Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic

LÉON GONDELMAN, Aarhus University, Denmark
SIMON ODDERSHEDE GREGERSEN, Aarhus University, Denmark
ABEL NIETO, Aarhus University, Denmark
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

We present the first specification and verification of an implementation of a causally-consistent distributed database that supports modular verification of full functional correctness properties of clients and servers. We specify and reason about the causally-consistent distributed database in Aneris, a higher-order distributed separation logic for an ML-like programming language with network primitives for programming distributed systems. We demonstrate that our specifications are useful, by proving the correctness of small, but tricky, synthetic examples involving causal dependency and by verifying a session manager library implemented on top of the distributed database. We use Aneris's facilities for modular specification and verification to obtain a highly modular development, where each component is verified in isolation, relying only on the specifications (not the implementations) of other components. We have used the Coq formalization of the Aneris logic to formalize all the results presented in the paper in the Coq proof assistant.

CCS Concepts: • **Theory of computation → Program verification**; **Distributed algorithms**; **Separation logic**.

Additional Key Words and Phrases: Distributed systems, causal consistency, separation logic, higher-order logic, concurrency, formal verification

ACM Reference Format:
Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic. *Proc. ACM Program. Lang.* 5, POPL, Article 42 (January 2021), 29 pages. https://doi.org/10.1145/3434323

1 Introduction

The ubiquitous distributed systems of the present day internet often require highly available and scalable distributed data storage solutions. The CAP theorem [Gilbert and Lynch 2002] states that a distributed database cannot at the same time provide *consistency*, *availability*, and *partition (failure) tolerance*. Hence, many such systems choose to sacrifice aspects of data consistency for the sake of availability and fault tolerance [Bailis et al. 2013; Chang et al. 2008; Lloyd et al. 2011; Tyulenev et al. 2019]. In those systems different replicas of the database may, at the same point in time, observe different, inconsistent data. Among different notions of weaker consistency guarantees, a popular one is *causal consistency*. With causal consistency different replicas can observe different data, yet, it is guaranteed that data are observed in a causally related order: if a node n observes an operation

Authors' addresses: Léon Gondelman, Aarhus University, Denmark, gondelman@cs.au.dk; Simon Oddershede Gregersen, Aarhus University, Denmark, gregersen@cs.au.dk; Abel Nieto, Aarhus University, Denmark, abeln@cs.au.dk; Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2475-1421/2021/1-ART42
https://doi.org/10.1145/3434323

$$\begin{array}{l|l} \mathsf{write}(x, 37) & \mathsf{wait}(y = 1) \\ \mathsf{write}(y, 1) & \mathsf{read}(x) \text{ // reads Some(37)} \end{array} \qquad \begin{array}{l|l|l} \mathsf{write}(x, 0) & \mathsf{wait}(x = 37) & \mathsf{wait}(y = 1) \\ \mathsf{write}(x, 37) & \mathsf{write}(y, 1) & \mathsf{read}(x) \text{ // reads Some(37)} \end{array}$$

Fig. 1. Two examples of causal dependency: direct (left) and indirect (right, [Lloyd et al. 2011]).

x originating at node m, then node n must have also observed the effects of any other operation that took place on node m before x. Causal consistency can, for instance, be used to ensure in a distributed messaging application that a reply to a message is never seen before the message itself.

Two simple, illustrative examples of causal dependency are depicted in Figure 1; programs executed on different nodes are separated using double vertical bars. Notice that in our setting all keys are uninitialized at the beginning and the read operation returns an optional value indicating whether or not the key is initialized. In both examples, the $\mathsf{read}(x)$ command returns the value 37, as indicated by the comment in the code, as the preceding wait command waits for the effects of $\mathsf{write}(y, 1)$ to be propagated. In the example on the left (illustrating direct causal dependence) the $\mathsf{write}(x, 37)$ command immediately precedes the $\mathsf{write}(y, 1)$ command on the same node; hence any node that observes 1 for key y should also observe 37 for key x. However, in the example on the right, the $\mathsf{write}(y, 1)$ command is executed on the middle node *only after* the value of 37 is observed for key x on that node; hence, in this example too, any node that observes 1 for key y should also observe 37 for key x.

Programming distributed systems is challenging and error-prone [Guo et al. 2013], especially in the presence of weaker consistency models like causal consistency which allow concurrent (causally independent) writes [Boehm and Adve 2012]. Consequently, there have been several efforts in recent years to provide tools for analysis and verification of distributed database systems with weak notions of consistency, *e.g.*, [Gotsman et al. 2016; Kaki et al. 2018; Lesani et al. 2016; Xiong et al. 2019]. Those works give a high-level model of a (programming) language with primitives for reading from and writing to a distributed database and provide semantics for that language. The semantics is then used to build sound program analysis tools or to verify correctness of implementations of distributed databases and/or their clients. The semantics presented in the aforementioned works usually keeps track of the history of operations and (directly or indirectly) their dependence graph. A common aspect of these works is that the developed systems, be it a program logic, a program analysis tool, or both, are designed with the express purpose of verifying correctness of closed programs w.r.t. a specific notion of consistency. Thus they do not support general modular verification where components of the program are verified separately, although Lesani et al. do support the verification of databases and their clients independently. Moreover, importantly, the aforementioned works do not scale to verification of full functional correctness of programs and do not scale to a larger setting where the replicated database is just one component of a distributed system.

In this paper, we present the first specification and verification of an implementation of a causally-consistent distributed database that supports modular verification of full functional correctness of clients and servers. In the rest of the Introduction we briefly discuss the implementation, our verification methodology, and the examples of clients that we have verified against our specification.

Implementation, Programming Language, and Program Logic. We implement the pseudo code presented in the seminal paper of Ahamad et al. [1995] for a replicated, distributed database in AnerisLang. AnerisLang [Krogh-Jespersen et al. 2020] is a concurrent (multiple threads on each node) ML-like programming language with network primitives (UDP-like sockets) designed for

programming distributed systems.[1] Our implementation makes use of a heap-allocated dictionary for storing the key-value pairs and uses networking primitives to propagate updates among replicas. Each replica has an *in-queue* and an *out-queue*. On each replica, there are three concurrently running threads: the send thread, the receive thread, and the apply thread. The send thread sends updates from the out-queue (enqueued by the write operation) to other replicas. The receiver thread receives updates from other replicas over the network and enqueues them in the in-queue. The apply thread applies updates from in-queue to the key-value store of the replica.

The operational semantics of AnerisLang is formally defined in the Coq proof assistant together with the Aneris program logic [Krogh-Jespersen et al. 2020]. The Aneris program logic is a *higher-order distributed concurrent separation logic* which facilitates modular specification and verification of partial correctness properties of distributed programs. The Aneris logic itself is defined on top of the Iris program logic framework [Jung et al. 2016, 2018, 2015]. We have used the Aneris logic, Iris, and the Iris proof mode [Krebbers et al. 2018, 2017] to formalize all the results presented in this paper in the Coq proof assistant.

Mathematical Model and Specification. Our Aneris specifications for the distributed database are based on a mathematical model tracking the abstract state of the local key-value stores, *i.e.*, the history of updates. Our specification represents this model using Iris's ghost theory to track auxiliary state (state that is *not* physically present at runtime and only tracked logically for verification purposes). We then use Iris invariants to enforce that the physical state of each replica is *consistent* with the tracked history at all times. We further enforce that the histories tracked by the ghost state are *valid*. We will define validity later; for now it suffices to say that, in our work, viewed at a high level of abstraction, causal consistency is a property of *valid* histories.

The history of updates in our mathematical model consists of the following information:

(1) For each replica, we track a *local history* of all memory updates that the replica has observed since its initialization. It includes both local write operations (which are observed immediately) and updates due to synchronization with other replicas.
(2) We also track an *abstract global memory* that, for each key, keeps track of all write events to that key (by any replica in the system).

We refer to the elements of local histories as *apply events* and to the elements of the abstract global memory as *write events*. Both apply and write events carry all the necessary information about the original update, including the logical time of the corresponding apply or write operation. We model logical time using a certain partial order; see §3 for more details. The ordering is defined such that it reflects causal order: if the time of event e is strictly less than the time of e', then e' *causally depends* on e, and if the time of e and e' are incomparable, then e and e' are *causally independent*. This allows us to formulate the causal consistency of the distributed database as follows:

> *If a node observes an apply event a, it must have already observed all write events of the abstract global memory that happened before (according to logical time) the write event corresponding to a.* (Causal Consistency)

Moreover, we can prove that this property is a consequence of the validity of histories.

[1] AnerisLang as presented in Krogh-Jespersen et al. [2020] features duplicate protection, *i.e.*, every sent message is received at most once. This feature has since been relaxed.

The specifications we give to the read and write operations essentially reflect the behaviors of these operations into the tracked histories. The intuitive reading of our specifications is as follows:[2]

- *Either,* $\mathsf{read}(k)$ *returns nothing, in which case we know that the local history contains no observed events for key* k*.*
- *Or,* $\mathsf{read}(k)$ *returns some value* v*, in which case we know that there is an apply event* a *in the local history with value* v*;* a *has a corresponding write event in the global memory; and* a *is a maximal element (w.r.t. time and hence causality) in the local history.*

(Read Spec)

After the write operation $\mathsf{write}(k, v)$ *there is a new write event* w *added to the global memory and a new apply event* a *corresponding to it in the local history, and* a *is the* maximum *element (w.r.t. time and hence causality) of the local history, i.e., the event* a *causally depends on all other events in the local history.*

(Write Spec)

Note that the specifications above do not say anything about the inter-replica communication. Neither do they refer to some explicit causal relation. They merely assert properties of the history tracked in the ghost state. Indeed, it is our invariants that associate the ghost state of a valid history with the physical states of the replicas that allow us to reason about causal consistency. Crucially, this indirection through histories enables us to use the above specifications modularly. This is essentially because our specifications only refer to the relevant parts of the history, *i.e.*, the global memory for the key in question and the local history for the replica performing the operation.

We present our formal specifications for the read and write operations in §4. However, the specification for write presented in §4, and used throughout most of the paper, is not general enough to support modular Iris-style reasoning about clients because it does not support reasoning about concurrent accesses to keys. The reason is that the write operation is not atomic, as required for Iris-style (and, more generally, concurrent-abstract-predicate-style [Dinsdale-Young et al. 2010]) reasoning using invariants. The read operation is, of course, not atomic either, but the specification for it only involves so-called persistent predicates and hence is general enough. The issue with the write operation is an instance of the known challenge of how to give modular specifications for concurrent modules, see, *e.g.*, [Dinsdale-Young et al. 2018] and [Birkedal and Bizjak 2017, Section 13]. Hence we use one of the solutions to this modularity challenge and present our official specification for the write operation in so-called HOCAP-style [Svendsen et al. 2013], see §7. With our HOCAP-style specification we do indeed support modular reasoning about clients using Iris invariants. (It does take a little while to get used to HOCAP-style specifications; that is why we present the official specification for the write operation relatively late in the paper.) Note however that the specification for the write operation given in §4 is *not* wrong but only weaker than the general HOCAP-style specs given in §7, and can in fact be derived from it. This rule, as we will see in §4, can be used in situations where there are no concurrent accesses to the key being written to.

Clients Verified. We demonstrate the utility of our specifications by verifying a number of interesting examples, including the two examples presented in Figure 1. As a more realistic case study, we implement a *session manager* library that allows clients to communicate with a replica over the network, on top of our distributed database; we use our specifications to prove that the session manager satisfies four *session guarantees* for client-centric consistency [**?**].

Contributions. In summary, we make the following contributions:

[2]Notice that the read operation returns an optional value.

- We present a simple and novel mathematical model of distributed causal memory amenable to building appropriate abstractions for reasoning about implementing and using such memory.
- On top of this model, we define high-level modular specifications for reading from and writing to a causally-consistent distributed database. Our specifications are *node-local* and *thread-local*: in the client's code where multiple threads (possibly on different nodes) interact with the database, all components can be verified separately from each other.
- We show that those high-level specifications are actually met by the original description of a causally-consistent distributed database from the 1995 seminal paper [Ahamad et al. 1995] which we have implemented in a realistic ML-like language with explicit network primitives.
- We show that our specifications provide the expected causality guarantees on standard examples from the literature. Moreover, we implement a session manager library that allows clients and replicas to run on different nodes, and show that our specifications imply the session guarantees for library clients.
- We have formalized all of our results on top of the Aneris Logic in the Coq proof assistant.

Outline. We start by presenting our implementation of a causally-consistent distributed database in AnerisLang in §2. This allows us to match the intuition behind the key ideas of our approach with concrete code. Then, in §3 we present those parts of our model of causality that are needed for client-side reasoning. In §4 we show how to turn the model into abstract program logic predicates and present the specifications of the distributed database operations. We also show how the specifications can be used to reason about the client program examples presented above. In §5 we present a more extensive case study of a client program, a sesssion-manager library, and show how we can use our specifications to reason about session guarantees. In §6 we then prove that the implementation of the distributed database meets our specification. In §7 we present the HOCAP-style specification for the write operation. After discussing related work in §8 we conclude and discuss future work in §9.

2 A Causally-Consistent Distributed Database

In this section we present our AnerisLang implementation[3] of the causally-consistent distributed database described in the seminal paper of Ahamad et al. [1995]. See Figure 2 for the implementation. Conceptually, the implementation can be split into two parts:

- Three operations, `init`, `read`, and `write` allow users respectively to initialize, read from, and write to the distributed database on a particular replica i.
- Three other operations, `send_thread`, `receive_thread`, and `apply` are spawned during the initialization as non-terminating concurrently running threads that propagate updates between initialized replicas and which enforce that every replica applies locally all other replicas' updates in some order that respects causal dependencies.

Because `init` returns to the user a pair of partially applied `read` and `write` functions, the user only needs to supply `read` with a key on which the local store `db` should be read and to supply `write` with a key and value for which the `db` should be updated. The sending, receiving, and apply threads, which are running in a loop concurrently with the user's calls to `read` and `write`, are hidden from the user, who does not need to know about the underlying message-passing implementation. Thus once a replica is initialized, the user will access the local memory on the replica as if they were manipulating locally one global distributed memory.

Each replica has a local heap on which it allocates and further makes use of the following data:

[3] AnerisLang is an ML-like language with a syntax close to OCaml. For readability purposes, the code we show here makes use of some primitive OCaml constructions that are slightly different in AnerisLang (*e.g.*, we write [], :: for lists, which are implemented in AnerisLang using pairs).

```
let init l i =
  let db = ref (dict_empty ()) in
  let t = ref (vc_make (length l) 0) in
  let (iq, oq) = (ref [], ref []) in
  let lock = newlock () in
  let skt = socket () in
  socketbind skt (list_nth l i);
  fork (apply db t lock iq i);
  fork (send_thread i skt lock l oq);
  fork (receive_thread skt lock iq);
  (read db lock, write db t oq lock i)

let read db lock k =
  acquire lock;
  let r = dict_lookup k !db in
  release lock; r

let write db t oq lock i k v =
  acquire lock;
  t := vc_incr !t i;
  db := dict_insert k v !db;
  oq := (k, v, !t, i) :: !oq;
  release lock

let receive_thread skt lock iq =
  let rec aux () =
    let msg = listen_wait skt in
    acquire lock;
    iq := (we_deser msg) :: !iq;
    release lock; aux ()
  in aux ()

let apply db t lock iq i =
  let rec aux () =
    acquire lock;
    (match (find (check !t i) !iq) with
     | Some (w, iq') →
        iq := iq';
        db := dict_insert (π1 w) (π2 w) !db;
        t := vc_incr !t (π4 w)
     | None → ());
    release lock; aux ()
  in aux ()

let check t i w =
  let (wt, wo) = (π3 w, π4 w) in
  let rec aux l r j = match (l, r) with
     | a :: l', b :: r' →
       (if j = wo then a = b + 1 else a ≤ b)
       && aux l' r' (j + 1)
     | [], [] → true
     | _      → false
  in (i != wo) && (wo < length t) && (aux wt t 0)

let send_thread i skt lock l oq =
  let rec aux () =
     acquire lock;
     match !oq with
     | [] → release lock; aux ()
     | w :: oq' →
       oq := oq'; release lock;
       sendToAll skt (we_ser w) i l; aux ()
  in aux ()
```

Fig. 2. Implementation of a causally-consistent distributed database replica.

- a dictionary `db` for storing the key-value pairs to implement causal memory locally;
- a vector clock `t` to timestamp outgoing updates and check dependencies of incoming updates. A vector clock t consists of a vector of n natural numbers, where the j^{th} number $t[j]$ indicates how many updates has been applied locally so far from the replica j;
- an in-queue `iq` and an out-queue `oq` for receiving/sending local updates among replicas;
- a UDP socket `skt` bound to the socket address of the replica;
- and a `lock` to control sharing of above-mentioned data among different concurrent threads.

Vector clocks are the key mechanism by which the implementation enforces that the order in which updates are applied locally on each replica respects the causal order of the entire system. This is enforced in the following way.

To make use of vector clocks, each `write k v` call is associated with a *write event* (k, v, t, i) where projection $t[i]$ describes the number of calls to `write` executed on a replica i (including the current call itself), and all other projections $t[j]$ describe the number of updates received from replica j and applied locally on replica i at the time when the update `write k v` takes place.

When the dictionary `db` and the vector clock `t` are updated by the call `write k v` on a replica i, before the call terminates, it adds the associated write event (k, v, t, i) to the outgoing queue `oq`.

Once the `send_thread` acquires the lock to get access to the queue `oq`, it serializes the write event into a message and broadcasts to all other replicas.

To receive those update messages, each replica runs a `receive_thread` which listens on the socket `skt`, and when it gets a new message `msg` from the replica o, it deserializes it back to a write event $w = (k, v, t, o)$, and adds it to the incoming queue `iq`. As a matter of notation, for a write event $w = (k, v, t, o)$, we write $w.k$ for the key k, $w.v$ for the value v, *etc.*

When the `apply` operation acquires the lock, it consults the incoming queue `iq` in search of a write event that can be applied locally. To this end, it calls the `find (check !t i) !iq` subroutine with the current value of the vector clock `!t` and the index of the replica i. The idea is to search through the queue `iq` until a write event w that passes the test is found (`check !t i w` holds) and retrieved from `iq`, which is the case when the following conditions are satisfied:

(1) The origin $w.o$ of the event w must be different from i, so that w corresponds indeed to an external write operation, and be within bounds $[0, n[$ (recall that `t` is a vector of length n).
(2) For the projection $j = w.o$ (the event w's own origin), the number $w.t[j]$ must be equal to $\texttt{!t}[j] + 1$, which captures the intuition that the event w must be *the most recent write from the replica j that the replica i did not yet observe.*
(3) For all other projections p different from j, the condition $w.t[p] \leq \texttt{!t}[p]$ should hold, which captures the intuition that if the write event w passes the dynamic check, it means that *any memory update on which w causally depends has already been locally applied by the replica i.*

We remark that the pseudo-code in the original paper by Ahamad et al. [1995] requires a reliable network, *e.g.*, that network communication happens using TCP. This is important for showing liveness properties (*e.g.*, every replica eventually applies all messages from other replicas). In this paper, we focus on safety properties and the properties we show (*e.g.*, on any replica, all updates that have been applied are causally consistent) are met by our implementation regardless of whether the network is reliable or not.

3 Mathematical Model

In this section we formalize the key ideas of our mathematical model of causality. Figure 3 shows the model definitions and properties needed to reason about clients.

In the model, a write event is represented much as in the implementation, namely as a four-tuple (k, v, t, o) consisting of a key, a value, the time, and the index of the replica on which the write event happened. In the concrete implementation time is represented using vector clocks, but to reason about client code, all we need is an abstract notion of time, and therefore our model uses a notion of logical time, represented by a partial order $\leq$ (we write $<$ for the strict version of it). We can decide whether two write events w_1 and w_2 are causally related by comparing their times: if $w_1.t < w_2.t$, then w_1 must have *happened before* w_2, and w_2 is *causally dependent* on w_1. When $w_1.t$ and $w_2.t$ are incomparable, then the events w_1 and w_2 are *causally unrelated.*

To account for how write events are applied locally on each replica we use the notion of an *apply event.* Thus an apply event only makes sense in the context of a particular replica. Formally, given a replica i, an *apply event* is represented by a five-tuple $a = (k, v, t, o, m)$, where m is the number of write events applied on replica i. We refer to m as the *sequence identifier* of a. When $i = o$, the apply event corresponds to a write operation invoked on the replica itself, whereas if $i \neq o$, then the apply event corresponds to a write event received from replica i. Given an apply event $a = (k, v, t, o, m)$, we denote by $\lfloor a \rfloor$ the write event (k, v, t, o), which we refer to as the *erasure of a.*

As explained in the Introduction, we keep track of all write and apply events. The *local history* of replica i, written s_i, is the set of all apply events observed by the replica since its initialization.

EVENTS

$(k, v, t, o) \in \mathit{WriteEvent} \triangleq \mathit{Keys} \times \mathsf{Value} \times \mathsf{Time} \times \mathbb{N}$

$(k, v, t, o, m) \in \mathit{ApplyEvent} \triangleq \mathit{Keys} \times \mathsf{Value} \times \mathsf{Time} \times \mathbb{N} \times \mathbb{N}$

$\mathit{Maximals}(X) \triangleq \{x \mid x \in X \land \forall y \in X.\, \neg(x.t < y.t)\}$

$$\mathit{Maximum}(X) \triangleq \begin{cases} \mathsf{Some}(x) & \text{if } x \in X \land \forall y \in X.\, x \neq y \implies y.t < x.t \\ \mathsf{None} & \text{otherwise} \end{cases}$$

$\mathit{Observe} : \mathcal{P}_{\mathrm{fin}}(\mathit{ApplyEvent}) \xrightarrow{\mathrm{fin}} \mathit{ApplyEvent}$

$\lfloor \cdot \rfloor : \mathit{ApplyEvent} \xrightarrow{\mathrm{fin}} \mathit{WriteEvent}$

MEMORY

$s_i \in \mathit{LocalHistory} \triangleq \mathcal{P}_{\mathrm{fin}}(\mathit{ApplyEvent})$

$M \in \mathit{GlobalMemory} \triangleq \mathit{Keys} \xrightarrow{\mathrm{fin}} \mathcal{P}_{\mathrm{fin}}(\mathit{WriteEvent})$

STATES

$\{|\, M ; s_1, \ldots, s_n \,|\} \in \mathit{GlobalState} \triangleq \mathit{GlobalMemory} \times \mathit{LocalHistory} \times \ldots \times \mathit{LocalHistory}$

$\mathit{Valid}_G : \mathit{GlobalState} \to \mathsf{Prop}$

PROPERTIES OF VALID STATES

(Local Extensionality) $\forall a_1, a_2 \in s_i.\, a_1.t = a_2.t \implies a_1 = a_2$

(Global extensionality) $\forall k_1, k_2 \in \mathrm{dom}(M), w_1 \in M(k_1), w_2 \in M(k_2).\, w_1.t = w_2.t \implies w_1 = w_2$

(Causal consistency) $\forall k \in \mathrm{dom}(M), w \in M(k).\, (\exists a \in s_i.\, w.t < a.t) \implies \exists a' \in s_i.\, \lfloor a' \rfloor = w$

(Origin of write events) $\forall k \in \mathrm{dom}(M), w \in M(k).\, \exists i \in \{0..n-1\}, a \in s_i.\, i = w.o \land \lfloor a \rfloor = w$

(Origin of apply events) $\forall a \in s_i.\, \exists k \in \mathrm{dom}(M), w \in M(k).\, \lfloor a \rfloor = w$

Fig. 3. Mathematical model of distributed causal memory with abstract notion of validity.

The *abstract global memory*, written M, is a finite map from keys to finite sets of write events. We model the local key-value store for a replica i simply as a finite map from keys to values.

Given a set X of write or apply events, $\mathrm{Maximals}(X)$ (resp. $\mathrm{Maximum}(X)$) denotes the set of maximal events (resp. the maximum event) w.r.t. the time ordering. Note that, for any events $e, e' \in \mathrm{Maximals}(X)$, the time of e and e' are incomparable and hence e and e' are causally unrelated. Given a non-empty set of apply events A, the event $\mathsf{Observe}(A)$ is the maximum element of A w.r.t. the ordering of sequence identifiers. (If A is empty, we let $\mathsf{Observe}(A)$ be some default apply event).

The global state $\{|\, M ; s_1, \ldots, s_n \,|\}$ consists of the abstract global memory and the local histories of all replicas. Just keeping track of apply and write events is obviously not enough; we also need to make sure that the local histories are always in a consistent state w.r.t. the abstract global memory. This will be expressed using a notion of *validity*. The client need not know the precise definition of validity, but only that there is some predicate Valid_G on global states, and that valid global states satisfy the properties listed in the figure. The local and global extensionality properties express that apply events and write events are uniquely identified by their times. The causal consistency property formalizes the intuitive description of causality from the Introduction. The origin of write property expresses that for every write event there is at least one corresponding apply event on the replica where the write occurred. The origin of apply event property says that every apply event must also be recorded in the abstract global memory.

$$
\begin{array}{rl}
P, Q \in \mathit{iProp} ::= & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \Rightarrow Q \mid P \vee Q \mid \forall x.\ P \mid \exists x.\ P \mid \cdots \quad \text{higher-order logic} \\
& \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_n v \mid \{P\}\ \langle n; e\rangle\ \{x.\ Q\} \mid \Box P \quad \text{separation logic} \\
& \mid \boxed{P}^{\mathcal{N}} \mid {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} \quad \text{Iris resources and invariants} \\
& \mid z \rightarrowtriangle \Phi \mid \mathsf{Fixed}(A) \mid \mathsf{IsNode}(n) \mid \mathsf{FreePorts}(ip, \mathcal{P}) \quad \text{Aneris specific}
\end{array}
$$

Fig. 4. The fragment of Iris and Aneris relevant for this paper.

4 Specification

As discussed in the Introduction, we use the Aneris program logic built on top of the Iris program logic framework for our verification. In this section we present the Aneris specifications of our distributed database operations: read, write, and init. A summary of the specification presented in this section is included in [Gondelman et al. 2020, Appendix A]

In §4.1 we call to mind those aspects of Aneris and Iris that are necessary for following the rest of the paper and introduce the abstract Iris predicates that are provided to clients and used in the specification of the database operations. In §4.2 we present some laws that hold for the abstract predicates and which the client may use for client-side reasoning; the laws are the program logic version of the mathematical model from the previous section. Then we present the specifications for read and write operations in §4.3; these specifications are node local and do not involve any distributed-systems-specific aspects. In §4.4 we explain how the distributed database is initialized and present the specification for the initialization operation; this specification naturally involves distributed-systems-specific aspects. Finally, in §4.5 we give a proof sketch of the client programs from the Introduction.

4.1 Iris, Aneris, Resources, and Tracking the State of the Distributed Database

Figure 4 contains the fragment of the Iris and Aneris logics that is relevant for this paper. Here P and Q range over Iris propositions. We write *iProp* for the universe of Iris propositions. Iris is a higher-order logic that features separation logic primitives: separating conjunction, $*$, and magic wand, $\mathrel{-\!\!*}$, also known as the separating implication. The points-to proposition, $\ell \mapsto_n v$, expresses exclusive ownership of memory location ℓ with value v in the heap of the node n. The *separating* nature of the separating conjunction, and the *exclusive* nature of the points-to propositions can be seen in the rule $\ell \mapsto_n v * \ell' \mapsto_n v' \vdash \ell \neq \ell'$, where $\vdash$ is the entailment relation on Iris propositions. This rule states that separating conjuncts assert ownership over *disjoint* parts of the heap. The Hoare triple $\{P\}\ \langle n; e\rangle\ \{x.\ Q\}$ is used for partial correctness verification of distributed programs. Intuitively, if the Hoare triple $\{P\}\ \langle n; e\rangle\ \{x.\ Q\}$ holds, then whenever the precondition P holds, e is safe to execute *on node* n and whenever e reduces to a value v on node n then that value should satisfy the postcondition $Q[v/x]$; note that x is a binder for the resulting value. The proposition $\boxed{P}^{\mathcal{N}}$ is an Iris invariant: it asserts that the proposition P should hold at all times. The invariant name $\mathcal{N}$ is used for bookkeeping, to prevent the same invariant from being reopened in a nested fashion which is unsound. The update modality, ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$, asserts that Iris resources can be updated in such a way that P would hold. The masks $\mathcal{E}_1$ and $\mathcal{E}_2$ (sets of invariant names) indicate which invariants can be accessed during this update (those in $\mathcal{E}_1$) and which invariants should remain accessible after the update $\mathcal{E}_2$. Whenever both masks of an update modality are the same mask $\mathcal{E}$, which is most often the case, we write $\Rightarrow_{\mathcal{E}}$ instead of ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}}$. We write $P\ {}^{\mathcal{E}_1}\!\mathrel{-\!\!*}^{\mathcal{E}_2} Q$ and $P \mathrel{-\!\!*}_{\mathcal{E}} Q$ as a shorthand for $P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q$ and $P \mathrel{-\!\!*} \Rightarrow_{\mathcal{E}} Q$, respectively. We write $\top$ for the mask that allows access to all invariants. We will explain the Aneris specific propositions later on.

Proposition	Intuitive meaning
$\mathsf{Seen}(i, s)$	The set s is *a causally closed subset of* the local history of replica i
$\mathsf{Snap}(k, h)$	The set h is *a subset of* the global memory for key k
$k \rightharpoonup_u h$	The global memory for key k is *exactly* h

We say s is a *causally closed* subset of s' if: $s \subseteq s' \land \forall a_1, a_2 \in s'.\ a_1.t < a_2.t \land a_2 \in s \Rightarrow a_1 \in s$.

Fig. 5. Propositions to track the state of the key-value store.

Ephemeral Versus Persistent Propositions. Iris, and by extension Aneris, is a logic of resources. That is to say that propositions can assert (exclusive) ownership of resources. In this regards, propositions can be divided into two categories: *ephemeral* propositions and *persistent* propositions. Ephemeral propositions represent transient facts, *i.e.*, facts that later stop from being true. The quintessential ephemeral proposition is the points-to proposition; the value of the memory location ℓ can change by performing a write on ℓ—this can be seen in the specs for writing to a memory location:

$$\{\ell \mapsto_n v\}\ \langle n; \ell \leftarrow w\rangle\ \{x.\ x = () * \ell \mapsto_n w\}$$

One important aspect of ephemeral propositions is that they give us precise information about the state of the program: having $\ell \mapsto_n v$ implies that the value stored in memory location ℓ on node n is v. The upshot of this is that while we own a points-to proposition for a location ℓ, no other concurrently running thread can update the value of ℓ. Hence, to allow concurrent accesses to a location, its points-to proposition should be shared, *e.g.*, using Iris invariants. Persistent propositions, as opposed to ephemeral propositions, express knowledge; these propositions never cease to be true. The persistently modality $\Box$ is used to assert persistence of propositions: $\Box P$ holds, if P holds, and P is persistent. Hence, the logical entailments $\Box P \vdash P$ and $\Box P \vdash \Box\Box P$ hold in Iris. Formally, we say a proposition is persistent if $P \dashv\vdash \Box P$, where $\dashv\vdash$ is the logical equivalence of Iris propositions. Persistent propositions, unlike ephemeral ones, can be freely duplicated, *i.e.*, $\Box P \dashv\vdash \Box P * \Box P$. The quintessential persistent propositions in Iris are Iris invariants. In addition, Hoare triples are also defined to always be persistent. This intuitively means that all the requirements for the correctness of the program with respect to the postcondition are properly captured by the precondition.

Iris Predicates to Represent the State of the Key-Value Store. Recall the intuitive specifications that we gave for the read and write operations on our distributed database in the Introduction. These specs only assert that certain write/apply events are added to the global memory/local history. Hence, it suffices to have a persistent proposition in the logic that asserts the partial information that certain events are indeed part of the local history or global memory. For this purpose, we introduce the persistent abstract predicates Seen and Snap which intuitively assert knowledge of a subset of the local history, and global memory, respectively. These abstract predicates and their intuitive meaning are presented in Figure 5. Notice that the Seen predicates assert knowledge of a subset of the local history that is *causally closed* as defined in the figure. We will discuss the significance of causal closure later. In addition to the partial knowledge about the global memory represented using the Snap predicate, it is also useful to track the precise contents of the global memory for each key—see the example presented in §4.5. We do this using the ephemeral abstract proposition $k \rightharpoonup_u h$ which, intuitively, asserts that the set of *all* write events for the key k is h. We can track the precise contents of the global memory because all write events in the global memory can only originate from a write operation on the distributed database. On the other hand, we cannot have precise knowledge about local histories because at any point in time, due to the concurrent execution of replica's apply function, a replica may observe new events.

Properties of global memory, *i.e.*, Snap and $\rightharpoonup_u$ predicates:

$$\mathsf{Snap}(k,h) * \mathsf{Snap}(k,h') \vdash \mathsf{Snap}(k, h \cup h') \qquad \text{(Snap union)}$$

$$k \rightharpoonup_u h \vdash k \rightharpoonup_u h * \mathsf{Snap}(k,h) \qquad \text{(Take Snap)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * k \rightharpoonup_u h * \mathsf{Snap}(k,h') \vdash \Rrightarrow_{\mathcal{E}} k \rightharpoonup_u h * h' \subseteq h \qquad \text{(Snap inclusion)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Snap}(k,h) * \mathsf{Snap}(k,h') \vdash \Rrightarrow_{\mathcal{E}} \forall w \in h, w' \in h'. \; w.t = w'.t \Rightarrow w = w' \qquad \text{(Snap extensionality)}$$

Properties of local histories, *i.e.*, the Seen predicate:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * \mathsf{Seen}(i,s') \vdash \Rrightarrow_{\mathcal{E}} \mathsf{Seen}(i, s \cup s') \qquad \text{(Seen union)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * \mathsf{Seen}(i',s') \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, a' \in s'. \; a.t = a'.t \Rightarrow a.k = a'.k \wedge a.v = a'.v \qquad \text{(Seen global extensionality)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * \mathsf{Seen}(i,s') \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, a' \in s'. \; a.t = a'.t \Rightarrow a = a' \qquad \text{(Seen local extensionality)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * a \in s \vdash \Rrightarrow_{\mathcal{E}} \exists h. \; \mathsf{Snap}(a.k, h) * \lfloor a \rfloor \in h \qquad \text{(Seen provenance)}$$

Causality in terms of resources and predicates:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * \mathsf{Snap}(k,h) \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, w \in h. \; w.t < a.t \Rightarrow \exists a' \in s_{|k}. \; \lfloor a' \rfloor = w \qquad \text{(Causality)}$$

The set $s_{|k}$ is the set of apply events in s with key k: $s_{|k} \triangleq \{a \in s \mid a.k = k\}$.

Fig. 6. Laws governing database resources. The mask $\mathcal{E}$ is any arbitrary mask that includes $\mathcal{N}_{\mathsf{GI}}$.

In addition to the abstract predicates just discussed, the client will also get access to a global invariant $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$ which, intuitively, asserts that there is a valid global state, and that the predicates Seen, Snap, and $\rightharpoonup_u$ track this global state.[4] Clients need not know the definition of this invariant and can just treat it as an abstract predicate.

(For Iris experts we remark that the abstract predicates in Figure 5 are all timeless, which simplifies client-side reasoning [Jung et al. 2018]).

4.2 Laws Governing Database Resources

The laws governing the predicates Seen, Snap, and $\rightharpoonup_u$, are presented in Figure 6. The laws presented in this figure, with the exception of one law that we will discuss in §7, are *all* the laws that are necessary for client-side reasoning about our distributed database. Notice that most of these laws only hold under the assumption that the global invariant holds. This can also be seen in the fact that they are expressed in terms of an update modality with a mask that enables access to the global invariant. All of these laws make intuitive sense based on the intuitive understanding of the predicates Seen, Snap, and $\rightharpoonup_u$. For instance, the law (Snap union) asserts that if we know that the sets h and h' are both subsets of the global memory for a key k, then so must the set $h \cup h'$. The extensionality laws essentially state that events are uniquely identifiable with their time: if two events have the same time, then they are the same event. The only caveat is that in case of the law

[4] $\mathcal{N}_{GI}$ is a fixed name of the global invariant.

(Seen global extensionality): if two apply events *on two different replicas* have the same time, then they must agree on their key and value, but not on their sequence identifiers which represent the order in which events are applied locally on the replica. Note that the law (Seen union), as opposed to the law (Snap union), requires access to the invariant. This is because, we need to establish causal closure (see Figure 5) for $s \cup s'$ in the conclusion of the law with respect to the local history tracked in the global invariant. The most important law in Figure 6 is the law (Causality). This law allows us to reason about causality: if a replica i has observed an event a that has a time greater than a write event w, w causally depends on a, then replica i must have also observed w (it must have a corresponding apply event a'). Notice that the *causal closure* property of local histories s for which we have $\mathsf{Seen}(i, s)$ in Figure 5 is crucial for the (Causality) law to hold.

4.3 Specs for the Read and Write Operations

The Fig. 7 shows the specification for reading from and writing to the distributed database locally on a replica i.

Read Specification. The post condition of the read operation states formally, in the language or Aneris logic, the intuitive explanation that we described in the Introduction. It asserts that the client gets back a set of apply events s', $\mathsf{Seen}(i, s')$, observed by replica i performing the read operation such that $s' \supseteq s$. The reason for the $s' \supseteq s$ relation is that during the time since performing the last operation by replica i, *i.e.*, when we had observed the set s, some write events from other replicas may have been applied locally.

When $\mathsf{read}(k)$ is executed on a replica i, it either returns None or $\mathsf{Some}(v)$ for a value v. If it returns None, then the local memory does not contain any values for key k. Hence the local history s' restricted to key k, ${s'}_{|k}$ should be empty *cf.* the definition of ${s'}_{|k}$ in Figure 6.

Otherwise, if it returns $\mathsf{Some}(v)$, then the local memory contains the value v for key k. This can happen only if the local memory of the replica at the key k has been updated, and the latest update for that key has written the value v. Consequently, the local history s' must have recorded this update as *the latest apply event* a for the key k, *i.e.*, $\mathsf{Observe}({s'}_{|k}) = a$. Hence $a \in \mathrm{Maximals}({s'}_{|k})$. The reader may wonder why a is not the maximum element, but only in the set of maximal elements. To see why, suppose that just before the read operation was executed, two external causally-unrelated writes have been applied locally on replica i, so that the local history recorded them as two distinct apply events whose times are incomparable. Naturally, one of two writes must have been applied before the other and the latest observed apply event *must* correspond to the subsequent second write event. However, as the apply operation is hidden from the client, there is no way for the client to observe which of the two writes is the latest. Consequently, all the client can know is that the latest observed event a is *one of the most recent* local updates for key k, *i.e.*, among the maximal elements. Naturally, the write event $\lfloor a \rfloor$ should be in the abstract global memory. This is expressed logically by the proposition $\mathsf{Snap}(k, \{\lfloor a \rfloor\})$.

Write Specification. The postcondition of the write specification ensures that after the execution of $\mathsf{write}(k, v)$, the client gets back the resources $k \rightharpoonup_u h \uplus \{\lfloor a \rfloor\}$ and $\mathsf{Seen}(i, s' \uplus \{a\})$, where a and $\lfloor a \rfloor$ are respectively the apply and write events that model the effect of the write operation. The mathematical operation $\uplus$ is the disjoint union operation on sets; $A \uplus B$ is undefined if $A \cap B \neq \emptyset$.

As for read, the new set of apply events s' can be a superset of s. Contrary to read, the postcondition for write states that $a = \mathrm{Maximum}(s' \uplus a)$, *i.e.*, that a is actually *the most recent* apply event. This matches the intuition that the update $\mathsf{write}(k, v)$ causally depends on any other apply event previously observed at this replica.

While a is the maximum apply event locally, its erasure $\lfloor a \rfloor$ is only guaranteed to be among the maximal write events, *i.e.*, $\lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\})$. Intuitively, this is because there can

ReadSpec
$\{\mathsf{Seen}(i, s)\}$

$\langle ip_i;\ \mathsf{read}(k)\rangle$

$$\left\{\begin{array}{l} v.\ \exists s' \supseteq s.\ \mathsf{Seen}(i, s') * \\ \quad \Big((v = \mathsf{None} \land s'_{|k} = \emptyset) \lor \\ \quad (\exists a \in s'_{|k}.\ v = \mathsf{Some}(a.v) * \mathsf{Snap}(k, \{\lfloor a \rfloor\}) * a \in \mathrm{Maximals}(s'_{|k}) * \mathsf{Observe}(s'_{|k}) = a)\Big) \end{array}\right\}$$

WriteSpec
$\{\mathsf{Seen}(i, s) * k \rightharpoonup_u h\}$

$\langle ip_i;\ \mathsf{write}(k, v)\rangle$

$$\left\{\begin{array}{l} (). \ \exists s' \supseteq s.\ \exists a.\ k = a.k * v = a.v * \mathsf{Seen}(i, s' \uplus \{a\}) * k \rightharpoonup_u h \uplus \{\lfloor a \rfloor\} * \\ \quad a = \mathrm{Maximum}(s' \uplus a) * \lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\}) \end{array}\right\}$$

Fig. 7. Read and write specifications.

InitSpec
$$\left\{\begin{array}{l} \mathsf{initToken}(i) * \mathsf{Fixed}(A) * \mathsf{IsNode}(ip_i) * \mathsf{Addrlist}[i] = (ip_i, p) * \\ (ip_i, p) \in A * \mathit{isList}(\mathsf{Addrlist}, v) * \mathsf{FreePorts}(ip_i, \{p\}) * \mathop{\Large *}_{z \in \mathsf{Addrlist}} z \Longmapsto \Phi_{\mathsf{DB}} \end{array}\right\}$$

$\langle ip_i;\ \mathsf{init}(i, v)\rangle$

$\{(\mathsf{rd}, \mathsf{wr}).\ \ \mathsf{Seen}(i, \emptyset) * \mathsf{readSpec}(\mathsf{rd}, i) * \mathsf{writeSpec}(\mathsf{wr}, i)\}$

Fig. 8. Specification for init.

be other write events in h, performed by other replicas, that we have not yet locally observed. As those events are not observed on our replica, the newly added write event $\lfloor a \rfloor$ does not causally depend on them and hence does not have a strictly greater time—in practice those write events have times that are incomparable to that of $\lfloor a \rfloor$ as neither depend on the other.

4.4 Initializing the Distributed Database

Our distributed database must be initialized before it is used. Initialization takes place in two phases:

(1) Initialization of resources and establishing the global invariant. This phase is a *logical* phase, *i.e.*, it does not correspond to any program code.
(2) Initialization of the replica. This phase corresponds to the execution of the init function—see Figure 2.

Importantly, in the spirit of modular verification, our methodology for verifying client programs is to *assume* that the library is initialized when we verify parts of the client program that interact with the read and write functions. We only later *compose* these proofs with the proof corresponding to the initialization of the system—we have indeed followed this methodology in verifying all the examples discussed in this paper. Below we discuss initialization, starting with phase 2.

The Specification of the init **Function (Phase 2).** Figure 8 shows the specification for the init function. The postcondition states that the new replica has not observed any events, *i.e.*, $\mathsf{Seen}(i, \emptyset)$. Furthermore, the init function returns a pair of functions that satisfy, respectively, the read and write specifications discussed earlier. This is formally written as $\mathsf{readSpec}(rd, i)$, and $\mathsf{writeSpec}(wr, i)$, respectively. The precondition of the init function is slightly more elaborate. Addrlist is the list of socket addresses (pairs of an ip address and a port) of all replicas of the database, *including* the

replica being initialized. Hence, the i^{th} element of the list should be the socket address (ip_i, p) which this replica uses for communication with other replicas; this is indicated in the precondition by the assertion $\mathsf{Addrlist}[i] = (ip_i, p)$. The predicate $isList(\mathsf{Addrlist}, v)$ asserts that the AnerisLang value v is a list of values corresponding to the mathematical list Addrlist. The init function requires an *initialization token*, $\mathsf{initToken}(i)$; initialization tokens are produced by the first initialization phase. Distributed systems modeled in Aneris always have a distinguished set A of fixed socket addresses, written using the persistent proposition $\mathsf{Fixed}(A)$. The socket address (ip_i, p) used by replica i should be a fixed address. Moreover, on the ip address ip_i, the port p must be free (*i.e.*, must not have any socket bound to it), as indicated by the ephemeral proposition $\mathsf{FreePorts}(ip_i, \{p\})$. In Aneris, fixed socket addresses, as opposed to dynamic socket addresses, are those that have globally fixed so-called socket protocols (explained in the following). The persistent Aneris proposition $\mathsf{IsNode}(ip_i)$ asserts that the node ip_i is a valid node, *i.e.*, all networking resources have been initialized. Finally, the precondition of the init function requires the knowledge that all socket addresses participating in the distributed database follow the same socket protocol Φ_{DB}. This is written as the persistent Aneris proposition $z \Rrightarrow \Phi_{\mathsf{DB}}$. In Aneris, a socket protocol is simply a predicate over messages which restricts what messages can be sent over and/or may be received through a socket. The protocol Φ_{DB} asserts that any message sent over the socket is the serialization of a write event $w = (k, v, t, o)$ that has been recorded in the abstract global memory, *i.e.*, for which $\mathsf{Snap}(k, \{w\})$ holds.

Phase 1. The purely logical nature of the first phase can be seen in the fact that it is expressed in terms of an update modality:

InitSetup

$$\mathsf{True} \vdash \Rrightarrow_{\mathcal{E}} \boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \left(\mathop{\Large *}_{0 \le i < length(\mathsf{Addrlist})} \mathsf{initToken}(i) \right) * \left(\mathop{\Large *}_{k \in Keys} k \rightarrow_u \varnothing \right) * \mathsf{initSpec}(\mathsf{init})$$

The InitSetup rule simply asserts that resources can be allocated so as to initialize the abstract global memory with an empty set for all keys (*i.e.*, elements of *Keys*), to establish the global invariant, and to provide initialization tokens for all declared replicas (*i.e.*, elements of Addrlist). Furthermore, after the InitSetup update is performed, we have that the init function meets the specification given in Figure 8, formally written as $\mathsf{initSpec}(\mathsf{init})$.

4.5 Client Reasoning about Causality

To illustrate how clients can reason about interactions with the distributed database, we give a proof sketch for the example of direct causal dependency from Figure 1. The core part of the proof is sketched in Figure 9 and assumes local replicas have already been properly initialized, that the specifications from Figure 7 hold for the read and write functions, and that writing is an atomic operation. Afterwards, we will show how to initialize local replicas and compose the distributed system. Notice that in this example, while y is being accessed concurrently, x is *not*; reading x on the right happens after the write to y on the left, and hence also after the write to x. Therefore, we use the rule WriteSpec to reason about $\mathsf{write}(x, 37)$ while we use the HOCAP-style specification (which we present in §7) for reasoning about $\mathsf{write}(y, 1)$; see the accompanying Coq formalization for the full formal proof. [Gondelman et al. 2020, Appendix B] provides a proof sketch of the example of indirect causal dependency from Figure 1.

We remark that the proof of the example is quite similar in structure to the proof of a similar message-passing example, *but for a weak memory model* with release-aquire and non-atomic accesses [Kaiser et al. 2017]; see §8 for a comparison to this related work.

For our presentation here, as a convention, we will only mention persistent assertions (such as invariants and equalities) once and use them freely later.

Both nodes operate on the key y concurrently so ownership of y is put into an invariant $\boxed{\mathsf{Inv}_y}$. The invariant essentially says that y is in one of two states: either 1 has been written to y or not. It will be the responsibility of node i to write 1 and advance the state. When node j reads 1, it will expect to be able to gain ownership of key x. Thus when writing 1 to the key y, node i will have to establish another invariant $\boxed{\mathsf{Inv}_x(w')}$ about x, for some write event w' that has happened *before* the write of 1 to y.

The invariant $\boxed{\mathsf{Inv}_x(w')}$ asserts either ownership of x, and that the maximum event is w' with value 37, or a token $\boxed{\diamond}$. The token is a uniquely ownable piece of ghost state (*i.e.*, $\boxed{\diamond} * \boxed{\diamond} \vdash \mathsf{False}$). Intuitively, when the first node establishes the invariant, the ownership of x is transferred into the invariant. The unique token is given to the second node and when it learns of the existence of $\boxed{\mathsf{Inv}_x(w')}$ it can safely swap out the token for the ownership of x.

Node i initially has knowledge of its local history s and ownership of key x with history h_x such that $h_x \subseteq \lfloor s \rfloor$ where $\lfloor s \rfloor \triangleq \{\lfloor a \rfloor \mid a \in s\}$. Intuitively, this means that all writes to x have been observed at node i and that any future writes to x will be causally dependent on these. Using the specification for the write function (*cf.* Figure 7), we obtain updated resources for the local history and the key x, *i.e.*, $\mathsf{Seen}(i, s' \uplus \{a_x\})$ and $x \rightharpoonup_u (h_x \uplus \{\lfloor a_x \rfloor\})$, such that $\mathrm{Maximum}(s' \uplus \{a_x\}) = \mathsf{Some}(a_x)$. From Snap extensionality, Seen global extensionality, and the definition of $\mathrm{Maximum}$ we conclude $\mathrm{Maximum}(h_x \uplus \{\lfloor a_x \rfloor\}) = \mathsf{Some}(\lfloor a_x \rfloor)$, which suffices for establishing the invariant for x, $\boxed{\mathsf{Inv}_x(\lfloor a_x \rfloor)}$. We then open the invariant for y in order to write 1 to y (this is where we cheat in this sketch and use the assumption that write is atomic); using the invariant we just established for x it is straightforward to reestablish the invariant for y after writing 1 to y as $\lfloor a_x \rfloor.t < \lfloor a_y \rfloor.t$ follows from $a_x \in s''$ and a_y being the maximum of $s'' \uplus \{a_y\}$, *cf.* global extensionality of Seen.

Node j initially has knowledge of its local history s and ownership of the token $\boxed{\diamond}$. After repeatedly reading y until we read 1 (call to the wait function[5]), the specification for the read function gives us $\mathsf{Snap}(y, \{\lfloor a_y \rfloor\})$ such that $a_y.v = 1$. By Snap inclusion and by opening the invariant for y we get $\boxed{\mathsf{Inv}_x(w_x)}$ such that $w_x.t < \lfloor a_y \rfloor.t$. We can now open the invariant for x and swap out the token for the ownership of key x and knowledge about its maximum write event. Intuitively, due to causality, as $w_x.t < \lfloor a_y \rfloor.t$ and a_y has been observed, we are guaranteed to read *something* when reading from x; as $\mathrm{Maximum}(h) = \mathsf{Some}(w_x)$ and $w_x.v = 37$ the value we read has to be 37. Formally, this argument follows from extensionality of Seen and Snap, Snap inclusion, and the definition of $\mathrm{Maximum}$ and $\mathrm{Maximals}$—we refer to the Coq formalization for all the details.

The proof sketched in Figure 9 verifies the two nodes individually, assuming local replicas have been properly initialized. To set up a distributed system, we spawn two nodes that each initialize a local replica using the init function:

$$\begin{array}{l} \texttt{let}\ (\mathsf{read}, \mathsf{write}) = \mathsf{init}(i, ips)\ \texttt{in} \\ \mathsf{write}(x, 37);\ \mathsf{write}(y, 1) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{let}\ (\mathsf{read}, \mathsf{write}) = \mathsf{init}(j, ips)\ \texttt{in} \\ \mathsf{wait}(y = 1);\ \mathsf{read}(x) \end{array}$$

where *ips* is the list of ip addresses of the participating replicas. The proof of the combined system follows from the specification for the init function (*cf.* Figure 8) and the proof sketch just given. In particular, the specification of init ensures that both the history for x and the history for y are empty and hence the precondition for node i holds.[6]

[5] The $\mathsf{wait}(k = n)$ operation is just a simple loop that repeatedly reads k until the read value is n. In particular, there are no locks/other synchronization code in the wait implementation.

[6] In AnerisLang there is a distinguished system node, which starts all the nodes in the distributed system. Technically, phase 1 of the initialization happens in the proof of the system node, which is then composed with the proof sketch given above for the two nodes.

Invariants

$$\mathsf{Inv}_x(w) \triangleq (\exists h.\, x \rightharpoonup_u h * \mathrm{Maximum}(h) = \mathsf{Some}(w) * w.v = 37) \vee \boxed{\diamond}$$
$$\mathsf{Inv}_y \triangleq \exists h.\, y \rightharpoonup_u h * \forall w \in h.\, w.v = 1 \Rightarrow \exists w'.\, w'.t < w.t * \boxed{\mathsf{Inv}_x(w')}$$

Node i, proof outline

$$\{\mathsf{Seen}(i, s) * x \rightharpoonup_u h_x * h_x \subseteq \lfloor s \rfloor * \boxed{\mathsf{Inv}_y}\}$$
$$\mathsf{write}(x, 37)$$
$$\left\{\begin{array}{l}\exists a_x, s' \supseteq s.\, \mathsf{Seen}(i, s' \uplus \{a_x\}) * x \rightharpoonup_u (h_x \uplus \{\lfloor a_x \rfloor\}) * \\ \mathrm{Maximum}(s' \uplus \{a_x\}) = \mathsf{Some}(a_x) * a_x.v = 37\end{array}\right\}$$
$$\{\mathsf{Seen}(i, s' \uplus \{a_x\}) * \boxed{\mathsf{Inv}_x(\lfloor a_x \rfloor)}\}$$
$$\left|\begin{array}{l}\text{open } \mathsf{Inv}_y \\ \{\mathsf{Seen}(i, s' \uplus \{a_x\}) * y \rightharpoonup_u h_y * \ldots\} \\ \quad \mathsf{write}(y, 1) \\ \left\{\begin{array}{l}\exists a_y, s'' \supseteq s' \uplus \{a_x\}\,.\, \mathsf{Seen}(i, s'' \uplus \{a_y\}) * y \rightharpoonup_u (h_y \uplus \{\lfloor a_y \rfloor\}) * \\ a_y.v = 1 * \mathrm{Maximum}(s'' \uplus \{a_y\}) = \mathsf{Some}(a_y)\end{array}\right\}\end{array}\right.$$
$$\{\mathsf{Seen}(i, s'' \uplus \{a_y\}) * \boxed{\mathsf{Inv}_y}\}$$

Node j, proof outline

$$\{\mathsf{Seen}(j, s) * \boxed{\mathsf{Inv}_y} * \boxed{\diamond}\}$$
$$\mathsf{wait}(y = 1)$$
$$\{\exists s' \supseteq s, a_y \in s', w_x.\, \mathsf{Seen}(j, s') * \boxed{\mathsf{Inv}_x(w_x)} * \boxed{\diamond} * w_x.t < \lfloor a_y \rfloor.t\}$$
$$\{\mathsf{Seen}(j, s') * x \rightharpoonup_u h_x * \mathrm{Maximum}(h_x) = \mathsf{Some}(w_x) * w_x.v = 37\}$$
$$\mathsf{read}(x)$$
$$\{v.\, \exists s'' \supseteq s'.\, \mathsf{Seen}(j, s'') * x \rightharpoonup_u h_x * v = \mathsf{Some}(37)\}$$

Fig. 9. Proof sketch, example with direct causal dependency.

Now we have a complete proof of the client program, under the assumption that the specifications for the distributed database operations hold. By combining this proof with the proof of the implementation (in §6) we get a closed correctness proof of the whole distributed system in Aneris. This means that we can apply the adequacy theorem of Aneris [Krogh-Jespersen et al. 2020, Section 4.2] to conclude that the whole system is safe, *i.e.*, that nodes and threads cannot get stuck (crash) in the operational semantics. (Safety is enough to capture the intuitive desired property for this example: if we included an assert statement after the read of x that would crash if the return value is not 37, then the adequacy theorem would guarantee that this would never happen. We have included such an assert statement in the Coq formalization.)

5 Case Study: Towards Session Guarantees for Client-Centric Consistency

In the examples in the Introduction and §4.5, each client program is co-located on the *same* node as the database replica that it reads from and writes to. By contrast, in a *client-server* architecture, a client might interact with *multiple* replicas (servers), and clients and replicas are located on *different* nodes.

The client-server setting is interesting for at least two reasons. First, it is the prevalent mode of use of databases within cloud computing (*e.g.*, MongoDB [Chodorow and Dirolf 2010] and

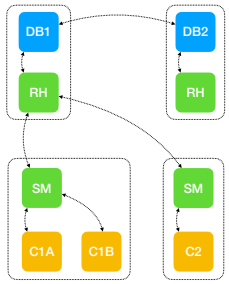

Fig. 10. Clients using the distributed database via the session manager library.

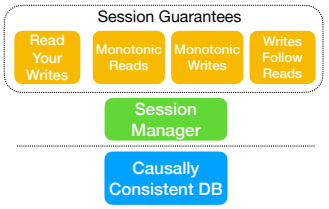

Fig. 11. Vertical compositionality of specifications.

DynamoDB [Sivasubramanian 2012]), where client applications transparently interact with a geo-replicated database running in the cloud. Second, there are consistency models that are tailored to the client-server setting [Tanenbaum and van Steen 2007]. In particular, *session*[7] *guarantees* (*read your writes*, *monotonic reads*, *monotonic writes*, and *writes follow reads*) [**?**] describe properties that programmers can use to reason about client-server interactions. For example, the monotonic writes (MW) guarantee ensures that writes happening within a session are propagated to *all* replicas *in program order*.

In this section, we show that our distributed database can be used *in a client-server setting*. Specifically, we build a *session manager library* that exposes the database's read and write operations to external clients. The library consists of two components: a client *stub* that proxies requests to the server, and a *request handler* that handles the requests server-side. Figure 10 illustrates how clients (C1A, C1B, and C2) running on different nodes communicate with multiple database replicas (DB1 and DB2) via the session manager stub (SM) and request handler (RH).

We give specifications for the session manager library that rely exclusively on *persistent* resources (as opposed to the database specifications in Figure 7, which use the *exclusive* ownership of the global memory predicate $k \rightharpoonup_u h$). This is important because multiple clients could be interacting with the same replica concurrently (and from different nodes) in an uncoordinated way.

In spite of being weaker than the underlying replicated database specifications, our session manager specifications are strong enough to prove versions of the four previously-mentioned *session guarantees*. This result is in line with prior work showing that, at the model level, causal consistency implies all four guarantees [Brzezinski et al. 2004]. In our case, we are able to establish this connection while reasoning about concrete programs.

We illustrate the guarantees with four examples that use the session manager library. In this way, the case study additionally illustrates the modularity of our approach, in particular the *vertical composability* of our specifications, cf. Figure 11: we are able to verify each layer using only the *specifications* of the layer below.

Session Manager Library. As previously mentioned, the session manager exposes the database's operations to the network. The client stub provides its user with three operations: sconnect, sread, and swrite. The client calls sconnect ip_i to start interacting with the replica located at ip_i. Reading returns an option with the retrieved value, or None if the key is not populated. All three operations are synchronous at the client-side and every call is blocking while waiting for a server to reply.

[7] A *session* is a consecutive sequence of reads and writes issued by a particular client.

$$
\begin{array}{lll}
\{\top\} & \langle ip_{client};\ \mathsf{sconnect}(ip_i)\rangle & \{\exists s.\mathsf{Seen}(i,s) * \mathop{\Large *}_{k\in\mathsf{Keys}} \exists h_k.\mathsf{Snap}(k,h_k)\} \\
\{\mathsf{Seen}(i,s) * \mathsf{Snap}(k,h)\} & \langle ip_{client};\ \mathsf{sread}(ip_i,k)\rangle & \{\exists s' \supseteq s, h' \supseteq h.\mathsf{Seen}(i,s') * \mathsf{Snap}(k,h') * \ldots\} \\
\{\mathsf{Seen}(i,s) * \mathsf{Snap}(k,h)\} & \langle ip_{client};\ \mathsf{swrite}(ip_i,k,v)\rangle & \{\exists s' \supset s, h' \supset h.\mathsf{Seen}(i,s') * \mathsf{Snap}(k,h') * \ldots\}
\end{array}
$$

Fig. 12. Simplified specifications of session manager operations. Full specifications are found in [Gondelman et al. 2020, Appendix C].

Guarantee	**Program**	**Description**
read your writes	swrite(ip,k,v); sread(ip,k)	Reads observe writes not older than preceding writes.
monotonic reads	sread(ip,k); sread(ip,k)	Reads observe writes not older than writes observed by preceding reads.
monotonic writes	swrite(ip,k1,v1); swrite(ip,k2,v2)	Writes propagate to all replicas in program order.
writes follow reads	sread(ip,k1); swrite(ip,k2,v)	Writes and writes observed through reads propagate to all replicas in program order.

Table 1. The four session guarantees.

$$
\begin{array}{l}
\textsc{SM-monotonic-writes} \\
\{ip_i \Rightarrow \Phi_i\} \\
\quad \langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{swrite}(ip_i,k_1,v_1); \mathsf{swrite}(ip_i,k_2,v_2)\rangle \\
\left\{\begin{array}{l}
(). \exists s_1, a_1, a_2.\, a_1.k = k_1 * a_1.v = v_1 * a_2.k = k_2 * a_2.v = v_2 \\
\quad * \mathsf{Seen}(i,s_1) * a_1, a_2 \in s_1 * a_1.t < a_2.t \\
\quad * (\forall a, s', j.\mathsf{Seen}(j,s') * a \in s' * a_2.t \le a.t \\
\qquad \Rrightarrow_\top \exists a_1', a_2'.\, \lfloor a_1' \rfloor = \lfloor a_1 \rfloor * \lfloor a_2' \rfloor = \lfloor a_2 \rfloor * a_1', a_2' \in s' * a_1'.t < a_2'.t)
\end{array}\right\}
\end{array}
$$

Fig. 13. Specification for monotonic writes example.

Specifications. Figure 12 presents a high-level view of the session manager specifications, focusing on how the resources are updated; the full specifications are found in [Gondelman et al. 2020, Appendix C]. The client, located on the network node at address ip_{client}, reasons about session manager operations using the snapshot predicates $\mathsf{Seen}(i,s)$ and $\mathsf{Snap}(k,h)$ for local and global histories, respectively. For example, to reason about the write $\mathsf{swrite}(ip_i,k,v)$, the user provides $\mathsf{Seen}(i,s)$ and $\mathsf{Snap}(k,h)$. Once the write operation completes (is processed by the server and a reply is received), the user gets back *updated* snapshots $\mathsf{Seen}(i,s')$ and $\mathsf{Snap}(k,h')$, such that $s \subseteq s'$, $h \subseteq h'$, and the written value v is stored in an apply (resp. write) event that is part of s' (resp. h'). This captures the idea that if we know that a replica observed *at least* a set s of writes, then after we write v the replica will have observed at least the set $s' = s \uplus \{a\}$, where $a.v = v$. We can then reuse the updated snapshots in the precondition of subsequent operations (we get the initial snapshots from the postcondition of sconnect).

Session Guarantees. Table 1 shows the four session guarantees and corresponding client programs we verify. Each guarantee describes what a client can infer from observing the effect of a pair of (read or write) operations within the same session. For space reasons we only describe our *monotonic writes* (MW) example in detail. See [Gondelman et al. 2020, Appendix C] or the Coq formalization for the others.

Figure 13 shows a simplified specification for the MW example (we omit some network-related predicates from the precondition), which involves two consecutive writes to a replica located at address ip_i. The precondition for the Hoare triple is just knowledge that address ip_i behaves according to socket protocol Φ_i. The definition of this socket protocol is a key part of verifying the session manager library, since it allows us to tie physical client requests to their logical counterparts, but is relegated to the Coq formalization for space reasons. Let us now unpack the postcondition. We obtain a snapshot $\mathsf{Seen}(i, s)$ that represents the replica state after the two writes. Both writes are recorded in the local history s through apply events a_1 and a_2 that respect program order ($a_1.t < a_2.t$). We ensure that the writes are propagated in the same order to *all* replicas through the following implication. Suppose we observe the snapshot $\mathsf{Seen}(j, s')$ of a replica j. Now suppose that *enough time has passed* so that there exists an event $a \in s'$ such that $a_2.t \leq a.t$; that is, the event observed at a_2 is *as recent* as the second write at a_1. Using the causality property from Figure 3 we show that the two writes at node i have been propagated to node j as the apply events a_1' and a_2' that respect program order ($a_1'.t < a_2'.t$). This way we express the MW guarantee.

We remark that our session manager library delegates the replica selection to clients. This is a simplification w.r.t practical implementations, where replica selection is done transparently by the session manager. In this simplified setting, our notion of causality is strong enough to provide session guarantees for the clients. Extending the case study to the general setting with transparent replica selection will require exposing a notion of time (*e.g.*, vector clocks) to the session manager and clients.

6 Verification of the Implementation

So far we have described how to use our specifications for client reasoning. In this section we show that our implementation from §2 does satisfy our specifications. Conceptually, the proof of the implementation can be split into the following three stages:

(1) We define a concrete notion of validity tying together all layers of the model (abstraction of the replicas' physical states, local histories, and the abstract global memory), and show that validity is preserved by the write and apply operations.
(2) We define the meaning of the abstract predicates ($\mathsf{Seen}(i, s)$, $\mathsf{Snap}(k, h)$, $k \rightharpoonup_u h$, $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$) using Iris ghost state.
(3) We define the lock invariant that governs replica-local shared data (the key-value dictionary, vector clock, in- and out-queues) and prove the correctness of the implementation of each operation.

In this section we discuss the key aspects of these three stages.

6.1 Local and Global Validity

Obviously, the local history s_i must be consistent with the physical state of the replica i for which it tracks the updates. We model the physical state for replica i as a *local state* (δ_i, t_i, s_i) defined as

$$(\delta_i, t_i, s_i) \in \mathit{LocalState} \triangleq (\mathit{Keys} \xrightarrow{\text{fin}} \mathsf{Option}(\mathsf{Value})) \times \mathsf{VectorClock} \times \mathit{LocalHistory}.$$

Here δ_i is a model of the local key-value store `db` used by the implementation at replica i, and t_i is the vector clock stored in the reference `vc` in the implementation.

We express consistency of the physical state as a validity property of the corresponding local state. To this end, we first observe that a local history at replica i can be partitioned into *sections* according to the origin of the apply events. Concretely, given a local history s_i, we define its j^{th} *section*, denoted by $s_{i,j}$, to be the subset of s_i events whose origin is j, *i.e.*, $s_{i,j} \triangleq \{a \in s_i \mid a.o = j\}$. Intuitively, for $j \in \{0..n-1\} \setminus \{i\}$, each section $s_{i,j}$ describes the external updates from replica j

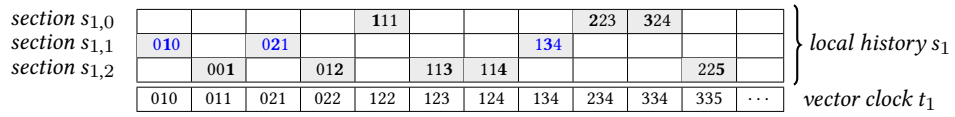

Fig. 14. A valid history s_1 partitioned into sections $s_{1,0}$, $s_{1,1}$, $s_{1,2}$, and a vector clock t_1 evolving through time. Each cell contains the time of apply events. For example, the apply events at section $s_{1,1}$ (colored blue in the table) are those events that come from the writes of replica 1. For each $s_{1,j}$, the j^{th} component of the vector clock is depicted in bold. Those numbers in bold reflect condition (2) in Definition 6.1.

applied locally on i, while the "diagonal" section $s_{i,i}$ describes the write operations executed on the replica i itself.[8]

Definition 6.1 (Valid Local Histories). Local history s_i is *valid* if the following conditions hold:[9]

(1) $\forall a_1, a_2 \in s_i.a_1.t = a_2.t \implies a_1 = a_2$
(2) $\forall k.1 \le k \le |s_{i,j}| \implies \exists a \in s_{i,j}.a.t[j] = k$
(3) $\forall a \in s_{i,i}.\forall j' \in \{0..n-1\}\backslash\{i\}\,.a.t[j'] = Sup\left\{b.t[j'] \mid b \in s_{i,j'} \wedge b.m < a.m\right\}$
(4) $\forall a \in s_{i,j}.\forall j' \in \{0..n-1\}\backslash\{j\}\,.j \neq i \implies a.t[j'] \le Sup\left\{b.t[j'] \mid b \in s_{i,j'} \wedge b.m < a.m\right\}$

The conditions above capture the fact that all apply events in the local history must have valid times. For instance, condition (2) reflects the fact that the set of apply events of a given section $s_{i,j}$ is downwards closed and complete w.r.t. the projection j of the vector clocks they carry. The most subtle are conditions (3) and (4), which ensure that for any event a that we have in our local history, we have observed all the events it depends on before observing a. To see this, note that $a.t[j']$ corresponds to the number of write events originating from replica j' that a depends on, and $Sup\left\{b.t[j'] \mid b \in s_{i,j'} \wedge b.m < a.m\right\}$ corresponds to the number of events we have observed from replica j' before a. Figure 14 illustrates the notion of validity with a concrete example.

Valid histories satisfy the following theorem which expresses the causality relation of the origin and the time projection of apply events:

Theorem 6.2. *If local history s_i is valid, then the following properties hold:*

- $s_{i,j} = \varnothing \iff \forall a \in s_i.a.t[j] = 0$ *(Empty section characterization)*
- $\forall a \in s_i.\forall j' \in \{0..n-1\}\,.\forall p \in \mathbb{N}^+.p \le a.t[j'] \implies \exists a' \in s_{i,j'}.a'.t[j'] = p$ *(Local causality)*
- $\forall a_1, a_2 \in s_{i,j}.a_1.t[j] = a_2.t[j] \implies a_1 = a_2$ *(Component extensionality)*
- $\forall k.1 \le k \le |s_{i,j}| \iff \exists a \in s_{i,j}.a.t[j] = k$ *(Strong completeness)*

Having defined the validity of local histories, we can now state the validity for local state (δ_i, t_i, s_i) as a predicate $Valid_{\mathrm{L}}(\delta_i, t_i, s_i)$ defined below.

Definition 6.3 (Valid local states). $Valid_{\mathrm{L}}(\delta_i, t_i, s_i)$ holds if the following conditions hold:

(1) $\forall k \in \mathrm{dom}(\delta_i), \forall v.\ \delta_i(k) = \mathsf{Some}(v) \implies \exists a \in s_i.a = \mathsf{Observe}(s_{i\,|k}) \wedge a \in \mathrm{Maximals}(s_{i\,|k})$
(2) $\forall k.\ \delta_i(k) = \mathsf{None} \implies s_{i\,|k} = \varnothing$
(3) $\forall j \in \{0 \ldots n-1\}\,.t_i[j] = Sup\left\{a.t[j] \mid a \in s_{i,j}\right\}$
(4) s_i is a valid local history

[8] In the following, we assume that all local histories and sections are well-formed: when we write $s_{i,j}$, we assume $i, j \in \{0..n-1\}$, and for any $a \in s_i$, the vector clock $a.t$ is of length n, the key $a.k$ belongs to the fixed set of keys, and the sequence identifier $a.m$ is less than or equal to the size of s_i.
[9] Here Sup is the supremum function. Note that $Sup(\varnothing) = 0$.

Predicate	Intuitive definition
$\mathsf{Snap}(k, h)$	ownership of $\circ_{\mathbb{S}}\ \{(k, h)\}$
$k \rightharpoonup_u h$	ownership of $\circ_{\mathbb{M}}\ \{(k, h)\}$ and $\circ_{\mathbb{S}}\ \{(k, h)\}$
$\mathsf{GM}(M)$	ownership of $\bullet_{\mathbb{M}}\ M$ and $\bullet_{\mathbb{S}}\ M$

Predicate	Intuitive definition
$\mathsf{Seen}(i, s)$	ownership of $\circ_{\mathbb{C}_i}\ s$ and $\circ_{\mathbb{L}_i}\ s$
$\mathsf{LH}_{\mathrm{G}}(i, s)$	ownership of $\circ_{\mathbb{C}_i}\ s$ and $\bullet_{\mathbb{L}_i}\ s$
$\mathsf{LH}_{\mathrm{L}}(i, s)$	ownership of $\bullet_{\mathbb{C}_i}\ s$ and $\circ_{\mathbb{L}_i}\ s$

Fig. 15. Predicates tracking abstract state of the distributed database defined in terms of resources.

Here conditions (1) and (2) express the consistency of the local history s_i with the local store δ_i, capturing the correctness argument of the `read` specification. Condition (3) states that in the local time of replica i, each projection $t_i[j]$ is equal to the projection $a.t[j]$, where a is the most recent event from section $s_{i,j}$, if such a exists, otherwise $a.t[j] = 0$. In particular, we have $\forall a \in s_i.\ a.t \leq t_i$.

Using the notion of validity for local histories, we can now finally define validity for global states.

Definition 6.4 (Valid global states). $Valid_{\mathrm{G}}(\{|\, M\,;\, s_1, \ldots, s_n\,|\})$ holds if

(1) $(\forall k \in \mathrm{dom}(M), w \in M(k).\ \exists a \in s_{w.o,w.o}.w = \lfloor a \rfloor) \wedge (\forall a \in \bigcup_{i=1}^{n} s_i.\ \exists w \in M(a.k).\ w = \lfloor a \rfloor)$

(2) All local histories $s_1, \ldots, s_n$ are valid.

Condition (1) defines a provenance relation between apply and write events in both directions. All the properties listed for valid global states in Figure 3 follow from this definition.

Crucially, the correctness argument for the `write` and `apply` operations relies on the following validity preservation theorem (which we here state only informally; see the Coq development for the formal statement):

Theorem 6.5 (Validity preservation). *Consider a valid global state $\{|\, M\,;\, s_1, \ldots, s_n\,|\}$ and a replica i whose local state (δ_i, t_i, s_i) is valid. The effects of both write and apply operations intuitively described below* ***preserve both local and global validity****.*

- *The effect of write event: adding a new apply event a with time $\mathsf{incr}_i(t_i)$ to s_i and a new write event $\lfloor a \rfloor$ to $M(a.k)$, where $\mathsf{incr}_j(t_i)$ is t_i with j^{th} incremented.*
- *The effect of apply event: adding a new apply event a to s_i such that a has passed the dynamic check and $\lfloor a \rfloor$ is already in $M(a.k)$.*

6.2 Ghost State

The theory of resource algebras in Iris [Jung et al. 2016] can be used to define so-called *ghost theories*, *i.e.*, to define resources and Iris propositions that assert ownership over resources. The exact combination of resource algebras and how they are used to define Iris propositions determines the properties of the ghost theories, *e.g.*, which propositions are persistent/ephemeral, the way resources can be updated, *e.g.*, allowing monotonic growth, allowing (de)allocation, *etc.* We refer the reader to Jung et al. [2018] and discussions therein for details of how resource algebras work.

One of the most important and versatile resource algebras is the so-called *authoritative* resource algebra, $\textsc{Auth}(\mathbb{A})$, where $\mathbb{A}$ is itself a resource algebra. The elements of the authoritative resource algebra are resources that are divided into two parts: the full part, of the form $\bullet_{\mathbb{A}}\ m$, and the fragment part, of the form $\circ_{\mathbb{A}}\ m$, for a resource $m \in \mathbb{A}$. The idea is that $\bullet_{\mathbb{A}}\ m$ is the central authoritative view of the ghost state, while $\circ_{\mathbb{A}}\ m'$ represents fragments of $\bullet_{\mathbb{A}}\ m$; we write this as $m' \preceq_{\mathbb{A}} m$, where $\preceq_{\mathbb{A}}$ is the resource inclusion relation for resources in $\mathbb{A}$. Hence, owning resource $\bullet_{\mathbb{A}}\ m$ is ephemeral, while $\circ_{\mathbb{A}}\ m$ can possibly be split up into multiple parts, depending on how elements of $\mathbb{A}$ can be split. Moreover, the ownership of $\circ_{\mathbb{A}}\ m$ may be ephemeral or persistent depending on whether ownership of elements of $\mathbb{A}$ is ephemeral or persistent.

$$\mathsf{GM}(M) * \mathsf{Snap}(k,h) \vdash \exists h'.M(k) = h' \wedge h \subseteq h' \qquad \text{(Snapshot inclusion)}$$
$$\mathsf{LH}_{\mathrm{G}}(i,s') * \mathsf{Seen}(i,s) \vdash s \text{ is a causally-closed subset of } s' \qquad \text{(Seen inclusion)}$$
$$\mathsf{LH}_{\mathrm{L}}(i,s) * \mathsf{LH}_{\mathrm{G}}(i,s') \vdash s = s' \qquad \text{(Local hist. agreement)}$$
$$\mathsf{GM}(M) * k \rightharpoonup_u h * h \subseteq h' \vdash \Rrightarrow_{\mathcal{E}} \mathsf{GM}(M[k := h']) * k \rightharpoonup_u h' \qquad \text{(Global mem. update)}$$
$$\mathsf{LH}_{\mathrm{L}}(i,s) * \mathsf{LH}_{\mathrm{G}}(i,s) * a \in \mathrm{Maximals}(s \cup \{a\}) \vdash \Rrightarrow_{\mathcal{E}} \mathsf{LH}_{\mathrm{L}}(i, s \cup \{a\}) * \mathsf{LH}_{\mathrm{G}}(i, s \cup \{a\}) \qquad \text{(Local hist. update)}$$

Fig. 16. Selected laws of the concrete ghost state.

Abstract Global Memory. We use two instance of the authoritative resource algebra, namely $\textsc{Auth}(\mathbb{S})$ and $\textsc{Auth}(\mathbb{M})$, for modeling the abstract global memory.[10] The resource algebra $\mathbb{S}$ is the resource algebra of finite maps from keys to finite sets of write events. It is defined so that the inclusion relation $M' \preceq_{\mathbb{S}} M$ holds if, and only if, $\forall x.\ x \in \mathrm{dom}(M') \implies M'(x) \subseteq M(x)$. That is, in $\textsc{Auth}(\mathbb{S})$ fragments track *lower bounds* of the sets of write events tracked in the authoritative part. Hence, ownership of fragments in $\textsc{Auth}(\mathbb{S})$ is persistent. The resource algebra $\mathbb{M}$ is the resource algebra of finite maps from keys to *exclusive* finite sets of write events. It is defined so that the inclusion relation $M' \preceq_{\mathbb{M}} M$ holds if, and only if, $\forall x.\ x \in \mathrm{dom}(M') \implies M'(x) = M(x)$. That is, in $\textsc{Auth}(\mathbb{M})$ fragments track *precisely* the sets of write events tracked in the authoritative part. Ownership of fragments in $\textsc{Auth}(\mathbb{M})$ is thus ephemeral. The authoritative parts of $\textsc{Auth}(\mathbb{S})$ and $\textsc{Auth}(\mathbb{M})$ are used to define $\mathsf{GM}(M)$ which is used in the global invariant to track the abstract global memory. The fragments are used to define $\mathsf{Snap}(k,h)$ and $k \rightharpoonup_u h$. Figure 15 gives the intuitive definition of these predicates. Note that fragments of both $\textsc{Auth}(\mathbb{S})$ and $\textsc{Auth}(\mathbb{M})$ are used in the definition of $k \rightharpoonup_u h$. This is why we can prove the rule (Take Snap) in Figure 6. An excerpt of the laws governing the use of predicates tracking ghost resources are presented in Figure 16.

Local History. We track local histories using two different kinds of resource algebras, $\textsc{Auth}(\mathbb{C})$ and $\textsc{Auth}(\mathbb{L})$, one instance of each per replica. When necessary, we write $\mathbb{C}_i$ and $\mathbb{L}_i$ instead of just $\mathbb{C}$ and $\mathbb{L}$ to distinguish instances used for replica i. Both $\mathbb{C}$ and $\mathbb{L}$ are similar in that they track sets of apply events. Moreover, the ownership of the fragments in both $\textsc{Auth}(\mathbb{C})$ and $\textsc{Auth}(\mathbb{L})$ are persistent. The main difference between $\mathbb{C}$ and $\mathbb{L}$ is in their inclusion relation:

$$s' \preceq_{\mathbb{L}} s \text{ if and only if } s' \subseteq s \qquad\qquad s' \preceq_{\mathbb{C}} s \text{ if and only if } s' \text{ is a } \textit{causally-closed} \text{ subset of } s$$

We need to track local history of a replica both in the global invariant and in the local lock invariant of the replica (see below). For this reason we define propositions $\mathsf{LH}_{\mathrm{G}}(i,s)$, and $\mathsf{LH}_{\mathrm{L}}(i,s)$, respectively. Figure 15 gives the intuitive definition of these predicates as well as that of the $\mathsf{Seen}(i,s)$. Note that $\mathsf{LH}_{\mathrm{G}}(i,s)$, and $\mathsf{LH}_{\mathrm{L}}(i,s)$ each have the full part of one of the two resource algebras and the fragment of the other. This fact, together with the inclusion relations above, is why we can prove the rule (Local hist. agreement) in Figure 16.

Global Invariant. The global invariant defined below simply states that there should exist an abstract global memory M and local histories $s_1, \ldots, s_n$ that we track using Iris resources such that the global state $\{|\, M\,; s_1, \ldots, s_n \,|\}$ is valid.

$$\mathsf{GlobalInv} \triangleq \exists M, s_1, \ldots, s_n.\ \mathit{Valid}_{\mathrm{G}}(\{|\, M\,; s_1, \ldots, s_n \,|\}) * \mathsf{GM}(M) * \mathop{\Large\ast}_{i=1}^{n} \mathsf{LH}_{\mathrm{G}}(i, s_i)$$

[10] We are eliding here the fact that yet another instance of the resource algebra $\textsc{Auth}(\mathbb{M})$ is used for modeling the abstract global memory. Both the authoritative as well as the fragment of this resource algebra are used as part of the definition of $\mathsf{GM}(M)$. This extra instance is used exclusively for defining the predicate $k \rightharpoonup_s h$ in §7. (Note that in Iris instances of resource algebras are named so as to allow us to have multiple instance of the same resource algebra.)

6.3 Proof of the Implementation

To verify the operations of the implementation, a crucial aspect is the choice of lock invariant. We use an Aneris lock module, which itself is implemented as a spin lock, and whose specification is very similar to the standard Iris lock, see, *e.g.*, [Birkedal and Bizjak 2017, Section 7.6]. The lock module uses an abstract predicate $\mathsf{isLock}(ip, \ell, P)$ to assert that the memory location ℓ is a lock on node ip protecting resources described by P.

The lock invariant for our distributed database is:

$$\mathsf{isLock}(ip_i, \mathtt{lock}, \Psi(i, \mathtt{db}, \mathtt{vc}, \mathtt{iq}, \mathtt{oq}))$$

where

$$\begin{aligned}
&\Psi(i, \mathtt{db}, \mathtt{vc}, \mathtt{iq}, \mathtt{oq}) \triangleq \\
&\quad \exists v_d, v_t, v_{iq}, v_{oq}.\ \exists \delta_i, t_i, s_i.\ \exists q_{in}, q_{out}.\ \exists ip_i, p. \\
&\qquad \mathsf{Addrlist}[i] = (ip_i, p) * \mathtt{db} \mapsto_{ip_i} v_d * \mathtt{vc} \mapsto_{ip_i} v_t * \mathtt{iq} \mapsto_{ip_i} v_{iq} * \mathtt{oq} \mapsto_{ip_i} v_{oq} * \\
&\qquad \mathit{isDictionary}(v_d, \delta_i) * \mathit{isVectorClock}(v_t, t_i) * \mathit{InQueue}(v_{iq}, q_{in}) * \mathit{OutQueue}(v_{oq}, q_{out}) * \\
&\qquad \mathsf{LH}_{\mathrm{L}}(i, s) * \mathit{Valid}_{\mathrm{L}}(\delta_i, t_i, s_i)
\end{aligned}$$

The predicate Ψ asserts that the dictionary `db`, the vector clock `vc`, and the queues `iq`, and `oq` are all allocated in the local heap of the replica i with values v_d, v_t, v_{iq}, and v_{oq}, respectively. It also enforces that the representation predicates $\mathit{isDictionary}$, $\mathit{isVectorClock}$, $\mathit{InQueue}$, $\mathit{OutQueue}$ tie together those program values with their logical counterparts δ_i, t_i, q_{in}, and q_{out}, respectively. Moreover, the predicate Ψ asserts that the local history s tracked for replica i (*cf.* $\mathsf{LH}_{\mathrm{L}}(i, s)$) together with the dictionary δ_i and vector clock t_i forms a valid local state, *i.e.*, $\mathit{Valid}_{\mathrm{L}}(\delta_i, t_i, s_i)$ holds. The $\mathit{InQueue}(v_{iq}, q_{in})$ and $\mathit{OutQueue}(v_{oq}, q_{out})$ predicates enforce that the contents of both queues are write events a for which we have $\mathsf{Snap}(a.k, \{a\})$.

With the lock invariant defined as above, we verify all operations of the database. The `init` function can use its precondition to establish the lock invariant—in fact, $\mathsf{initToken}(i)$ is defined as $\mathsf{LH}_{\mathrm{L}}(i, \varnothing)$. For the `write` operation we essentially need to prove that, given the precondition, it preserves the lock invariant and the global invariant, and that we can establish the post condition afterwards. The bulk of the proof, apart from reasoning about Iris resources, involves showing preservation of validity which follows directly from Theorem 6.5. For the `read` operation we only need to access the lock invariant and the global invariant in order to establish the postcondition—the lock invariant and the global invariant are trivially preserved as we do not change the state of the database. Recall that the postcondition of the `read` function almost follows from the definition of local state validity (Definition 6.3).

For the `apply` operation we essentially need to prove that it preserves the lock invariant and the global invariant. This follows from Theorem 6.5.

For the send thread we only need to show that the write events we send over the network adhere to the socket protocol Φ_{DB}. This immediately follows from $\mathit{OutQueue}(v_{oq}, q_{out})$ in the lock invariant. For the receive thread we need to show that the write events we receive over the network can be enqueued in q_{in}, *i.e.*, these are write events w for which we have $\mathsf{Snap}(w.k, \{w\})$. This immediately follows from the socket protocol Φ_{DB}.

7 HOCAP-style Specification for the Write Operation

In this section we present our HOCAP-style specification for the write operation, *cf.* the earlier discussion in the Introduction and in §4. Recall that the need for this more general specification comes from the fact that the natural specification of `write` involves ephemeral resources (the $k \rightharpoonup_u h$ resource used in the WriteSpec), which the clients should be able to govern by an Iris invariant in

case the clients concurrently access keys. For a client to use invariants, the write operation must be atomic since otherwise the client cannot open and close invariants around the write operation. But since the write operation is not atomic, we need to use another approach—and thus we use the HOCAP-style specification approach (see [Birkedal and Bizjak 2017] for an introduction to this style of specification).

In the HOCAP-style approach, there are in fact *two* views of the abstract state of the global memory of the key-value store: the client view $k \rightharpoonup_u h$, which we have seen before, and the module view $k \rightharpoonup_s h$; both of the abstract predicates are provided to the client as part of the modular specification interface of the replicated database. These two views always agree on the abstract state of the global memory, *i.e.*, $k \rightharpoonup_s h * k \rightharpoonup_u h' \vdash h = h'$. One of the key ideas is that neither the client nor the module can update its own view of the abstract global memory on their own. Instead, the module delegates updating the abstract global memory to the client (so that the client can control what happens in case the client needs to coordinate concurrent accesses using invariants). Thus the HOCAP-style write specification is parametrized by view shifts (update modalities) which the client has to prove and which allow the client to update the module's view of the abstract global memory $k \rightharpoonup_s h$ by combining it with its own view $k \rightharpoonup_u h$ of the abstract global memory. The latter is done using the following law

$$\forall w, \mathcal{E}.\ k \rightharpoonup_s h * k \rightharpoonup_u h \vdash \Rrightarrow_{\mathcal{E}} k \rightharpoonup_s h \uplus \{w\} * k \rightharpoonup_u h \uplus \{w\} \qquad \text{(System User Update)}$$

which we provide to the client as part of the modular interface describing the laws governing database resources. This law is in fact the single law that is missing in Figure 6 from §4.2.[11]

With this in mind, our the HOCAP-style specification for write is formally stated as follows:

$$\begin{aligned}
&\forall \mathcal{E}, k, v, s, P, Q.\ \mathcal{N}_{\mathsf{GI}} \subseteq \mathcal{E} \Rightarrow \\
&\square\big(\forall s', a.\ (s \subseteq s' * a \notin s' * a.k = k * a.v = v * P) \\
&\quad {}^{\top}\!\Rrightarrow\!\!*^{\mathcal{E}}\ \forall h.\ \begin{pmatrix} \lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\}) * k \rightharpoonup_s h * \\ \mathsf{Seen}(i, s' \uplus \{a\}) * \mathrm{Maximum}(s' \uplus \{a\}) = a \end{pmatrix} \\
&\qquad \Rrightarrow\!\!*_{\mathcal{E} \setminus \mathcal{N}_{\mathsf{GI}}}\ k \rightharpoonup_s h \uplus \{\lfloor a \rfloor\} * {}^{\mathcal{E}}\!\Rrightarrow^{\top} Q\ a\ h\ s'\big) \mathrel{-\!\!*} \\
&\{P * \mathsf{Seen}(i, s)\}\ \langle ip_i; \mathsf{write}(k, v)\rangle\ \{v.v = () * \exists h, s', a.\ s \subseteq s' * Q\ a\ h\ s'\}
\end{aligned}$$

where P has type *iProp* and Q has type $\mathit{ApplyEvent} \to \mathcal{P}_{\mathrm{fin}}(\mathit{WriteEvent}) \to \mathit{LocalHistory} \to \mathit{iProp}$.

Consider the view shifts before the Hoare triple for write. The client has to show, for the client's choice of predicates P and Q, that, given P and an apply event a corresponding to the write, if the client opens up invariants from the mask $X = \top \setminus \mathcal{E}$ and then additionally gets access to the module's view of the abstract state $k \rightharpoonup_s h$, then the client must be able to (1) update the abstract state to $k \rightharpoonup_s h \uplus \{\lfloor a \rfloor\}$, and, in doing so, they may open (and close) all invariants in $\mathcal{E}$, *except* for the global invariant $\mathcal{N}_{\mathsf{GI}}$ (which makes sense since that invariant is used internally by the implementation), and (2) close the invariants in X and then establish Q.

We remark that this use of view shifts is slightly more advanced than standard HOCAP-style specifications because here the client is allowed to open some invariants (those in X) before updating the abstract state and then close the invariants in X again to establish the postcondition Q.

This HOCAP-style specification of the write operations is our official specification and the one we have proved and use in our Coq formalization to verify client programs in a modular way.

As we mentioned earlier, the specification for the write operation in Figure 7 (WriteSpec) is derivable from the HOCAP-style specification above. To prove this, take $\mathcal{E}$ to be $\top$, let $P \triangleq k \rightharpoonup_u h$

[11] We define the meaning of these abstract predicates using appropriate Iris resource algebras and prove this law when verifying the implementation of the database.

(provided in the precondition of WriteSpec) and let

$$Q\ a\ h'\ s' \triangleq h = h' * a.k = k * a.v = v * \lfloor a \rfloor \in \mathrm{Maximals}(h' \uplus \{\lfloor a \rfloor\}) * \\ k \rightharpoonup_u h' \uplus \{\lfloor a \rfloor\} * \mathsf{Seen}(i, s' \uplus \{a\}) * a = \mathrm{Maximum}(s' \uplus \{a\}).$$

We can prove the equality $h = h'$ in Q because we have $k \rightharpoonup_u h$ (as P) and we know that the client and the module always agree on the view of the abstract global memory. Additionally, we use the rule System User Update with $w = \lfloor a \rfloor$ and $\mathcal{E} = \top \setminus \mathcal{N}_{\mathsf{GI}}$ to prove the update modality $\Rrightarrow\!\!*_{\mathcal{E} \setminus \mathcal{N}_{\mathsf{GI}}}$ in HOCAP-style spec above and obtain $k \rightharpoonup_s h' \uplus \{\lfloor a \rfloor\}$ and $k \rightharpoonup_u h' \uplus \{\lfloor a \rfloor\}$ (needed to prove Q).

8 Related Work

Lesani et al. [2016] present an abstract causal operational semantics for replicated key-value store implementations and their client programs. Through a refinement argument, two implementations in Coq's functional language, Gallina, are shown to realize this semantics. As a result of their approach, client programs can automatically be verified by model checking. In comparison, our work allows both the distributed database and clients to be implemented in a realistic ML-like language and verified using a separation logic in a completely modular way. This mean we can build libraries and provide abstractions on top of clients, as exemplified by our session manager library, and compose the database with other components to build and verify larger distributed systems. It is unclear how the approach of Lesani et al. would scale to a larger setting where a key-value store is just one component of a distributed system.

Several approaches exist for reasoning about weaker consistency models of distributed databases and their clients, including declarative approaches, *e.g.*, Adya et al. [2000]; Ahamad et al. [1995]; Burckhardt et al. [2012]; Cerone et al. [2015, 2017]; Cooper et al. [2008]; Gotsman et al. [2016] as well as operational approaches, *e.g.*, Crooks et al. [2017]; Kaki et al. [2018]; Schewe and Zhang [2018]; Xiong et al. [2019]. Common for all these works is that they reason about high-level models of distributed replicated databases and protocols with tools tailored for reasoning about databases, specific combinations of consistency models, and specific consistency guarantees. In constrast, our approach is aimed at the verification of concrete implementations and allows databases and clients to be composed with other components to build and verify larger distributed systems *while also* allowing us to reason about the weak consistency offered by the implementation.

Formal specification and verification of distributed systems and algorithms has been carried out by means of model checking [Holzmann 1997; Killian et al. 2007; Lamport 1992; Pnueli 1977] and, more recently, using a variety of program logics: Disel [Sergey et al. 2018] is a Hoare Type Theory for distributed program verification in Coq with ideas from separation logic. IronFleet [Hawblitzel et al. 2015] allows for building provably correct distributed systems by combining TLA-style state-machine refinement with Hoare-logic verification in a layered approach, all embedded in Dafny [Leino 2010]. Verdi [Wilcox et al. 2015] is a framework for writing and verifying implementations of distributed algorithms in Coq. Here we build on the Aneris logic, which supports horizontal and vertical composability of distributed systems implemented using sockets, node-local state and concurrency, and higher-order functions. Moreover, we rely on the Coq formalization of Aneris to mechanize all of our program correctness proofs.

In recent years, there has been a lot of work on formally specifying memory models of modern processors, *e.g.*, Alglave et al. [2010]; Armstrong et al. [2019]; Crary and Sullivan [2015]; Lahav and Boker [2020]; Mador-Haim et al. [2012]; Sevcík et al. [2011], and there has also been work on program logics for formal reasoning on top of such memory models, *e.g.*, Abe and Maeda [2016]; Bornat et al. [2015]; Doko and Vafeiadis [2016]; Turon et al. [2014]; Vafeiadis and Narayan [2013]. In particular, Kaiser et al. [2017] provide a framework for proving programs in a fragment of C11 containing release-acquire (RA) and non-atomic (NA) accesses. Their specifications for read

and write rely on a global view of the weak memory and a local view of each thread. While our specifications of read and write are at a very different level (for an implementation of a distributed database rather than for an operational semantics model of a processor), our specifications follow a similar pattern, as we track both the abstract global memory and the local history of each replica in our specifications. However, in *loc. cit.* each update is explicitly tracked only globally, and the local thread view only associates each location with the time of its latest update. We further note that in *loc. cit.* the consistency model corresponds to RA consistency of the weak memory, while our model describes causal consistency for a distributed system implementation. Bouajjani et al. [2017] show that causal consistency is equivalent to a WRA (weak release acquire) model which is strictly weaker than RA consistency. According to Lahav [2019], understanding how concurrent separation logics for the RA model can be weakened to the causal consistency is an interesting research question, and we hope that our specifications may serve as inspiration for future investigations in that direction.

9 Conclusion and Future Work

We have presented a modular formal specification of a causally-consistent distributed database in Aneris, a higher-order distributed separation logic, and proved that a concrete implementation of the distributed algorithm due to Ahamad et al. [1995] meets our specification. We have demonstrated that our specifications are useful, by proving the correctness of small, but tricky, synthetic examples involving causal dependency and by verifying a sesssion-manager library implemented on top of the distributed database. For the session-manager we have, moreover, verified formal program logic versions of the session guarantees known from the distributed systems literature.

We have relied on Aneris's facilities for modular specification and verification, in particular node-local reasoning *qua* socket protocols, to achieve a highly modular development, where each component is verified in isolation, relying only on the specifications (not the implementations) of other components. In particular, the distributed database is specified in the same style as other libaries and data structures are specified in distributed / concurrent separation logics, and thus it can be freely combined with other client programs and libraries (as evidenced by the session-manager library case study).

Future work includes implementing and verifying a strengthened session-manager library with transparent replica selection. It would also be interesting to verify other implementations of causally-consistent databases.

Acknowledgments

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

References

Tatsuya Abe and Toshiyuki Maeda. 2016. Observation-Based Concurrent Program Logic for Relaxed Memory Consistency Models. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*. 63–84. https://doi.org/10.1007/978-3-319-47958-3_4

Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. 67–78. https://doi.org/10.1109/ICDE.2000.839388

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Comput.* 9, 1 (1995), 37–49. https://doi.org/10.1007/BF01784241

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 258–272. https://doi.org/10.1007/978-3-642-14295-6_25

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL (2019), 71:1–71:31. https://doi.org/10.1145/3290384

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 761–772. https://doi.org/10.1145/2463676.2465279

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. (2017). http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf

Hans-Juergen Boehm and Sarita V. Adve. 2012. You don't know jack about shared variables or memory models. *Commun. ACM* 55, 2 (2012), 48–54. https://doi.org/10.1145/2076450.2076465

Richard Bornat, Jade Alglave, and Matthew J. Parkinson. 2015. New Lace and Arsenic: adventures in weak memory with a program logic. *CoRR* abs/1512.01416 (2015). arXiv:1512.01416 http://arxiv.org/abs/1512.01416

Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 626–638. http://dl.acm.org/citation.cfm?id=3009888

Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. 2004. From Session Causality to Causal Consistency. In *PDP*. IEEE Computer Society, 152–158.

Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. 2012. Eventually Consistent Transactions. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 67–86. https://doi.org/10.1007/978-3-642-28869-2_4

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2017. Algebraic Laws for Weak Consistency. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany (LIPIcs)*, Roland Meyer and Uwe Nestmann (Eds.), Vol. 85. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:18. https://doi.org/10.4230/LIPIcs.CONCUR.2017.26

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26. https://doi.org/10.1145/1365815.1365816

Kristina Chodorow and Michael Dirolf. 2010. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly.

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1, 2 (2008), 1277–1288. https://doi.org/10.14778/1454159.1454167

Karl Crary and Michael J. Sullivan. 2015. A Calculus for Relaxed Memory. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 623–636. https://doi.org/10.1145/2676726.2676984

Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*. 73–82. https://doi.org/10.1145/3087801.3087802

Thomas Dinsdale-Young, Pedro da Rocha Pinto, and Philippa Gardner. 2018. A perspective on specifying and verifying concurrent modules. *J. Log. Algebraic Methods Program.* 98 (2018), 1–25. https://doi.org/10.1016/j.jlamp.2018.03.003

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. https://doi.org/10.1145/564585.564601

Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2020. Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic - Technical Appendix. https://iris-project.org/pdfs/2021-popl-ccddb-appendix.pdf

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 371–384. https://doi.org/10.1145/2837614.2837625

Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure Recovery: When the Cure Is Worse Than the Disease. In *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*. https://www.usenix.org/conference/hotos13/session/guo

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 1–17. https://doi.org/10.1145/2815400.2815428

Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2018. Alone together: compositional reasoning and inference for weak isolation. *Proc. ACM Program. Lang.* 2, POPL (2018), 27:1–27:34. https://doi.org/10.1145/3158115

Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. 2007. Mace: language support for building distributed systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 179–188. https://doi.org/10.1145/1250734.1250755

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 205–217. http://dl.acm.org/citation.cfm?id=3009855

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

Ori Lahav. 2019. Verification under causally consistent shared memory. *ACM SIGLOG News* 6, 2 (2019), 43–56. https://doi.org/10.1145/3326938.3326942

Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 211–226. https://doi.org/10.1145/3385412.3385966

Leslie Lamport. 1992. Hybrid Systems in TLA$^+$. In *Hybrid Systems*. 77–102. https://doi.org/10.1007/3-540-57318-6_25

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 357–370. https://doi.org/10.1145/2837614.2837622

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. 401–416. https://doi.org/10.1145/2043556.2043593

Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer*

Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. 46–57. https://doi.org/10.1109/SFCS.1977.32

Sven Schewe and Lijun Zhang (Eds.). 2018. *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*. LIPIcs, Vol. 118. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. http://www.dagstuhl.de/dagpub/978-3-95977-087-3

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 43–54. https://doi.org/10.1145/1926385.1926393

Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 729–730. https://doi.org/10.1145/2213836.2213945

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

Andrew S. Tanenbaum and Maarten van Steen. 2007. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 691–707. https://doi.org/10.1145/2660193.2660243

Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. 2019. Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 636–650. https://doi.org/10.1145/3299869.3314049

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 867–884. https://doi.org/10.1145/2509136.2509532

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 357–368. https://doi.org/10.1145/2737924.2737958

Shale Xiong, Andrea Cerone, Azalea Raad, and Philippa Gardner. 2019. Data Consistency in Transactional Storage Systems: a Centralised Approach. *CoRR* abs/1901.10615 (2019). arXiv:1901.10615 http://arxiv.org/abs/1901.10615