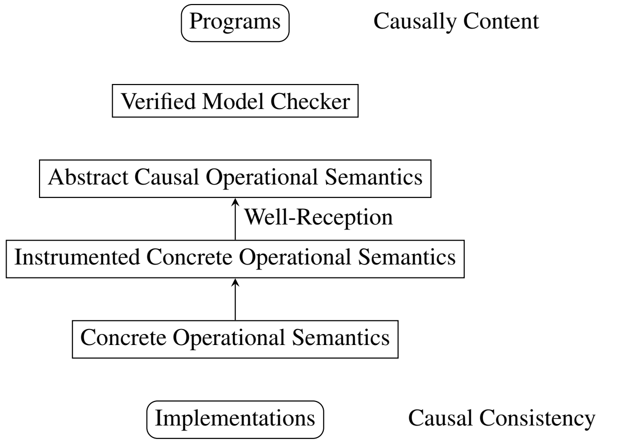
Programs
Causally Content
Verified Model Checker
Abstract Causal Operational Semantics
Well-Reception
Instrumented Concrete Operational Semantics
Concrete Operational Semantics
Implementations
Causal Consistency

Figure 3. Overview of Chapar Framework

We have developed a verification framework called Chapar for causally consistent key-value stores using the Coq proof assistant. Figure 3 shows an overview of the framework. We explain each part in turn.

Previous work [4, 13, 42] has presented denotational definitions of causal consistency for execution histories. In this work, we present an *abstract operational semantics* for causal consistency. The semantics defines all the causally consistent executions of a *program* on the map interface independently of any concrete implementation of the interface. It serves as a layer that separates the verification of concrete implementations from the verification of client programs. Implementations are verified to comply with, and clients are verified on top of, the abstract semantics. (For brevity, throughout this paper we adopt the convention that *implementations* refers to implementations of key-value stores, while *client code* refers to programs running on top, even though those programs are reasonably considered as "implementations" with standard terminology.)

The abstract semantics provides a convenient execution model to build automatic model checkers. We refer to programs that avoid assertion failures when executed with the abstract semantics as *causally content*. We present a simple automatic model checker that can verify that a closed program is causally content.

We present a common *interface for the key-value store implementations*. The interface captures the type of the node state and broadcast updates; the signatures of the put, get, and update operations; and a precondition guard on updates. We present a *concrete operational semantics for implementations* of this interface, parametric in a choice of implementation. We define that an implementation is *causally consistent* if and only if the concrete operational se-

n	:	NId	Node identifier
k	:	K	Key
v	:	V	Value
x			Variable
s	:	S	Statement
	$::=$	$put(k, v);\ s$	
	$\mid$	$x \leftarrow get(k);\ s$	
	$\mid$	$skip$	
	$\mid$	$assertfail$	
N	:	$\mathbb{P}(NId)$	Node identifier set
p	:	$N \rightarrow S$	Program
$\mathbb{P}(S)$	$\triangleq$	The powerset of the set S	

Figure 4. Program

mantics instantiated with the implementation is a refinement of the abstract operational semantics.

We present a novel *proof technique* for causal consistency of key-value store implementations. We present a condition called *well-reception* for implementations and prove that it is a *sufficient condition for causal consistency*. The well-reception condition is defined based on an *instrumented concrete operational semantics* that can track dependencies by instrumenting values with the unique identifiers of the put operations that originated them. The proof of sufficiency of the well-reception condition factors out a significant part of the proof of causal consistency for standard implementations. Thus, the well-reception proof technique requires a small number of specific obligations to be proved for each implementation.

We have modeled two key-value store implementations from the literature [4, 32] (and a variant of the first one) as implementations of the common interface and *verified their causal consistency* using the well-reception proof technique. All the definitions and lemmas presented in the paper are formalized and machine-checked in Coq. Our Coq and OCaml code are available online at:

```
https://github.com/mit-plv/chapar
```

We have extracted the verified implementations to OCaml and built executable key-value stores. We compare the performance results of the resulting stores.

In summary, this paper presents the following.

- An abstract causal operational semantics (§ 3)
- An automatic program verifier for client programs when run with the abstract semantics (§ 6)
- A programming interface and a concrete operational semantics for distributed key-value store implementations (§ 2)
- A proof technique for causal consistency of key-value store implementations (§ 4)
- A verification framework and the proof of causal consistency of two key-value store implementations from the literature (§ 5)
- The performance results of the resulting key-value stores (§ 7)

2. Key-Value Stores and Clients, Formally

In this section, we define an interface for distributed key-value store implementations. (In § 5, we present two implementations of this interface.) Then, we define an operational semantics parameterized on this interface. The operational semantics models the executions of the asynchronous distributed system of replicated stores. To motivate the developments of the following sections, we finish this section with an example of a final theorem that we have proved in Coq, verifying a system that includes a client program and a key-value store implementation. The following sections will fill in more details on how to do such proofs modularly.

$$\begin{array}{l}
\mathbb{I} = (\mathsf{State}, \mathsf{Update}, \mathsf{init}, \mathsf{put}, \mathsf{get}, \mathsf{guard}, \mathsf{update}) \\
\mathsf{State}\colon (V\colon Type) \rightarrow \\
\quad Type \\
\mathsf{init}\colon (V\colon Type) \rightarrow \\
\quad \mathsf{State}(V) \\
\mathsf{get}\colon (V\colon Type) \rightarrow \\
\quad (N, \mathsf{State}(V), K) \rightarrow (V \times \mathsf{State}(V)) \\
\mathsf{Update}\colon (V\colon Type) \rightarrow \\
\quad Type \\
\mathsf{put}\colon (V\colon Type) \rightarrow \\
\quad (N, \mathsf{State}(V), K, V) \rightarrow (\mathsf{State}(V) \times \mathsf{Update}(V)) \\
\mathsf{guard}\colon (V\colon Type) \rightarrow \\
\quad (N, \mathsf{State}(V), K, V, \mathsf{Update}(V)) \rightarrow Bool \\
\mathsf{update}\colon (V\colon Type) \rightarrow \\
\quad (N, \mathsf{State}(V), K, V, \mathsf{Update}(V)) \rightarrow \mathsf{State}(V)
\end{array}$$

Figure 5. Key-Value Store Interface

Let us define the client programs first. Figure 4 defines the syntax of programs. We use n to denote a node identifier, k to denote a key, v to denote a value, x to denote a variable, and s to denote a statement. A statement is either put, get, $skip$, or $assertfail$. A put updates the mapping for the given key to the given value. A get returns the value that the given key is mapped to. The $skip$ statement terminates a sequence of statements. We omit trailing $skip$ statements for brevity. The $assertfail$ statement is used to represent invariant violations. An assert statement $assert(b); s$ is desugared to the statement **if** b **then** s **else** $assertfail$, where, thanks to a Coq encoding in the style of higher-order abstract syntax [38], we use Coq's base functional programming language, Gallina, to provide extra features like conditional branching. Note that after $assert$ is desugared, $assertfail$ always appears as the last statement. A program p is a map from a finite set of node identifiers N to statements, which run concurrently on the nodes with a shared map abstract data type. Programs 1 and 2 were examples of this style.

Figure 5 presents the signature of the functions that a key-value store implements. The definitions are parametric in terms of the type V of values that are stored in the store. The State function (i.e., type family) defines the state type that each node maintains. The init function returns the initial state of every node. The get function, given the identifier and the state of the current node, and the input key, returns the value for the key and the new state of the node. After the execution of a put operation by a node, an update is sent to every other node. The Update type family defines the type of update payloads. The put function, given the identifier and the state of the current node, and the input key and value, returns the new state of the node and the update payload. To preserve consistency conditions, the application of updates should be delayed until certain conditions hold. The algorithm designer can program these conditions in the guard function. The semantics that follows applies an update available from another node only if the guard is satisfied for the update at the current node. Given the identifier and the state of the current node, the key, the value, and the update payload, the guard function returns whether the update is allowed to be applied. The update function, given the same arguments, returns the new state resulting from applying the update.

Figure 6 presents the concrete operational semantics $\rightarrow_{\mathcal{C}(\mathbb{I})}$ for key-value store implementations of the interface defined above. It is a labeled transition system that is parametric on the implementation $\mathbb{I} = (\mathsf{State}, \mathsf{Update}, \mathsf{init}, \mathsf{put}, \mathsf{get}, \mathsf{guard}, \mathsf{update})$ and the type of values V. The type variable of the implementation state State, update payload Update, and operation functions is instantiated with the value type V. First, let us consider the states and labels of the transition system that are presented in the lower part of Figure 6. Worlds $W_{\mathcal{C}}$ are pairs of the host states h and the in-transit messages t. The host states h is a map from each node identifier to the pair

PUT

$$\frac{\begin{array}{c}\mathsf{put}(V,n,\sigma,k,v) \rightsquigarrow^* (\sigma',u)\\ t' = t \cup \{(n',k,v,u) \mid n' \in N \setminus \{n\}\}\end{array}}{\begin{array}{c}(h[n \mapsto (put(k,v);s,\sigma)],t)\\ \xrightarrow{n\triangleright\, put(k,v)}_{\mathcal{C}(\mathbb{I})}\\ \big(h[n \mapsto (s,\sigma')],t'\big)\end{array}}$$

GET

$$\frac{\mathsf{get}(V,n,\sigma,k) \rightsquigarrow^* (v,\sigma')}{\begin{array}{c}(h[n \mapsto (x \ \leftarrow \ get(k);s,\sigma)],t)\\ \xrightarrow{n\triangleright\, get(k):\ v}_{\mathcal{C}(\mathbb{I})}\\ \big(h[n \mapsto (s[x := v],\sigma')],t\big)\end{array}}$$

UPDATE

$$\frac{\begin{array}{c}\mathsf{guard}(V,n,\sigma,k,v,u) \rightsquigarrow^* true\\ \mathsf{update}(V,n,\sigma,k,v,u) \rightsquigarrow^* \sigma'\end{array}}{\begin{array}{c}(h[n \mapsto (s,\sigma)],t \cup \{(n,k,v,u)\})\\ \xrightarrow{n\triangleright\, update(k,v)}_{\mathcal{C}(\mathbb{I})}\\ \big(h[n \mapsto (s,\sigma')],t\big)\end{array}}$$

ASSERTFAIL

$$\frac{}{(h[n \mapsto (assertfail,\sigma)],t) \xrightarrow{assertfail}_{\mathcal{C}(\mathbb{I})} (h[n \mapsto (skip,\sigma)],t)}$$

$$\begin{array}{rcll}
W_{\mathcal{C}} & : & H \times T & \text{World}\\
h & : & H = N \rightarrow (S \times \mathsf{State}(V)) & \text{Hosts}\\
t & : & T = \mathbb{PM}(M) & \text{Transit}\\
m & : & M = N \times K \times V \times \mathsf{Update}(V) & \text{Message}\\
\sigma & : & \mathsf{State}(V) & \text{Alg State}\\
u & : & \mathsf{Update}(V) & \text{Alg Update}\\
l_{\mathcal{C}} & ::= & n \triangleright put(k,v) & \text{Label}\\
 & \mid & n \triangleright get(k)\colon v & \\
 & \mid & n \triangleright update(k,v) & \\
 & \mid & assertfail & \\
h_{\mathcal{C}} & ::= & {l_{\mathcal{C}}}^* & \text{History}
\end{array}$$

$$\begin{array}{rcl}
\mathbb{PM}(S) & \triangleq & \text{The multiset powerset of the set } S\\
W_{\mathcal{C}0}(p) & \triangleq & (\lambda n.(p(n),\mathsf{init}(V,v_0)),\emptyset)\\
v_0 & \triangleq & \text{The initial value}
\end{array}$$

Figure 6. Concrete Operational Semantics $\rightarrow_{\mathcal{C}(\mathbb{I})}$ for the implementation $\mathbb{I} = (\mathsf{State}, \mathsf{Update}, \mathsf{init}, \mathsf{put}, \mathsf{get}, \mathsf{guard}, \mathsf{update})$ and the value type V

of the statement of the node and the state of the implementation for the node. Implementation state is replicated across the nodes of the distributed system, and each node may be running some distinct statement. The in-transit messages t is a multiset of messages. Messages are generated only by put operations broadcasting new updates to all nodes. A message is the tuple of the receiver node identifier, the key, the value, and the update payload (which is specific to the implementation). The initial world for a program p is $W_{\mathcal{C}0}(p)$. A label $l_{\mathcal{C}}$ is either a put, get, update, or assertion failure. The first three labels encode the issuing node identifier and the corresponding key and value, where the symbol $\triangleright$ in labels is just a separator. A history $h_{\mathcal{C}}$ is a sequence of labels. We use $\rightsquigarrow^*$ to denote the big-step operational semantics of the metalanguage in which the implementation is programmed (in our case, Coq's Gallina).

The rule PUT executes a *put* statement with the key k and the value v. It executes the put function of the implementation on the current node identifier n and state σ and the given k and v to yield the new state σ' and the update u for other nodes. A message for every other node n', containing the key k, the value v, and the update payload u, is added to the in-transit messages t. The rule GET executes a *get* statement with the key k. It executes the get function of the implementation on the current node identifier n and state σ and the given k to yield the value v of k and the new state σ'. The rule UPDATE removes a message that is sent to the current node n from the set of in-transit messages and applies the update that it carries. The update of a message is applied only if the guard condition is satisfied for the current state of the node and the update. The update function of the implementation is executed on the current node identifier n and state σ, the key k, value v, and update payload u to yield the new state σ'. The rule ASSERTFAIL steps an *assertfail* statement. The components of the world remain unchanged, except for the statement of the current node n, which is changed to *skip*. Updating the statement of a node, whose assertion fails, to *skip* prevents further execution in the node.

The operational semantics is nondeterministic in the choice of the node that takes the step, and, if the step is an update, the message that is received. With this nondeterminism, messages can be delivered out of order. Further, the operational semantics *models executions with node crashes and message losses as well.* A crashed node is not chosen to take a step. Further, a node can be considered crashed for some period of time and then return and take steps. A message that is lost is never chosen to be received. Thus, messages of a node can appear to be lost for some nodes but be received and processed by other nodes.

In the next section, we will present an abstract operational semantics for causal consistency (§ 3). We will later present the key-value store implementation $\mathbb{I}_1$ from the literature and prove its correctness (§ 5). More precisely, we prove that the above concrete operational semantics instantiated with $\mathbb{I}_1$, $\rightarrow_{\mathcal{C}(\mathbb{I}_1)}$, is a refinement of the abstract operational semantics. We will also sketch our automatic verifier for client programs, which is able to verify the client program p_1 from § 1 against the abstract operational semantics (§ 6). More precisely, we prove that no execution of p_1 with the abstract operational semantics involves an assertion failure. With these two results in place, we derive the following final theorem showing that the execution of the program p_1 with the implementation $\mathbb{I}_1$ never results in an assertion failure. This theorem is proved with black-box composition of natural correctness results for p_1 and $\mathbb{I}_1$.

Theorem 1.

$$\forall h_{\mathcal{C}}, W_{\mathcal{C}} \colon W_{\mathcal{C}0}(p_1) \xrightarrow{h_{\mathcal{C}}}{}^{*}_{\mathcal{C}(\mathbb{I}_1)} W_{\mathcal{C}} \ \Rightarrow \ assertfail \notin h_{\mathcal{C}}$$

That is, no trace generated by the combined system will ever contain an assertion-failure label. We use the usual transitive-reflexive closure $\rightarrow^*$ of a labeled transition system $\rightarrow$, which concatenates labels of individual steps to form a history.

3. Operational Causal Consistency

In this section, we present a novel abstract operational semantics for causal consistency. Given a program, it defines causally consistent executions of the program. The semantics is abstract from any concrete implementation and does not involve message passing. It forms an effective interface between key-value store implementations and their client programs, letting us verify each side against a suitable abstraction of the other. For the client side, in this section, we define a safe program as a program that never reaches an assertion failure when executed with the abstract causal operational semantics. In § 6, we will present a model checker for the safety of client programs. For the implementation side, in the next section, we define a causally consistent implementation as an implementation that, combined with the concrete operational semantics, refines the abstract operational semantics.

The semantics uniquely identifies each put operation. For each, it tracks the identifiers of other put operations that it depends on. The update of a put operation *put* is applied to a node only if the

PUT

$$\frac{\begin{array}{c} u' = u +\!\!+ [(k, v, d)] \qquad a' = a[n \mapsto a(n) + 1] \\ m' = m[k \mapsto (v, n, |u'|, \emptyset)] \qquad d' = d \cup \{(n, |u'|)\} \end{array}}{\begin{array}{c} W_{\mathcal{A}}[n \mapsto (put(k, v); s, d, u, a, m)] \\ \xrightarrow{n,\ |u'|\ \rhd\ put(k,v)}_{\mathcal{A}} \\ W_{\mathcal{A}}[n \mapsto (s, d', u', a', m')] \end{array}}$$

GET

$$\frac{\begin{array}{c} m(k) = (v, n'', c'', d'') \\ d' = \begin{cases} d \cup \{(n'', c'')\} \cup d'' & \text{if } n'' \neq n_0 \\ d & \text{otherwise} \end{cases} \end{array}}{\begin{array}{c} W_{\mathcal{A}}[n \mapsto (x \ \leftarrow \ get(k); s, d, u, a, m)] \\ \xrightarrow{n'',\ c'',\ n\ \rhd\ get(k):\ v}_{\mathcal{A}} \\ W_{\mathcal{A}}[n \mapsto (s[x := v], d', u, a, m)] \end{array}}$$

UPDATE

$$\frac{\begin{array}{c} a_1(n_2) < |u_2| \qquad u_2[\![a_1(n_2)]\!] = (k, v, d) \\ \bigwedge_{(n,c)\in d} c \leq a_1(n) \qquad a_1' = a_1[n_2 \mapsto a_1(n_2) + 1] \\ m_1' = m_1[k \mapsto (v, n_2, a_1'(n_2), d)] \end{array}}{\begin{array}{c} W_{\mathcal{A}}[n_1 \mapsto (s_1, d_1, u_1, a_1, m_1)][n_2 \mapsto (s_2, d_2, u_2, a_2, m_2)] \\ \xrightarrow{n_2,\ a_1'(n_2),\ n_1\ \rhd\ update(k,v)}_{\mathcal{A}} \\ W_{\mathcal{A}}[n_1 \mapsto (s_1, d_1, u_1, a_1', m_1')][n_2 \mapsto (s_2, d_2, u_2, a_2, m_2)] \end{array}}$$

ASSERTFAIL

$$\frac{}{C[n \mapsto (assertfail, d, u, a, m)] \xrightarrow{assertfail}_{\mathcal{A}} C[n \mapsto (skip, d, u, a, m)]}$$

$$\begin{array}{rcll} c & : & C & \text{Clock number} \\ d & : & D = \mathbb{P}(N \times C) & \text{Dependencies} \\ u & : & U = (K \times V \times D)^* & \text{Updates} \\ a & : & A = N \to C & \text{Applied} \\ m & : & M = K \to (V \times N \times C \times D) & \text{Store} \\ W_{\mathcal{A}} & : & N \to (S \times D \times U \times A \times M) & \text{World} \\ l_{\mathcal{A}} & ::= & n, c \rhd put(k, v) & \text{Label} \\ & | & n', c', n \rhd get(k) \colon v & \\ & | & n', c', n \rhd update(k, v) & \\ & | & assertfail & \\ h_{\mathcal{A}} & ::= & {l_{\mathcal{A}}}^* & \text{History} \end{array}$$

$$W_{\mathcal{A}0}(p) \quad \triangleq \quad (\lambda n.(p(n), \emptyset, [], \lambda n.0, \lambda k.(v_0, n_0, 0, \emptyset))$$

$$\begin{array}{rcl} [] & \triangleq & \text{The empty list} \\ |ls| & \triangleq & \text{The length of a list } ls \\ +\!\!+ & \triangleq & \text{The append function for lists} \\ ls[\![i]\!] & \triangleq & \text{The } i\text{th element of the list } ls \\ n_0 \in NId \setminus N & \triangleq & \text{The dummy initial node} \end{array}$$

Figure 7. Abstract Causal Operational Semantics $\to_{\mathcal{A}}$

updates of the put operations that *put* is dependent on are already applied to that node.

A put operation *put* by a node n is dependent on the preceding get or put operations by n (node-order dependence). In addition, if a get operation *get* returns a value that the put operation *put* has put, then *get* is dependent on *put* (gets-from dependence). Further, if op is dependent on op' and op' is dependent on op'', then op is dependent on op'' (transitive dependence). We say that a dependency from an operation to another is a put dependency if the former is a put operation.

The semantics stores a set of put identifiers for each node, each update, and each stored value. The dependency information stored for a node is the set of identifiers of the past put operations of that node and the put operations that the past operations of that node are transitively dependent on.

Upon a put operation *put* by a node n, the update that is issued for other nodes includes the identifier of *put* so that the nodes that read the value in the future become dependent on *put*. It also stores the stored dependencies of n so that the transitive dependencies of *put* are propagated. In addition, upon a put operation *put* by a node n, the identifier of *put* is added to the stored dependencies of n so that future put operations by n become dependent on *put*. Once an update with the identifier of the originating put *put* and its stored dependencies d is received, it is checked that all the dependencies d are already received, and the new value together with the identifier of *put* and d are stored in the local store. Upon a get operation by a node n, the local store is read, and the identifier of *put* and d are added to the stored dependencies of n.

Figure 7 defines the abstract causal operational semantics. Let us first see the structure of the states and labels of the transition system. The definition of programs is unchanged from the previous section (Figure 4). A world $W_{\mathcal{A}}$ is a mapping from nodes to tuples with the following fields: the current statement s of the node, the set of put-operation identifiers d that the current node depends on, the list of updates u that the current node has issued for other nodes, the map a tracking which updates have been applied from other nodes, and the store m that contains the current values of the keys and their dependencies. Each node maintains a clock number that is advanced on each put operation. (The clock number of each node is stored in the mapping of its "applied" map a for the node itself.) We use c to denote a clock number. A put operation is uniquely identified by the pair of the issuing node identifier and clock number. The applied-updates field a is a map from node identifiers to clock numbers that keeps track of how far the current node has applied updates from each other node. Each update is the tuple of the key, value, and dependencies of the value. The store maps each key to the tuple of a value, the identifier of its originating put operation, and the dependencies of that put operation. The following paragraphs explain these transitions in more detail.

We use $l_{\mathcal{A}}$ to denote the put, get, update, and assertion-failure labels. A put label carries the identifier of the put operation (i.e. the pair of the issuing node identifier and clock number). A get or update label carries the identifier of the originating put operation before the identifier of the issuing node.

The rule PUT executes a *put* statement with the key k and the value v by the node n. The triple of k, v, and the current dependencies of the node d is appended at the end of the existing update list u to result in the new update list u'. Other nodes that apply this update will be dependent on the current put operation and d. Since the node n is executing a put operation, it is effectively applying an update from itself. Therefore, the mapping of the "applied" map a for n itself is incremented. The updates field u' has an entry for each put operation executed by n. Therefore, its length $|u'|$ is the number of executed put operations by n. Thus, the current clock number for n is $|u'|$, and the identifier of the current put operation is the pair of n and $|u'|$. The store m is updated to map input key k to a triple of the input value v, the identifier of the current put operation $(n, |u'|)$, and an empty dependency set. The dependency set is empty, as getting a value that is put by the node itself incurs no new dependencies. Finally, the identifier of the current put operation is added to the set of dependencies of the node so that any future put by the node will be dependent on the current put operation.

The rule GET executes a *get* statement with the key k by the node n. The mapping for k in the store m represents its value v, the identifier n'' of the node that put the value, the clock number c'' of the originating put operation, and the dependencies d'' of that put operation. If n'' is not the dummy node n_0 (stored initially for all keys), by reading the value, the current node becomes dependent on the originating put and its dependencies. Therefore, the identifier of the originating put (n'', c'') and its dependencies d'' are added to the dependencies of the node.

$$
\begin{array}{rcll}
l_{\mathcal{E}} & ::= & n \triangleright put(k, v) & \text{External Label} \\
 & \mid & n \triangleright get(k) \colon v & \\
 & \mid & assertfail & \\
h_{\mathcal{E}} & ::= & {l_{\mathcal{E}}}^{*} & \text{External History} \\
\mathsf{Ext}_{\mathcal{A}}(l_{\mathcal{A}}) & \triangleq & \begin{cases} n \triangleright put(k, v) & \text{if } l_{\mathcal{A}} = n, c \triangleright put(k, v) \\ n \triangleright get(k) \colon v & \text{if } l_{\mathcal{A}} = n', c', n \triangleright get(k) \colon v \\ \epsilon & \text{if } l_{\mathcal{A}} = n', c', n \triangleright update(k, v) \\ assertfail & \text{if } l_{\mathcal{A}} = assertfail \end{cases} & \\
\mathsf{Ext}_{\mathcal{A}}({l_{\mathcal{A}}}^{*}) & \triangleq & \mathsf{Ext}_{\mathcal{A}}(l_{\mathcal{A}})^{*} & \\
\mathsf{Ext}_{\mathcal{C}}(l_{\mathcal{C}}) & \triangleq & \begin{cases} n \triangleright put(k, v) & \text{if } l_{\mathcal{C}} = n \triangleright put(k, v) \\ n \triangleright get(k) \colon v & \text{if } l_{\mathcal{C}} = n \triangleright get(k) \colon v \\ \epsilon & \text{if } l_{\mathcal{C}} = n \triangleright update(k, v) \\ assertfail & \text{if } l_{\mathcal{C}} = assertfail \end{cases} & \\
\mathsf{Ext}_{\mathcal{C}}({l_{\mathcal{C}}}^{*}) & \triangleq & \mathsf{Ext}_{\mathcal{C}}(l_{\mathcal{C}})^{*} &
\end{array}
$$

Figure 8. External Labels

The rule UPDATE applies an update from node n_2 to node n_1. The number of updates that n_1 has received from n_2 is $a_1(n_2)$. If this number is less than the size of the updates u_2 of n_2, there are more updates from n_2 for n_1 . With this condition, the next update of u_2 is read. An update is the triple of a key k, a value v, and dependencies d. Before the update is applied to n_1, it is critical to check that all the dependencies in d are already applied in n_1, which is done by consulting the "applied" map a_1. A put with the identifier (n, c) is applied in n_1 if the clock number c is less than or equal to $a_1(n)$, the number of updates applied from n in n_1. As an update from node n_2 is applied, the mapping of a_1 for n_2 is advanced. The mapping of store m for k is updated to the tuple of v, the identifier of its originating put, and its dependencies. Note that the dependencies d_1 of node n_1 remain unchanged, as n_1 only updated the mapping for a key but did not *get* its value.

We now define the safety of client programs. The class of programs whose invariants can be maintained by the relatively weak consistency guarantees of causally consistent stores can benefit from the responsiveness and availability of these stores. We define causally content programs as the class of programs that do not experience assertion failures when executed with the abstract causal operational semantics.

Definition 1 (Cause-Content Program).
$\mathsf{CauseContent}(p) \triangleq \forall h_{\mathcal{A}}, W_{\mathcal{A}} :$

$$W_{\mathcal{A}0}(p) \xrightarrow{h_{\mathcal{A}}}{}^{*}_{\mathcal{A}} W_{\mathcal{A}} \;\Rightarrow\; assertfail \notin h_{\mathcal{A}}$$

A program is causally content if and only if every abstract causal execution of the program is assertion-fail-free. An execution is assertion-fail-free if and only if it includes no assertion-fail step.

An interesting property of our causal semantics is that it is directly executable. For medium-sized, terminating programs, it is tractable to consider all possible interleaved executions of the rules from Figure 7. This observation foreshadows the verified, automatic checker that we sketch in § 6. Our next step, however, is to turn to proof obligations on the other side of the client/implementation divide.

4. Verification of Implementations

In this section, we present a technique for the proof of causal consistency of key-value store implementations. We first define the causal consistency condition for a key-value store implementation and then introduce the *well-reception* condition, which is sufficient to establish causal consistency but also easier to reason about directly.

Before defining causal consistency, we define external labels and histories in Figure 8. We define external labels $l_{\mathcal{E}}$ and functions $\mathsf{Ext}_{\mathcal{A}}$ and $\mathsf{Ext}_{\mathcal{C}}$ that map the labels of the abstract and concrete operational semantics to external labels. External labels capture the external behavior of the operational semantics that is observable to client programs. Update transitions are not issued by the client program but are nondeterministically issued by the operational semantics. Therefore, update labels are not externally observable. On the other hand, assertion-failure labels are externally observable. Empty labels ϵ are ignored in constructing traces, in the usual style of labeled transition systems.

An implementation $\mathbb{I}$ is causally consistent if and only if the concrete operational semantics instantiated with the implementation $\rightarrow_{\mathcal{C}(\mathbb{I})}$ is a refinement of the abstract causal operational semantics $\rightarrow_{\mathcal{A}}$. An operational semantics is a refinement of another operational semantics if and only if every external trace of the former is an external trace of the latter. In other words, an implementation is causally consistent if and only if, for any concrete execution of any program (with the implementation), there exists an abstract execution of the program with the same external history:

Definition 2 (Causal Consistency).
$\mathsf{CauseConst}(\mathbb{I}) \triangleq \forall p, h_{\mathcal{C}}, W_{\mathcal{C}} :$

$$W_{\mathcal{C}0}(p) \xrightarrow{h_{\mathcal{C}}}{}^{*}_{\mathcal{C}(\mathbb{I})} W_{\mathcal{C}} \;\Rightarrow$$
$$\exists h_{\mathcal{A}}, W_{\mathcal{A}} : W_{\mathcal{A}0}(p) \xrightarrow{h_{\mathcal{A}}}{}^{*}_{\mathcal{A}} W_{\mathcal{A}} \wedge \mathsf{Ext}_{\mathcal{C}}(h_{\mathcal{C}}) = \mathsf{Ext}_{\mathcal{A}}(h_{\mathcal{A}})$$

By the above definition, if an implementation is causally consistent, any program executed with the implementation exhibits external behaviors that it would exhibit with the abstract operational semantics as well. In particular, if there is an execution of the program with the implementation that exhibits an assertion failure, there exists an abstract execution of the program that exhibits some assertion failure. We defined in Definition 1 that a program is causally content if and only if every abstract execution of the program is assertion-fail-free. Thus, we can conclude that every execution of a causally content program with a causally consistent implementation is assertion-fail-free.

Lemma 1.
$\forall \mathbb{I}, p, h_{\mathcal{C}}, W_{\mathcal{C}} :$

$$(\mathsf{CauseConst}(\mathbb{I}) \wedge \mathsf{CauseContent}(p)$$
$$\wedge\, W_{\mathcal{C}0}(p) \xrightarrow{h_{\mathcal{C}}}{}^{*}_{\mathcal{C}(\mathbb{I})} W_{\mathcal{C}}) \;\Rightarrow\; assertfail \notin h_{\mathcal{C}}$$

The immediate implication of this lemma is that the implementations and the clients can be verified separately to be causally consistent and causally content respectively, and the combination of every pair of verified implementation and verified program is safe. In the remainder of this section, we focus on proving the causal consistency of implementations. We consider proving that programs are causally content in § 6.

Definition 2 gives the direct way of proving the causal consistency of an implementation: prove that the concrete semantics instantiated with the implementation refines the abstract semantics. However, establishing a refinement for every implementation can be repetitive. We present a proof technique called *well-reception* for the proof of causal consistency of implementations. It factors out a significant part of the proofs of causal consistency for implementations and requires a limited number of specific obligations to be proved for each implementation.

In the following paragraphs, we present the well-reception condition WellRec for key-value store implementations. The following theorem proves that it is a sufficient condition for causal consistency.

Theorem 2 (Sufficiency of Well-Reception).
$\forall \mathbb{I} \colon \mathsf{WellRec}(\mathbb{I}) \Rightarrow \mathsf{CauseConst}(\mathbb{I})$

The theorem states that every well-receptive implementation is causally consistent. With this theorem, to prove that an implementation is causally consistent, it is sufficient to prove that the implementation is well-receptive. Well-reception is defined based on an instrumented operational semantics. First, we consider the

PUT

$$\frac{\begin{array}{c} c' = c + 1 \\ \mathsf{put}(IV, n, \sigma, k, (n, c', v)) \rightsquigarrow^* (\sigma', u) \\ l = n, c' \triangleright \mathit{put}(\sigma, k, v) \colon \sigma',\ u \\ t' = t \cup \{(n, c', n', k, v, u, l) \mid n' \in N \setminus \{n\}\} \end{array}}{\begin{array}{c} (h[n \mapsto (\mathit{put}(k, v); s, \sigma, c)], t) \\ \xrightarrow{n, c' \triangleright\, \mathit{put}(\sigma, k, v) \colon\, \sigma',\, u}_{\mathcal{I}(\mathbb{I})} \\ \big(h[n \mapsto (s, \sigma', c')], t'\big) \end{array}}$$

GET

$$\frac{\mathsf{get}(IV, n, \sigma, k) \rightsquigarrow^* ((n', c', v), \sigma')}{\begin{array}{c} (h[n \mapsto (x\ \leftarrow\ \mathit{get}(k); s, \sigma, c)], t) \\ \xrightarrow{n', c', n \triangleright\, \mathit{get}(\sigma, k) \colon\, \sigma',\, v}_{\mathcal{I}(\mathbb{I})} \\ \big(h[n \mapsto (s[x := v], \sigma', c)], t\big) \end{array}}$$

UPDATE

$$\frac{\begin{array}{c} m = (n', c', n, k, v, u, l) \\ \mathsf{guard}(IV, n, \sigma, k, (n', c', v), u) \rightsquigarrow^* \mathit{true} \\ \mathsf{update}(IV, n, \sigma, k, (n', c', v), u) \rightsquigarrow^* \sigma' \end{array}}{\begin{array}{c} (h[n \mapsto (s, \sigma, c)], t \cup \{m\}) \\ \xrightarrow{n', c', n \triangleright\, \mathit{update}(\sigma, k, v, m) \colon\, \sigma'}_{\mathcal{I}(\mathbb{I})} \\ \big(h[n \mapsto (s, \sigma', c)], t\big) \end{array}}$$

ASSERTFAIL

$$\frac{}{(h[n \mapsto (\mathit{assertfail}, \sigma, c)], t) \xrightarrow{\mathit{assertfail}}_{\mathcal{I}(\mathbb{I})} (h[n \mapsto (\mathit{skip}, \sigma, c)], t)}$$

$$\begin{array}{rcll} & & IV = N \times C \times V & \text{Instrumented Val} \\ h & : & H = N \rightarrow (S \times \mathsf{State}(IV) \times C) & \text{Hosts} \\ m & : & M = N \times C \times N \times & \text{Message} \\ & & \quad K \times V \times \mathsf{Update}(IV) \times L & \\ t & : & T = \mathbb{PM}(M) & \text{Transit} \\ W_{\mathcal{I}} & : & H \times T & \text{World} \\ \sigma & : & \mathsf{State}(IV) & \text{Alg State} \\ u & : & \mathsf{Update}(IV) & \text{Alg Update} \\ l_{\mathcal{I}} & : & L & \text{Label} \\ & ::= & n, c \triangleright \mathit{put}(\sigma, k, v) \colon \sigma', u & \\ & \mid & n', c', n \triangleright \mathit{get}(\sigma, k) \colon \sigma', v & \\ & \mid & n', c', n \triangleright \mathit{update}(\sigma, k, v, m) \colon \sigma' & \\ & \mid & \mathit{assertfail} & \\ h_{\mathcal{I}} & ::= & {l_{\mathcal{I}}}^* & \text{History} \\ W_{\mathcal{I}0}(p) & \triangleq & (\lambda n.(p(n), \mathsf{init}(IV, (n_{init}, 0, v_0)), 0), \emptyset) & \end{array}$$

Figure 9. Concrete Instrumented Operational Semantics $\rightarrow_{\mathcal{I}(\mathbb{I})}$ for the implementation $\mathbb{I} = (\mathsf{State}, \mathsf{Update}, \mathsf{init}, \mathsf{put}, \mathsf{get}, \mathsf{guard}, \mathsf{update})$ and the value type V

instrumented operational semantics, and then we flesh out the well-reception condition.

Figure 9 presents the concrete instrumented operational semantics $\rightarrow_{\mathcal{I}(\mathbb{I})}$. It is parametric on the implementation $\mathbb{I} = (\mathsf{State}, \mathsf{Update}, \mathsf{init}, \mathsf{put}, \mathsf{get}, \mathsf{guard}, \mathsf{update})$ and the value type V. The instrumented semantics is similar to the (non-instrumented) concrete operational semantics $\rightarrow_{\mathcal{C}(\mathbb{I})}$ presented in Figure 6. The crucial difference is that it uniquely identifies put operations to track causal dependencies between them. Each node maintains a clock number that is advanced on every put operation by the node. The unique identifier of a put operation is the pair of its issuing node identifier and the clock number of the node when the put is issued. To track the origins of values, every value is coupled with the identifier of its originating put operation. Thus, an *instrumented value* (IV) is a triple of a node identifier, clock number, and value. The semantics instruments the values that the client program puts, employs the implementation to store and retrieve instrumented values, and deinstruments values before returning them to the client program.

$$\begin{array}{l} \mathsf{LNode}(l_{\mathcal{I}}) \triangleq \\ \quad \begin{cases} n & \text{if } l_{\mathcal{I}} = n, c \triangleright \mathit{put}(\sigma, k, v) \colon \sigma', u \\ n & \text{if } l_{\mathcal{I}} = n', c', n \triangleright \mathit{get}(\sigma, k) \colon \sigma', v \\ n & \text{if } l_{\mathcal{I}} = n', c', n \triangleright \mathit{update}(\sigma, k, v, m) \colon \sigma' \\ n_0 & \text{if } l_{\mathcal{I}} = assertfail \end{cases} \\ \mathsf{LClock}(l_{\mathcal{I}}) \triangleq \\ \quad \begin{cases} c & \text{if } l_{\mathcal{I}} = n, c \triangleright \mathit{put}(\sigma, k, v) \colon \sigma', u \\ c' & \text{if } l_{\mathcal{I}} = n', c', n \triangleright \mathit{get}(\sigma, k) \colon \sigma', v \\ c' & \text{if } l_{\mathcal{I}} = n', c', n \triangleright \mathit{update}(\sigma, k, v, m) \colon \sigma' \\ 0 & \text{if } l_{\mathcal{I}} = assertfail \end{cases} \\ \mathsf{LPostState}(l_{\mathcal{I}}) \triangleq \\ \quad \begin{cases} \sigma' & \text{if } l_{\mathcal{I}} = n, c \triangleright \mathit{put}(\sigma, k, v) \colon \sigma', u \\ \sigma' & \text{if } l_{\mathcal{I}} = n', c', n \triangleright \mathit{get}(\sigma, k) \colon \sigma', v \\ \sigma' & \text{if } l_{\mathcal{I}} = n', c', n \triangleright \mathit{update}(\sigma, k, v, m) \colon \sigma' \\ \sigma_0 & \text{if } l_{\mathcal{I}} = assertfail \end{cases} \\ l_{\mathcal{I}} \prec_{h_{\mathcal{I}}} l'_{\mathcal{I}} \triangleq \\ \qquad l_{\mathcal{I}} \text{ precedes } l'_{\mathcal{I}} \text{ in the sequence of labels } h_{\mathcal{I}} \end{array}$$

Figure 10. Label Functions

We instantiate the functions of $\mathbb{I}$ with IV (whereas our concrete semantics instantiate them with the plain value type, V). As part of the proof of Theorem 2, a refinement from $\rightarrow_{\mathcal{C}(\mathbb{I})}$ to $\rightarrow_{\mathcal{I}(\mathbb{I})}$ is proved for every implementation $\mathbb{I}$, and a refinement from $\rightarrow_{\mathcal{I}(\mathbb{I})}$ to $\rightarrow_{\mathcal{A}}$ is proved for every well-receptive implementation $\mathbb{I}$.

The definition of programs is unchanged from Figure 4. Similarly to the (non-instrumented) concrete semantics, worlds $W_{\mathcal{I}}$ are pairs of the host states H and the in-transit messages T. We only mention the differences in the states and labels. Each node maintains a clock number in addition to the statement and the implementation state of the node. A message tuple includes the identifier of the originating put operation and the label of the put transition that created it. A label $l_{\mathcal{I}}$ is either a put, get, update, or assertion-failure label. The first three labels include the implementation prestate σ and poststate σ' of the transition. A put label contains the unique identifier of the performed put operation. A get label contains the identifier of the put operation whose put value is returned. An update label contains the identifier of the originating put operation and the processed message as well. We use the predicates LIsPut, LIsGet, and $\mathsf{LIsUpdate}$ on labels that are satisfied respectively on only put, get, and update labels. We also use the auxiliary functions LNode, LClock, and $\mathsf{LPostState}$ and the precedence relation $\prec$ on labels that are defined in Figure 10.

The rule PUT executes a put statement with the key k and the value v by the node n. It increments the clock c to yield the current clock c'. It instruments the value v with the current put identifier n, c' and calls the put function of the implementation with the instrumented value. Thus, the implementation stores the put identifier together with the value. The label includes the current put identifier. The inclusion of the put identifier in the put label allows us to uniquely identify put labels. The rule GET executes a get statement on the key k. Calling the get function of the implementation yields an instrumented value (n', c', v) where the identifier of the originating put is n', c' and the value is v. The value v is returned to the client program, and the identifier of the originating put is included in the get label. The inclusion of this put identifier in the get label allows us to identify the put transition that created the returned value. The rule UPDATE receives a message sent to the current node and applies the update that it carries. The received message is included in the update label. The PUT rule includes the label of the put transition in the sent messages. Therefore, given an update label, the put label whose update is applied is immediately available in the message field of the label.

The correctness condition $\mathsf{WellRec}$ is presented in Figure 11. To make the condition more readable, we use underscore _ to skip the record fields that are not used in the context. An imple-

$$
\begin{array}{l}
\mathsf{WellRec}(\mathbb{I}) \ \triangleq \\
\quad \mathit{let}\ (\mathsf{State}, \_, \_, \_, \_, \_, \_) = \mathbb{I}\ \mathit{in} \\
\quad \exists \mathsf{Rec}\colon (\mathsf{State}(IV), N) \rightarrow C\colon \\
\quad \mathit{let}\ \mathsf{Rec}'(W, n', n) = \\
\qquad \mathit{let}\ (H[n' \mapsto (\_, \sigma, \_)], \_) = W\ \mathit{in} \\
\qquad \mathsf{Rec}(\sigma, n)\ \mathit{in} \\
\quad \mathsf{InitCond}(\mathbb{I}, \mathsf{Rec}') \wedge \mathsf{StepCond}(\mathbb{I}, \mathsf{Rec}') \wedge \mathsf{CauseCond}(\mathbb{I}, \mathsf{Rec}') \\
\quad \wedge\ \mathsf{SeqCond}(\mathbb{I})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{InitCond}(\mathbb{I}, \mathsf{Rec}') \ \triangleq\ \forall p, n, n'\colon \\
\quad \mathsf{Rec}'(W_{\mathcal{I}0}(p), n, n') = 0
\end{array}
$$

$$
\begin{array}{l}
\mathsf{StepCond}(\mathbb{I}, \mathsf{Rec}') \ \triangleq\ \forall p, h_{\mathcal{I}}, W_{\mathcal{I}}, l_{\mathcal{I}}, W'_{\mathcal{I}}\colon \\
\quad (W_{\mathcal{I}0}(p) \xrightarrow{h_{\mathcal{I}}}{}^{*}_{\mathcal{I}(\mathbb{I})} W_{\mathcal{I}} \ \wedge\ W_{\mathcal{I}} \xrightarrow{l_{\mathcal{I}}}_{\mathcal{I}(\mathbb{I})} W'_{\mathcal{I}}) \Rightarrow \\
\quad \begin{cases}
\textsc{Case}\ l_{\mathcal{I}} = n, \_ \triangleright \mathit{put}(\_, \_, \_)\colon \_, \_ \\
\quad \mathsf{Rec}'(W'_{\mathcal{I}}, n, n) = \mathsf{Rec}'(W_{\mathcal{I}}, n, n) + 1 \wedge \\
\quad \forall n'\colon n' \neq n \Rightarrow \mathsf{Rec}'(W'_{\mathcal{I}}, n, n') = \mathsf{Rec}'(W_{\mathcal{I}}, n, n') \\
\textsc{Case}\ l_{\mathcal{I}} = n, \_ \triangleright \mathit{get}(\_, \_)\colon \_, \_ \\
\quad \forall n'\colon \mathsf{Rec}'(W'_{\mathcal{I}}, n, n') = \mathsf{Rec}'(W_{\mathcal{I}}, n, n') \\
\textsc{Case}\ l_{\mathcal{I}} = n', c', n \triangleright \mathit{update}(\_, \_, \_, \_)\colon \_ \\
\quad \mathsf{Rec}'(W_{\mathcal{I}}, n, n') + 1 = c' \wedge \\
\quad \mathsf{Rec}'(W'_{\mathcal{I}}, n, n') = \mathsf{Rec}'(W_{\mathcal{I}}, n, n') + 1 \wedge \\
\quad \forall n''\colon n'' \neq n' \Rightarrow \mathsf{Rec}'(W'_{\mathcal{I}}, n, n'') = \mathsf{Rec}'(W_{\mathcal{I}}, n, n'')
\end{cases}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{CauseCond}(\mathbb{I}, \mathsf{Rec}') \ \triangleq\ \forall p, h_{\mathcal{I}}, W_{\mathcal{I}}, l_{\mathcal{I}}, W'_{\mathcal{I}}, l''_{\mathcal{I}}\colon \\
\quad (W_{\mathcal{I}0}(p) \xrightarrow{h_{\mathcal{I}}}{}^{*}_{\mathcal{I}(\mathbb{I})} W_{\mathcal{I}} \ \wedge\ W_{\mathcal{I}} \xrightarrow{l_{\mathcal{I}}}_{\mathcal{I}(\mathbb{I})} W'_{\mathcal{I}} \\
\qquad \wedge\ \mathsf{LIsUpdate}(l_{\mathcal{I}}) \wedge \\
\qquad \mathit{let}\ \_, \_, n \triangleright \mathit{update}(\_, \_, \_, m)\colon \_ = l_{\mathcal{I}} \\
\qquad\quad (\_, \_, \_, \_, \_, \_, l'_{\mathcal{I}}) = m\ \mathit{in} \\
\qquad \mathsf{LIsPut}(l''_{\mathcal{I}})\ \wedge\ l''_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l'_{\mathcal{I}}) \Rightarrow \\
\quad \mathit{let}\ n'', c'' \triangleright \mathit{put}(\_, \_, \_)\colon \_, \_\ = l''_{\mathcal{I}}\ \mathit{in} \\
\quad \mathsf{Rec}'(W_{\mathcal{I}}, n, n'') \geq c''
\end{array}
$$

$$
\begin{array}{l}
\mathsf{SeqCond}(\mathbb{I}) \ \triangleq\ \forall p, h_{\mathcal{C}}, W_{\mathcal{C}}\colon \\
\quad W_{\mathcal{C}0}(p) \xrightarrow{h_{\mathcal{C}}}{}^{*}_{\mathcal{C}(\mathbb{I})} W_{\mathcal{C}} \ \Rightarrow\ \exists W'_{\mathcal{S}}\colon W_{\mathcal{S}0} \xrightarrow{\mathsf{Eff}(h_{\mathcal{C}})}{}^{*}_{\mathcal{S}} W'_{\mathcal{S}}
\end{array}
$$

Figure 11. Correctness condition WellRec for implementation $\mathbb{I}$

$$
\begin{array}{l}
l_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l'_{\mathcal{I}} \triangleq \\
\qquad \mathsf{StepCause}^{+}_{h_{\mathcal{I}}}(l_{\mathcal{I}}, l'_{\mathcal{I}}) \\
\mathsf{StepCause}_{h_{\mathcal{I}}} \triangleq \\
\qquad \mathsf{NodeOrderCause}_{h_{\mathcal{I}}} \cup \mathsf{GetsFromCause}_{h_{\mathcal{I}}} \\
\mathsf{NodeOrderCause}_{h_{\mathcal{I}}}(l_{\mathcal{I}}, l'_{\mathcal{I}}) \triangleq \\
\qquad l_{\mathcal{I}} \prec_{h_{\mathcal{I}}} l'_{\mathcal{I}}\ \wedge\ \mathsf{LNode}(l_{\mathcal{I}}) = \mathsf{LNode}(l'_{\mathcal{I}}) \wedge \\
\qquad ((\mathsf{LIsGet}(l_{\mathcal{I}})\ \wedge\ \mathsf{LIsPut}(l'_{\mathcal{I}})) \\
\qquad\ \ \vee\ (\mathsf{LIsPut}(l_{\mathcal{I}})\ \wedge\ \mathsf{LIsPut}(l'_{\mathcal{I}}))) \\
\mathsf{GetsFromCause}_{h_{\mathcal{I}}}(l_{\mathcal{I}}, l'_{\mathcal{I}}) \triangleq \\
\qquad l_{\mathcal{I}} \in h_{\mathcal{I}} \wedge l'_{\mathcal{I}} \in h_{\mathcal{I}}\ \wedge\ \mathsf{LIsPut}(l_{\mathcal{I}}) \wedge \mathsf{LIsGet}(l'_{\mathcal{I}}) \wedge \\
\qquad \mathit{let}\ n, c \triangleright \mathit{put}(\_, \_, \_)\colon \_, \_ = l_{\mathcal{I}} \\
\qquad\quad n', c', n'' \triangleright \mathit{get}(\_, \_)\colon \_, \_ = l'_{\mathcal{I}}\ \mathit{in} \\
\qquad n = n' \wedge c = c'
\end{array}
$$

Figure 12. Causal Relation $\curvearrowright$

mentation is *well-receptive* if and only if there exists a function Rec for the implementation such that the four conditions InitCond, StepCond, CauseCond, and SeqCond are satisfied. The function Rec (mnemonic for "received"), given a node state σ and a node identifier n, should return the number of updates that σ has received from n. We define an auxiliary function Rec$'$ that given a world $W_{\mathcal{I}}$ and node identifiers n' and n, returns the value of Rec for the implementation state of node n' in $W_{\mathcal{I}}$ and the node identifier n. Now, let us look at each of the conditions in turn.

$$
\textsc{Put}\quad \frac{}{W_{\mathcal{S}}[n \mapsto m] \xrightarrow{n \triangleright\ put(k,v)}_{\mathcal{S}} W_{\mathcal{S}}[n \mapsto m[k \mapsto v]]}
$$

$$
\textsc{Get}\quad \frac{m(k) = v}{W_{\mathcal{S}}[n \mapsto m] \xrightarrow{n \triangleright\ get(k)\colon\ v}_{\mathcal{S}} W_{\mathcal{S}}[n \mapsto m]}
$$

$$
\textsc{AssertFail}\quad \frac{}{W_{\mathcal{S}} \xrightarrow{assertfail}_{\mathcal{S}} W_{\mathcal{S}}}
$$

$$
\begin{array}{rcll}
W_{\mathcal{S}} & : & N \rightarrow K \rightarrow V & \text{World} \\
W_{\mathcal{S}0}(p) & \triangleq & \lambda n, k.\ v_0 & \\
\mathsf{Eff}(l_{\mathcal{C}}) & \triangleq & \begin{cases}
n \triangleright \mathit{put}(k, v) & \text{if } l_{\mathcal{C}} = n \triangleright \mathit{put}(k, v) \\
n \triangleright \mathit{get}(k)\colon v & \text{if } l_{\mathcal{C}} = n \triangleright \mathit{get}(k)\colon v \\
n \triangleright \mathit{put}(k, v) & \text{if } l_{\mathcal{C}} = n \triangleright \mathit{update}(k, v) \\
\mathit{assertfail} & \text{if } l_{\mathcal{C}} = \mathit{assertfail}
\end{cases} & \\
\mathsf{Eff}(l_{\mathcal{C}}{}^{*}) & \triangleq & \mathsf{Eff}(l_{\mathcal{C}})^{*} &
\end{array}
$$

Figure 13. Sequential Operational Semantics $\rightarrow_{\mathcal{S}}$

- InitCond: This condition requires that any initial state returned by the implementation's init function has a 0 received-message count according to Rec. In other words, the initial state should not have received any update from any node.
- StepCond: This condition requires the implementation to affect the mapping of the received function Rec for a node only when an update from that node is received. More precisely, in a put step, the mapping of Rec for the current node should be incremented, and the mapping of Rec for every other node should remain unchanged. In a put step, the current node is implicitly receiving an update from itself. In a get step, the mapping of Rec for every node should remain unchanged. In an update step by a node n that applies an update from another node n' with the clock number c', the clock number of the next expected update from n' should be c'. In other words, the successor of the number of already-received updates from n' should be c'. The mapping of Rec for the originating node n' should be incremented, and the mapping of Rec for every other node should remain unchanged.
- CauseCond: This condition is the most important of the four. Let us first see the definition of the causality relation $\curvearrowright$ on labels in Figure 12. It is a partial order that is defined as the transitive closure of the step-causal relation StepCause. The step-causal relation is the union of the two relations NodeOrderCause and GetsFromCause.
  - Node-order relation NodeOrderCause: A put label is causally dependent on the preceding get and put labels of the same node.
  - Gets-from relation GetsFromCause: A get label is causally dependent on the label of the originating put transition that provided the value. Note that the unique put identifiers that the instrumented values and labels carry help us match the get label to the originating put label.

  The condition CauseCond requires that if an update is being received that is originated from the put label $l'_{\mathcal{I}}$, and $l'_{\mathcal{I}}$ is causally dependent on another put label $l''_{\mathcal{I}}$, then the update of $l''_{\mathcal{I}}$ should have been received already. More precisely, for every step with an update label $l_{\mathcal{I}}$, if m is the message received by the step, the put label that created m is $l'_{\mathcal{I}}$, and $l'_{\mathcal{I}}$ is dependent on another put label $l''_{\mathcal{I}}$, then the number of updates received from the node of $l''_{\mathcal{I}}$ should be greater than or equal to the clock of $l''_{\mathcal{I}}$.

```
$\mathbb{I}_1$ (ALGORITHM 1)
State
  ($store$: $K \to V$,
   $clock$: $N \to C$)
Update
  ($unode$: $N$,
   $uclock$: $N \to C$)
init ($v_0$)
  ret ($\lambda k.v_0, \lambda n.0$)
put ($self, this$)($k, v$)
  $(s, c) \leftarrow this$;
  $c' \leftarrow c[self \mapsto c[self] + 1]$;
  $s' \leftarrow s[k \mapsto v]$;
  ret $((s', c'), (self, c'))$;
get ($self, this$)($k$)
  $(s, c) \leftarrow this$;
  ret $(s[k], (s, c))$
guard ($self, this$)($k, v, u$)
  $(s, c) \leftarrow this$;
  $(n', c') \leftarrow u$;
  ret forall $(\lambda n.\ n \neq n' \Rightarrow c'[n] \leq c[n])$ $N$
    $\wedge\ c'[n'] = c[n'] + 1$
update ($self, this$)($k, v, u$)
  $(s, c) \leftarrow this$;
  $(n', c') \leftarrow u$;
  $c'' \leftarrow c[n' \mapsto c'[n']]$;
  $s'' \leftarrow s[k \mapsto v]$;
  ret $(s'', c'')$
```

Figure 14. Causally Consistent Map Implementation 1 [4]

For example, in Figure 1, the second put of Alice is dependent on her first put. Therefore, when the second update is applied to Bob's replica, her first update should have already been applied. As another example, in Figure 2, Alice's second put is dependent on her first put, and Bob's put is dependent on Alice's second put. Thus, when Alice's second update is being applied to Carol's replica, Alice's first update should have been applied. Similarly, when Bob's update is being applied to Carol's replica, Alice's second update should have been applied.

- SeqCond: This condition checks that the functions put and update mutate the mapping for and only for the given key to the given value, and that the function get returns the mapping for the given key. In other words, it requires that if we treat updates as simple puts, then the implementation refines a sequential map. Figure 13 defines the sequential operational semantics $\rightarrow_{\mathcal{S}}$ and the effect function Eff. The operational semantics $\rightarrow_{\mathcal{S}}$ defines the semantics of a separate sequential map for each node. The function Eff translates an update label to a put label with the same key and value, leaving the other labels unchanged. The function is naturally lifted to histories. Based on these two definitions, the SeqCond condition requires the following for the implementation $\mathbb{I}$: for every program p, for every execution of p with the concrete operational semantics $\rightarrow_{\mathcal{C}(\mathbb{I})}$, there exists an execution of the sequential operational semantics $\rightarrow_{\mathcal{S}}$ whose history is the effect history of the concrete execution. Unlike the previous two conditions that are stated on the instrumented concrete operational semantics $\rightarrow_{\mathcal{I}(\mathbb{I})}$, SeqCond is stated on the concrete operational semantics $\rightarrow_{\mathcal{C}(\mathbb{I})}$. We have separately proved a refinement from $\rightarrow_{\mathcal{C}(\mathbb{I})}$ to $\rightarrow_{\mathcal{I}(\mathbb{I})}$ for every $\mathbb{I}$.

5. Implementations

Now, we present and verify two key-value store implementations from the literature. We also present and verify a variant of the first implementation in Appendix C. We have proved the causal consistency of these implementations using the well-reception proof technique and mechanically checked the proofs with Coq. In this section, we present explanations and parts of the proof sketches. The full proof sketches are available in the appendix.

5.1 Implementation 1

The first implementation that we consider is by Ahamad and others [4], which we present in Figure 14. This implementation maintains a vector clock for each node to serve as (an over-approximation of) the node's dependencies. It is a map from node identifiers to the number of updates from that node that the current node has applied. It is notable that a node's applied updates is an over-approximation of the node's dependencies as there may be updates that are applied but whose values are not returned by a get operation. An update sent from a node to another contains the vector clock of the sender. The receiving node ensures that the dependencies of the update are satisfied by checking that its own vector clock is ahead of the sender's vector clock.

The state of each node contains a store function, $store$, and a vector-clock function, $clock$. The store maps keys to values, and the vector clock maps each node identifier to the number of updates received from it. An update contains the identifier ($unode$) and vector clock ($uclock$) of the sender node.

The put function, on key k and value v, first increments the mapping of the vector clock for the current node $self$. (Because a node never issues an update to itself, the vector clock of each node maps $self$ to the number of put operations that it has performed.) Then it updates the mapping of the store for k to v. The get function on k returns the current mapping for k.

When applying an update u, where n' is the sender node, c' is the sender's vector clock, and c is the vector clock of the current node, the guard function checks the following two conditions. Firstly, it checks that for every node n other than n', the value of c' for n is less than or equal to the value of c for n. This means that for every node n other than the sender node n', the number of messages that the sender node had received from n is less than or equal to the number of messages that the current node has received from n. Secondly, the guard method checks that the clock value of c' for n' (i.e. $c'(n')$) is the successor of the clock value of c for n' (i.e. $c(n')$). As mentioned above, the clock value of every node for the node itself is the number of put operations that the node has issued. Thus, the update that is being received is sent by the $c'(n')$-th put operation of n'. The current node has received $c(n')$ updates from n'. Thus, this condition checks that the update that is being received is the next expected update from n'. In summary, the guard condition makes sure that every update that had been applied to the sender is already applied to the current node as well.

If the guard condition is satisfied, then the update function applies the update from n'. It updates the mapping of c for n' to the sender's vector clock c' value for n'. Note that, by the second condition that the guard function checks, $c'(n')$ is equal to the successor of $c(n')$. Thus, the update method effectively increments the mapping of c for n'. It also updates the mapping of the current store s for the key k to the new value v.

Now, we prove the causal consistency of the implementation using the well-reception proof technique. In the following paragraphs, we prove that $\mathbb{I}_1$ is well-receptive.

Theorem 3. WellRec($\mathbb{I}_1$)

By Theorem 2, we immediately conclude that $\mathbb{I}_1$ is causally consistent.

Corollary 1. CauseConst($\mathbb{I}_1$)

The condition WellRec($\mathbb{I}$), defined in Figure 11, requires us to provide a function Rec. Given the state of a node, σ, and a node identifier, n, Rec must return the number of updates that σ has received from n. In this implementation, $clock$ stores the number of updates received from other nodes. Therefore, we define Rec to be $clock$. Let us define the function $clock'$ that mirrors the function Rec' in Figure 11 as follows

$$clock'(W, n', n) \triangleq \textit{let } (H[n' \mapsto (\_, \sigma, \_)], \_) = W \textit{ in } clock(\sigma)(n)$$

To prove the $\mathsf{CauseCond}$ condition, we need to prove the following monotonicity property for vector clocks. We refer to the vector clock of the poststate of a label as the vector clock of that label. If a label $l_{\mathcal{I}}$ causally precedes another label $l'_{\mathcal{I}}$, the clock of $l_{\mathcal{I}}$ is less than or equal to the clock of $l'_{\mathcal{I}}$ for every node. Further, if $l'_{\mathcal{I}}$ is a put label, the clock of $l_{\mathcal{I}}$ is strictly less than the clock of $l'_{\mathcal{I}}$ for the node identifier of $l'_{\mathcal{I}}$.

Lemma 2 (Clock Monotonicity).

$$\begin{array}{l} \forall p, h_{\mathcal{I}}, W_{\mathcal{I}}, l_{\mathcal{I}}, l'_{\mathcal{I}}, n\colon \\ (W_{\mathcal{I}0}(p) \xrightarrow{h_{\mathcal{I}}}{}^{*}_{\mathcal{I}(\mathbb{I}_1)} W_{\mathcal{I}} \ \wedge \ l_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l'_{\mathcal{I}}) \Rightarrow \\ (clock(\mathsf{LPostState}(l_{\mathcal{I}}), n) \leq clock(\mathsf{LPostState}(l'_{\mathcal{I}}), n) \\ \quad \wedge \ (\mathsf{LIsPut}(l'_{\mathcal{I}}) \wedge n = LNode(l'_{\mathcal{I}})) \Rightarrow \\ \qquad clock(\mathsf{LPostState}(l_{\mathcal{I}}), n) < clock(\mathsf{LPostState}(l'_{\mathcal{I}}), n)) \end{array}$$

Let us see why the above lemma holds. By the definitions in Figure 12, the causal order holds by either the node order, gets-from relation, or transitivity. Firstly, if it holds by the node order, the conclusion is immediate by noting the following facts. On every step, the mapping of the vector clock for every node is nondecreasing. On a put step, the vector clock of the node for the node itself is incremented. Secondly, we consider the case that the causal relation holds by a gets-from relation from the put label $l_{\mathcal{I}}$ to the get label $l'_{\mathcal{I}}$. Let n and n' be the node identifiers of $l_{\mathcal{I}}$ and $l'_{\mathcal{I}}$ respectively. The get label $l'_{\mathcal{I}}$ can get the value put by the put label $l_{\mathcal{I}}$ only if there exists an update label $l''_{\mathcal{I}}$ by n' before $l'_{\mathcal{I}}$ that receives the update of $l_{\mathcal{I}}$. When the update is being received, the guard function checks that the vector clock value of $l_{\mathcal{I}}$ for every node n'' other than n is less than or equal to the current vector clock value for n''. Then the update function remaps the current vector clock value for n to the vector clock value of $l_{\mathcal{I}}$ for n. Thus, the vector clock of $l_{\mathcal{I}}$ is less than or equal to the vector clock of $l''_{\mathcal{I}}$ for every node. As mentioned above, the vector clock of a node is nondecreasing. Therefore, as $l''_{\mathcal{I}}$ precedes $l'_{\mathcal{I}}$, and they are by the same node n', the vector clock of $l''_{\mathcal{I}}$ is less than or equal to the vector clock of $l'_{\mathcal{I}}$ for every node. The inequality of the conclusion is immediate from the transitivity of the above two inequalities. Thirdly, if the causal order holds by the transitivity of other causal orders, the conclusion is immediate from the transitivity of the equalities and inequalities of the induction hypotheses.

5.2 Implementation 2

Our next case study is the implementation by Lloyd and others [32]. The implementation is presented in Figure 15. To keep the focus on causal consistency, we model only their causally consistent implementation and not their support for convergence. Instead of keeping track of *all* the dependencies of a put operation, this implementation only tracks its *one-hop*, or immediate, dependencies. As we will see, it is sufficient to only record the one-hop dependencies because of the transitive property of dependencies.

Each node maintains four components. The $clock$ stores the current clock number of the node. It counts the number of puts that the node has issued. The $store$ maps keys to pairs of values and timestamps, where a timestamp is a tuple of the identifier and clock number of the value's originating node. The "received" function rec is a function from a node identifier to the clock of the latest update that is received from the node. The dependencies dep is a list of timestamps of put operations that the current state of the node, and thus its future put operations, depends on. Each update is a record of the following fields. The fields $unode$ and $uclock$ store the node identifier and the clock number of the sender node. The field $udep$ stores the dependencies of the update.

```
$\mathbb{I}_2$ (ALGORITHM 2)
State
  ($store$: $K \to (V, N, C)$,
   $rec$: $N \to C$,
   $clock$: $C$,
   $dep$: $List[(N, C)]$)
Update
  ($unode$: $N$,
   $uclock$: $C$,
   $udep$: $List[(N, C)]$)
init ($v_0$)
  ret ($\lambda k.(v_0, n_0, 0), \lambda n.0, 0, []$)
put ($self, this$)($k, v$)
  $(s, r, c, d) \leftarrow this$;
  $c' \leftarrow c + 1$;
  $s' \leftarrow s[k \mapsto (v, self, c')]$;
  $r' \leftarrow r[self \mapsto c']$;
  $d' \leftarrow (self, c') :: []$;
  ret $((s', r', c', d'), (self, c', d))$
get ($self, this$)($k$)
  $(s, r, c, d) \leftarrow this$;
  $(v, n', c') \leftarrow s[k]$;
  $d' \leftarrow$ if $n' \neq n_0$ then $(n', c') :: d$ else $d$;
  ret $(v, (s, r, c, d'))$
guard ($self, this$)($k, v, u$)
  $(\_, r, \_, \_) \leftarrow this$;
  $(\_, \_, d') \leftarrow u$;
  ret forall $(\lambda(n', c').\ r(n') \geq c')\ d'$
update ($self, this$)($k, v, u$)
  $(s, r, c, d) \leftarrow this$;
  $(n', c', \_) \leftarrow u$;
  $s' \leftarrow s[k \mapsto (v, n', c')]$;
  $r' \leftarrow r[n' \mapsto c']$;
  ret $(s', r', c, d)$
```

Figure 15. Causally Consistent Map Implementation 2 [32]

On invocation of put on the key k with the value v, the state is updated as follows. The clock c is incremented to yield the clock c' of the current put operation. The timestamp of the current put operation is the current node identifier $self$ and c'. The store s is updated to map k to v with the current timestamp. The "received" function r is updated to record that the current node is immediately receiving an update from itself. The broadcast update contains the current node identifier $self$, the current clock c', and the dependencies d of the $prestate$. The dependency list is updated to a singleton list d' containing the current timestamp. Every future put operation put' of the current node will be dependent on the current put operation put. Although put' will be dependent on the prestate dependencies d, the poststate dependencies d' do not include d. Only the one-hop dependency is kept. Let us see how transitivity implicitly preserves the dependencies d. We have that put' is dependent on put and put is dependent on d. As we will see, the guard function enforces that other nodes receive the update of put' only if they have already received the update of put. Similarly, it is enforced that they receive the update of put only if they have already received the updates of put operations whose identifiers are in d. Therefore, the dependency of put' on d is implicitly checked by only checking that put is already received.

On invocation of get on key k, the store entry for k is read and the value of the entry is returned to the client. The timestamp of the entry is added to the dependencies only if it is not the dummy timestamp (initialized for all keys originally).

The guard function checks that an update with the dependencies d' is applied only if for every timestamp (n, c) in d', the current node has already received an update with the same or a larger clock value c from the node n. This means that the current node has already received all the dependencies of the update. On invocation of the update function on an update with the timestamp (n', c'), the key k, and the value v, the store function s is updated to map k to v, n', and c'; and the received function r is updated to record c' as the latest clock value received from the sender node n'.

Now, we state the causal consistency of the implementation using the well-reception proof technique.

Theorem 4. $\mathsf{WellRec}(\mathbb{I}_2)$

By Theorem 2, the following corollary follows.

Corollary 2. $\mathsf{CauseConst}(\mathbb{I}_2)$

The condition $\mathsf{WellRec}(\mathbb{I})$, defined in Figure 11, requires us to provide function Rec. In this implementation, the function *rec* stores the number of updates received from other nodes. Therefore, we choose function Rec to be *rec*. Let us define the function *rec*′ that mirrors the definition of Rec' from Figure 11 as follows

$$\begin{array}{lcl} \mathit{rec}'(W, n, n') & \triangleq & \mathit{let}\ (H[n \mapsto (\_, \sigma, \_)], \_) = W\ \mathit{in} \\ & & \mathit{rec}(\sigma)(n') \end{array}$$

To prove the condition $\mathsf{CauseCond}$, we first state two important invariants of the implementation. The first invariant states the transitivity property explained above that if a label $l_\mathcal{I}$ is causally dependent on a put operation $l'_\mathcal{I}$, the identifier of $l'_\mathcal{I}$ is either directly or indirectly in the dependencies of $l_\mathcal{I}$.

Lemma 3 (Update Dependency Transitivity).

$$\begin{array}{l} \forall p, h_\mathcal{I}, W_\mathcal{I}, l_\mathcal{I}, l'_\mathcal{I}\colon \\ (W_{\mathcal{I}0}(p) \xrightarrow{h_\mathcal{I}}{}^{*}_{\mathcal{I}(\mathbb{I}_2)} W_\mathcal{I} \\ \quad \wedge \mathsf{LIsPut}(l_\mathcal{I}) \wedge \mathsf{LIsPut}(l'_\mathcal{I}) \wedge l'_\mathcal{I} \curvearrowright_{h_\mathcal{I}} l_\mathcal{I}) \Rightarrow \\ \mathit{let}\ \_, \_ \rhd \mathit{put}(\_, \_, \_)\colon \_, u = l_\mathcal{I}\ \mathit{in} \\ ((\mathsf{LNode}(l'_\mathcal{I}), \mathsf{LClock}(l'_\mathcal{I})) \in \mathit{udep}(u) \\ \quad \vee (\exists l''_\mathcal{I}\colon \mathsf{LIsPut}(l''_\mathcal{I}) \wedge l'_\mathcal{I} \curvearrowright_{h_\mathcal{I}} l''_\mathcal{I} \\ \qquad\quad \wedge (\mathsf{LNode}(l''_\mathcal{I}), \mathsf{LClock}(l''_\mathcal{I})) \in \mathit{udep}(u))) \end{array}$$

The above lemma states that for every put label $l_\mathcal{I}$ that emits the update u and every put label $l'_\mathcal{I}$ that causally precedes $l_\mathcal{I}$, either the timestamp of $l'_\mathcal{I}$ is directly in $\mathit{udep}(u)$ or there exists a put label $l''_\mathcal{I}$ that depends on $l'_\mathcal{I}$ and the timestamp of $l''_\mathcal{I}$ is in $\mathit{udep}(u)$.

The second invariant states that, if a put label $l_\mathcal{I}$ depends on another put label $l'_\mathcal{I}$ and some node has received the update for $l_\mathcal{I}$, then it has received the update for $l'_\mathcal{I}$ as well.

Lemma 4.

$$\begin{array}{l} \forall p, h_\mathcal{I}, W_\mathcal{I}, l_\mathcal{I}, l'_\mathcal{I}, n\colon \\ (W_{\mathcal{I}0}(p) \xrightarrow{h_\mathcal{I}}{}^{*}_{\mathcal{I}(\mathbb{I}_2)} W_\mathcal{I} \\ \quad \wedge \mathsf{LIsPut}(l_\mathcal{I}) \ \wedge \ \mathsf{LIsPut}(l'_\mathcal{I}) \wedge l'_\mathcal{I} \curvearrowright_{h_\mathcal{I}} l_\mathcal{I} \\ \quad \wedge \mathsf{LClock}(l_\mathcal{I}) \leq \mathit{rec}'(W_\mathcal{I}, n, \mathsf{LNode}(l_\mathcal{I})) \Rightarrow \\ \mathsf{LClock}(l'_\mathcal{I}) \leq \mathit{rec}'(W_\mathcal{I}, n, \mathsf{LNode}(l'_\mathcal{I})) \end{array}$$

The lemma above can be proved by induction on step transitions. The interesting case is the update transition. Consider an update step that receives an update u that is originated from a put label $l_\mathcal{I}$ and that $l_\mathcal{I}$ is causally dependent on another put label $l'_\mathcal{I}$. We want to show that the update of $l'_\mathcal{I}$ is already received. By Lemma 7, we have two cases. Case 1: The identifier of $l'_\mathcal{I}$ is directly in $\mathit{udep}(u)$. The guard method checks that its update is already received. Case 2: The identifier of $l'_\mathcal{I}$ is indirectly in $\mathit{udep}(u)$; that is, there exists another label $l''_\mathcal{I}$ that is causally dependent on $l'_\mathcal{I}$, and the timestamp of $l''_\mathcal{I}$ is in $\mathit{udep}(u)$. As the timestamp of $l''_\mathcal{I}$ is in $\mathit{udep}(u)$, from the guard method checks, we have that the update of $l''_\mathcal{I}$ is already received. As $l''_\mathcal{I}$ is causally dependent on $l'_\mathcal{I}$, and the update of $l''_\mathcal{I}$ is already received, by the induction hypothesis, we have that the update of $l'_\mathcal{I}$ is already received as well.

6. Verification of Client Programs

Having considered proofs of correctness for key-value stores, we return to proofs of safety of their client programs. We defined in § 3 that a program is causally content if and only if every abstract causal execution of the program is assertion-fail-free. Model checking is one approach to showing that a client program is causally content. The basic idea is to automatically explore all executions of the client program to decide whether any assertion fail is reachable. In this section, we present how we implemented and proved a very simple model checker to prove that programs are causally content. The model checker gives us mechanically checked proofs for closed (i.e. all parameters are instantiated with values), terminating programs. Both the implementation and client verification are carried out in Coq. This enables us to seamlessly compose the two results to get end-to-end correctness results as we presented in Theorem 1.

Our first approach was to implement the checker in Coq's tactic language Ltac, which applied repeated inversion on inductive judgments about initial program states evolving to arbitrary final states, thereby considering all possible behaviors. However, this requires a $O((tk)!/tk!)$ search for t nodes and k instructions per node. Even for our short examples, it produced a proof term exceeding 2 GB that made Coq run out of memory. In other words, it is infeasible to directly construct a proof of $\mathsf{CauseContent}$ in Coq.

We instead used *proof by reflection*: we wrote an executable function that explores all possible behaviors of a client program, then we proved that for any input program, this function will return *true* if and only if there *exists* a proof that the client program is assertion-fail-free. The advantage of this approach is that Coq knows how to run an executable function relatively quickly and without building a giant proof term. Proof by reflection avoids generating the giant proof term because the correctness proof of the function proceeds by induction on any execution (i.e. we only need to consider one step), whereas our first approach considered all steps.

Our abstract semantics (Figure 7) is already executable, so it was easy to write a Coq function that, given a schedule of threads and updates, determines whether the schedule leads to a valid execution (i.e. does not get stuck) and, if so, whether it emits an $\mathsf{assertfail}$. Then, we wrote a function to recursively generate all schedules up to some maximum number of steps M, run the program for each schedule, and return *false* iff the program emits $\mathsf{assertfail}$. A naive implementation may be excessively time consuming. However, a simple optimization was to prune schedules as soon as they generate an invalid step by exploring and executing one step at a time, i.e. to use a breadth-first search of schedules rather than a depth-first search. The final step was to prove that this function returns *true* iff no execution of less than M steps emits an $\mathsf{assertfail}$. Using the "fuel" design pattern, if M is too small for a given program, then the proof of correctness fails. In practice, it is not hard to choose a big enough value of M for terminating programs.

To verify that our example Programs 1 and 2 satisfy $\mathsf{CauseContent}$, we run our checker with $M = 20$ steps of fuel. It takes less than one second to check each. We also verify a third client, listed in Appendix D, that concurrently constructs and traverses a linked list (using a total of 17 operations); it takes just under 8 minutes to check. Finally, we prove corollary theorems that none of these programs will fail when run with $\mathbb{I}_1$ (Figure 14) or $\mathbb{I}_2$ (Figure 15) in our concrete semantics.

7. Experimental Results

In contrast to many verification efforts that work on abstract models of code, our development leads to executable code. In this section, we compare the performance of the two verified key-value store implementations. We extracted the two implementations from Coq to OCaml using Coq's built-in extraction mechanism. We wrote a

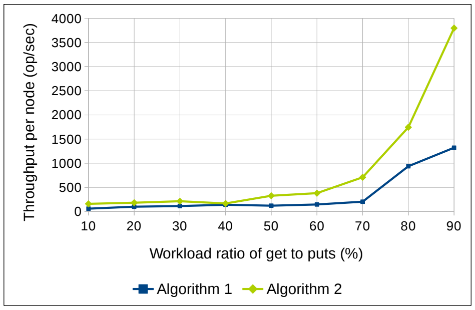

Figure 16. Performance comparison of the key-value stores resulting from the extraction of the two verified implementations.

shim in OCaml that implements network primitives such as UDP message passing, queuing, and event handling. We compiled and linked the extracted implementations together with the shim to obtain executable key-value stores. The trusted computing base of these stores includes the following components: the assumption that the concrete semantics matches the physical network and the shim, Coq's proof checker and OCaml code extractor, and the OCaml compiler, runtime, and libraries.

The experiments are done on a four-node cluster. Each node had an Intel Xeon 2.27GHz CPU with 2GB of RAM and ran Linux Ubuntu 14.04.2 with the kernel version 3.13.0-48-generic #80-Ubuntu. The nodes were connected to a gigabit switch. We used Coq version 8.4pl3, OCaml compiler version 4.01.0, and the OCaml library Batteries version 2.3. All the reported numbers are the arithmetic means of results from five repetitions of the experiment.

We have conducted a simple experiment that measures the throughput (i.e. the number of processed requests per second) of the two stores on a range of workloads (i.e. the get versus put ratio of operations). In every run, each node processed 60,000 random requests with a specific get versus put ratio of operations. Figure 16 presents the throughput of the two stores on workloads ranging from 10% to 90% get operations. We divided the processing time of the last replica that finished its share of requests by the number of its requests to compute the throughput of each node.

As expected, both stores' throughputs increase as the ratio of get operations increases. The second implementation consistently shows a higher throughput than the first implementation. The reason is twofold. Firstly, in the first implementation, the *clock* function of a node keeps an overapproximation of the dependencies of the node. This overapproximation incurs extra dependencies on updates. On the other hand, the second implementation does not require any extra dependencies. Therefore, in the first implementation compared to the second, the updates can have longer waiting times, and the update queues tend to be longer; so the traversal of the update queue is more time-consuming in the first implementation than the second. Secondly, the update payload that is sent and received by the first implementation contains the function *clock*. OCaml cannot *marshal* functions. However, as the *clock* function has the finite domain of the participating nodes, it can be serialized to and deserialized from a list. Nonetheless, serialization and deserialization on every sent and received message add performance cost. Using a finite-map data structure would likely improve the performance. On the other hand, the payload of the second implementation consists of only data types that can be directly marshaled. Therefore, the second implementation has no extra marshalling cost.

8. Related Work

Eventually and causally consistent systems. The consistency and partition-tolerance trade-off formulated by CAP [19] and the consistency and latency trade-off [5] stated by PACELC [3] have motivated industrial storage systems to target relaxed notions of consistency that are collectively called eventual consistency [48]. Eventually consistent key-value stores include Amazon's Dynamo [17], Facebook's Cassandra [27], Yahoo's PNUTS [16], LinkedIn's Project Voldemort [1], and memcached [2]. Cassandra can be configured to provide linearizability by relaxing availability and partition tolerance. PNUTS can provide per-key serializability. Researchers [13, 44, 46] have proposed eventually consistent algorithms for common datatypes like registers, counters, and finite sets. In particular, they have proposed the class of conflict-free replicated data types (CRDTs) with sufficient convergence conditions.

Ahamad and others designed and implemented a distributed causal memory [4]. It has been shown [4, 42] that certain classes of programs exhibit sequential consistency when executed with a causal memory. ISIS [10] and lazy replication [26] support replication methods that provide causal as well as two stronger ordering guarantees for updates. The Bayou system [37, 47] and PRACTI [7] both support partial replication of causally consistent data stores. Recent causally consistent systems include COPS and Eiger [31, 32] and Bolt-On [6]. COPS is a scalable causally consistent key-value store that can provide causal consistency between keys stored across the cluster. It was later extended to Eiger that supports a column-family data model. Bolt-On provides a shim layer on top of eventually consistent stores to provide causal consistency.

Verification and testing of distributed algorithms. Operational semantics [39, 40] and refinement [22] has been successfully applied [11, 43] to model and reason about distributed algorithms. In addition, specification of a distributed algorithm as a state transition system is the common base shared by the I/O Automata [33] and TLA [29] formalisms.

Model checking has been applied to test [23, 24, 36, 50, 51, 53] and synthesize [18] distributed algorithms. On the other hand, theorem provers have been used to establish the absence of bugs. EventML [41] is a domain-specific language for the specification of distributed algorithms. Algorithms specified in EventML can be automatically translated to the Logic of Events [8] and then interactively verified in Nuprl [15]. It has been used [45] to verify a total broadcast service and build serializably consistent replicated databases. A recent effort [25] similar to ours builds a mechanized verification framework but for consensus algorithms and in Isabelle. Verdi [49] models several network semantics with different fault models and provides transformers from correct algorithms in one semantics to another. Recent work [12, 14, 54] has formalized and verified the eventual consistency of replicated objects. To the best of our knowledge, this paper provides the first specification and verification framework that specifically targets causal consistency.

9. Conclusion

We have presented a formal verification framework in Coq called Chapar for causal consistency of distributed key-value stores. The proof structure is composed of an abstract operational semantics for causal consistency, an automatic program verifier for the client programs that execute on causally consistent stores, and a proof technique for causal consistency of store implementations. We have verified a number of store implementations and client programs from the literature. We have extracted and compared the performance of the verified implementations.

Acknowledgments

This research was supported in part by NSF awards CCF-1217501 and CCF-1253229 and by DARPA under agreement number FA8750-12-2-0293.

References

[1] LinkedIn's Voldemort. http://www.project-voldemort.com/.

[2] Memcached. http://memcached.org/.

[3] D. Abadi. Consistency tradeoffs in modern distributed database system design. *Computer*, 45(2), 2012.

[4] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[5] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2), 1994.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proc. SIGMOD*, 2013.

[7] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. NSDI*, 2006.

[8] M. Bickford. Component specification using event classes. In *Component-Based Software Engineering*, volume 5582 of *Lecture Notes in Computer Science*. 2009.

[9] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. SOSP*, 1985.

[10] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1), 1987.

[11] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proc. POPL*, 2006.

[12] A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *Proc. POPL*, 2014.

[13] S. Burckhardt. *Principles of Eventual Consistency*. Foundations and Trends in Programming Languages. 2014.

[14] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Proc. POPL*, 2014.

[15] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-451832-2.

[16] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2), 2008.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, 2007.

[18] A. Gascón and A. Tiwari. Synthesis of a simple self-stabilizing system. In *Proc. 3rd Workshop on Synthesis (SYNT)*, 2014.

[19] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), June 2002.

[20] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proc. HotOS*, 2013.

[21] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proc. ASPLOS*, 2015.

[22] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. ESOP*, 1986.

[23] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proc. FMCAD*, 2013.

[24] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Proc. PLDI*, 2007.

[25] P. Kufner, U. Nestmann, and C. Rickmann. Formal verification of distributed algorithms. In *Theoretical Computer Science*, volume 7604 of *LNCS*. 2012.

[26] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4), 1992.

[27] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[29] L. Lamport. Introduction to TLA. Technical report, 1994.

[30] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16 (2), 1998.

[31] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. SOSP*, 2011.

[32] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. NSDI*, 2013.

[33] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2, 1989.

[34] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, The University of Texas at Austin, 2011.

[35] P. J. Marandi, S. Benz, F. Pedone, and K. P. Birman. The performance of Paxos in the cloud. In *Proc. SRDS*, 2014.

[36] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proc. NSDI*, 2004.

[37] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP*, 1997.

[38] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. PLDI*, 1988.

[39] F. Pfenning and R. J. Simmons. Substructural operational semantics as ordered logic programming. In *Proc. LICS*, 2009.

[40] G. D. Plotkin. The origins of structural operational semantics. In *Proc. Journal of Logic and Algebraic Programming*, pages 60–61, 2004.

[41] V. Rahli. Interfacing with proof assistants for domain specific programming using EventML. 10th International Workshop on User Interfaces for Theorem Provers, 2012.

[42] M. Raynal and A. Schiper. From causal consistency to sequential consistency in shared memory systems. volume 1026 of *LNCS*. 1995.

[43] T. Ridge. Verifying distributed systems: the operational approach. In *Proc. POPL*, 2009.

[44] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

[45] N. Schiper, V. Rahli, R. van Renesse, M. Bickford, and R. Constable. Developing correctly replicated databases using formal tools. In *Proc. DSN*, 2014.

[46] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, 2011.

[47] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, 1995.

[48] W. Vogels. Eventually consistent. *Queue*, 6(6), 2008.

[49] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed system. In *Proc. PLDI*, 2015.

[50] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proc. NSDI*, 2009.

[51] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proc. NSDI*, 2009.

[52] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures:

An analysis of production failures in distributed data-intensive systems. In *Proc. OSDI*, 2014.

[53] P. Zave. Using lightweight modeling to understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2).

[54] P. Zeller, A. Bieniusa, and A. Poetzsch-Heffter. Formal specification and verification of CRDTs. volume 8461 of *LNCS*. 2014.