Verifying a high-performance distributed transaction system using permissioned state machines

Yun-Sheng Chang, Joseph Tassarotti,† M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL † *New York University*

Abstract

Tulip is a high-performance distributed transaction system that uses sharding for scalability, replication within each shard for fault tolerance, and TAPIR-style inconsistent replication for high performance. Tulip comes with a machine-checked proof of correctness showing that its implementation meets a simple specification identical to a local strictly serializable transaction system, abstracting away implementation details such as crash recovery, multi-versioning, replication, sharding, and coordinator recovery.

The contribution of this paper is the *permissioned state machine* (PSM) approach, which extends TLA-style protocol reasoning with ideas from concurrent separation logic. PSM enables the developer to specify and verify complex protocols, such as Tulip, by breaking them down into smaller modules. PSM makes all dependencies between modules explicit using permissions, which limits the ways these modules can interact, and thereby reduces proof effort.

The prototype of Tulip consists of 3,956 lines of Go code, achieving performance competitive with that of TAPIR. Tulip's proof is decomposed into 10 types of modules; the majority of logical proof steps ("actions") involve just one module, demonstrating that PSM enables local reasoning.

1 Introduction

A common approach to formally verify the correctness of a distributed system implementation is to specify the protocol as a state machine, in the style of TLA+. This specification consists of the protocol's abstract state variables, an initial state assigning values to those variables, and a step function that describes legal transitions from one abstract state to another. For example, IronFleet [6] shows how this approach can be used to verify distributed system implementations written in Dafny. The core proof of protocol correctness with this approach rests on proving *invariants* about the system's abstract state. After the developer formulates a precise invariant, they must prove that every legal step preserves this invariant.

This paper's goal is to scale up this approach to verify complex distributed systems, with the driving example being Tulip, a high-performance sharded replicated transaction system that uses inconsistent replication in the style of TAPIR [31]. A key challenge in verifying a complex protocol using the state-machine approach is that the state and invariants are *global*, even if the protocol itself is composed of largely independent sub-protocols, which makes it difficult for the developer to verify the sub-protocols in isolation. For example, Multi-Paxos is a sub-protocol used by Tulip for executing consistent operations across a group of replicas. In a state-machine specification, the invariants of both Tulip and Multi-Paxos refer to the shared variable representing the list of pending operations, which were submitted by Tulip and are now waiting for Multi-Paxos to linearize them. This means that, when either Tulip or Multi-Paxos updates this variable, all of these invariants might be affected, and the developer must prove that the invariant is preserved by every such update. The same modularity issue arises inside the Multi-Paxos library as well, where proposers and acceptors share the variables representing proposals and votes.

This paper presents the *permissioned state machine* (PSM) approach for making state-machine reasoning about distributed systems more modular. The contribution of PSM lies in adopting ideas from concurrent separation logic (CSL) to reason about shared state in TLA-style state machine specifications: instead of allowing every invariant to refer to the entire abstract state (such as the shared variables described in the above paragraph), PSM requires each invariant to explicitly declare what part of the shared state it depends on (called a *permission*). For example, the invariant of the Multi-Paxos library depends on a prefix of the pending operations; the invariant of the Paxos proposer depends on a prefix of the votes; and the invariant of the Paxos acceptor depends on specific proposal numbers, rather than all of the proposals. This, in turn, allows updating the shared state in a way that does not violate the invariants of other modules. For instance, Tulip can append a new pending operation without breaking the current invariant of Multi-Paxos (since it only depends on a prefix), and a Paxos acceptor can issue a new vote without invalidating the proposer's invariant (which depends only on a prefix of the votes).

A second advantage of PSM is that it provides a modular approach to verifying a concurrent implementation of

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author.

Copyright is held by the owner/author(s).
SOSP '26, September 29–October 2, 2026, Prague, Czech Republic
ACM ISBN 979-8-4007-2585-2/26/09.
http://dx.doi.org/10.1145/3830418.3843873

a distributed system. Proving the correctness of an implementation requires establishing a correspondence between implementation state, such as the contents of memory, and the protocol state. In the presence of concurrent operations, this correspondence can be complicated, because the implementation state has partial updates for many in-progress operations, but the protocol state typically performs atomic execution of protocol steps, one at a time. PSM's use of CSL enables developers to state this correspondence in a modular fashion. For example, in a key-value store, the developer can relate the implementation state owned by a given thread (such as a particular key currently locked by that thread) to the corresponding protocol state (such as the logical value of that key) in the proof of that thread, without having to mention the other threads or other keys (it will be up to other threads and invariants to establish their correspondence).

To demonstrate the PSM approach, we developed and verified Tulip, a transactional storage system that implements sharding, replication, crash recovery, multi-versioning, and coordinator recovery, using ideas similar to those in Spanner [5], TAPIR [31], and other state-of-the-art systems. Tulip implements inconsistent replication (IR), in the style of TAPIR, which cuts across the replicated-state-machine abstraction, but allows Tulip to commit in one network round-trip, with Tulip clients directly contacting all servers in each replica group. Inconsistent replication further complicates crash recovery: since there is no consistent snapshot of correct replica state, IR requires replicas to follow a complex protocol to obtain the correct state after recovery. The key novelty of Tulip is that it comes with a machine-checked proof of correctness for its executable implementation, as written in Go, including crash recovery. Our verification evaluation reports on how PSM allowed us to decompose the specs and proofs of Tulip into smaller components with only the necessary dependencies between them. Tulip is the first verified distributed transaction system implementation, and no other verified distributed system implementations match Tulip's level of complexity, both in terms of its protocol-level features (inconsistent replication, Multi-Paxos replication, sharding, transactions) as well as its implementation features (crash recovery, concurrency, network failures, timestamps, multi-versioning).

The Tulip implementation consists of ~4,000 lines of Go code, including multi-version concurrency control, crash-safe on-disk storage, Paxos replication, RPC handling, coordinator recovery, and a client library for writing transactional applications. The client library provides a simple specification to the application—any code running in a Tulip transaction will execute with strict serializability—which is the same specification that has been previously proven for a single-node MVCC-based database [4]. We used Goose [3], Perennial [2], and Grove [25] to formally verify that Tulip meets this specification; the proof consists of ~42,000 lines of Rocq code developed with the Iris Proof Mode (IPM) [10, 12]. The proof is about 11× the number of lines of executable code, which is comparable to verified distributed systems that have strong layering [25]; this shows that the PSM approach makes it possible to verify sophisticated distributed transaction systems, such as Tulip, without an explosion in the proof effort.

To summarize, the contributions of this paper are:

- The PSM approach for combining concurrent separation logic ideas with state-machine reasoning, which enables modular decomposition of the proofs of complex distributed systems.
- Case studies of using the PSM approach to verify distributed system implementations in a modular fashion, including the single-decree Paxos protocol, which we use as a running example, and the Tulip distributed transactional storage system.
- A detailed analysis of the benefit of PSM in verifying Tulip, including a breakdown of Tulip's components and the dependencies between them as captured by PSM.
- A performance evaluation of Tulip demonstrating that it is competitive with TAPIR and implements the same sophisticated optimizations (but formally verified).

One of the limitations of Tulip is that its specification does not capture liveness: Tulip's proof does not guarantee that, for example, all transactions will eventually either commit or abort. Like TAPIR, Tulip does not support reconfiguration.

2 Related work

To the best of our knowledge, PSM is the first to integrate the idea of permissions, from concurrent separation logic, into TLA-style protocol reasoning.

Verification. TLA+ has been widely used to verify protocols, including Paxos [14] and TAPIR [30]. PSM combines CSL's techniques for modular verification with TLA-style protocol proofs. This allows developers to precisely capture the dependencies between sub-protocols in a complex system such as Tulip, and then verify these sub-protocols largely independently of one another, which makes it possible to scale verification to sophisticated protocols. PSM further connects protocol-level proofs to proofs of concurrent code. This ensures the absence of implementation bugs, such as the timestamp inversion bug that was present in TAPIR despite its TLA+ proof [18].

CSL is widely used for verifying concurrent software, and has been used to verify concurrent distributed systems [13, 25]. However, the challenge in applying CSL directly to a complex system like Tulip is the lack of structure. Going directly from code to Hoare-triple specifications, as is normally done with CSL, means constantly mixing reasoning about protocol correctness with reasoning about implementation details. PSM separates protocol-level reasoning from code-level reasoning, but retains CSL's benefits in supporting reasoning about fine-grained implementation concurrency.

PaxosMach

VARIABLES *Proposals*, *Votes*

$EqLatest(n, v, m) \triangleq$
$\exists Q \in Quorum, c \in -1 \,..\, (n-1).$
C1 $\wedge \forall d \in (c+1) \,..\, (n-1), a \in Q.\ m[a][d] = \texttt{None}$
C2 $\wedge (c \neq -1) \implies \exists a \in Q.\ m[a][c] = \texttt{Some}(v)$

$Inv2a\dagger \triangleq \forall \langle n, v\rangle \in Proposals.\ EqLatest(n, v, Votes)$

$Inv2b\dagger \triangleq \forall a, n, v.\ Votes[a][n] = \texttt{Some}(v) \implies \langle n, v\rangle \in Proposals$

$Action2a\dagger(n, v) \triangleq$
E1 $\wedge EqLatest(n, v, Votes)$
E2 $\wedge n \notin dom(Proposals)$
A1 $\wedge Proposals' = Proposals \cup \{\langle n, v\rangle\}$

$Action2b\dagger(a, n, v) \triangleq$
E3 $\wedge len(Votes[a]) \leq n$
E4 $\wedge \langle n, v\rangle \in Proposals$
A2 $\wedge Votes' = [Votes \text{ EXCEPT } ![a] = extend(Votes[a], n, v)]$

Figure 1: A monolithic Paxos state-machine specification adapted from Lamport written in a TLA-like language. *Proposals* and *Votes* are global state variables. This partial specification shows two safety invariants and two actions in Paxos. The function $extend(l, n, v)$ fills the list l with None up to index n, then appends Some (v) at index n. The expression $[m \text{ EXCEPT } ![i] = v]$ denotes m with its entry at i set to v.

Tulip builds on ideas from prior distributed systems with machine-checked proofs [6, 8, 9, 13, 23, 25, 26, 28], and in particular, uses Grove [25] for its verification. This paper's key contribution lies in articulating PSM for modular decomposition of distributed system proofs, and applying it to verify a high-performance distributed transaction system.

Monotonic states and stable permissions. PSM allows developers to capture stable dependencies of monotonic states, such as knowing a prefix of the votes in a Paxos library. Prior work has explored monotonic states and stable permissions in type systems [21], their use in distributed computing [1, 7, 24], and their use in mechanized proofs [26, 32] including work on resource algebras [10, 25, 27]. PSM's key benefit is using permissions more generally, including stable permissions as a special case, to enable modular verification of protocols and to integrate protocol-level proofs with proofs of concurrent code.

Distributed transactions. Tulip is inspired by TAPIR and its inconsistent-replication optimization. The main difference is that Tulip uses Paxos to consistently replicate commit decisions, which simplifies the task of replica synchronization. If replicas get out-of-sync with each other (e.g., due to network partitions, crashes, etc.), TAPIR requires the use of a specialized reconciliation protocol (which the TAPIR prototype did not implement), whereas Tulip simply applies the longest global log from the highest Paxos term (which we did implement and verify).

3 Motivating example: single-decree Paxos

To motivate the need for the PSM approach, consider single-decree Paxos as a running example. Single-decree Paxos is a distributed consensus algorithm that allows a set of processes to agree on a single value. Processes performing the algorithm are divided into roles, including *proposers* and *acceptors*. At a high level, a proposer makes a proposal, which consists of a *proposal number* and a *proposed value*, to acceptors. Once a majority of acceptors vote for a proposal, the proposed value is agreed upon.

Figure 1 shows an excerpt of Lamport's Paxos TLA+ specification [16], with modifications for ease of presentation. In particular, we store proposals and votes separately in two variables, *Proposals* and *Votes*, as opposed to storing them in a single set of network messages. Also, we use a map of lists, rather than sets, to represent the votes. Each element in the list $Votes[a]$ is an optional value with the following meaning: Some (v) at index n means: "proposal $\langle n, v\rangle$ accepted by the acceptor a", and None at index n means: "no proposal numbered n accepted by the acceptor a". We will explain in §4.6 why representing votes this way helps achieve modularity.

Given a state-machine specification like the one shown in Figure 1, the standard way to reason about its correctness is to show that every transition described by the actions takes a state satisfying the invariants to another state that also satisfies them. Below, we discuss two invariants and two actions central to single-decree Paxos.

Inv2a† requires that, in order to make a proposal $\langle n, v\rangle$, a proposer must first obtain promises from a quorum Q of acceptors not to vote for any proposal numbered below n. Additionally, there must exist a number c that is either the highest proposal number any acceptor $a \in Q$ has voted for, or -1 if no acceptor has voted (conjunct C1); if c is not -1, then v must match the value voted for by some acceptor with number c (conjunct C2). *Inv2b*† requires each acceptor to only vote for proposals that have actually been proposed.

Action2a†(n, v) models the act of making a proposal $\langle n, v\rangle$ (conjunct A1) provided that the proposal satisfies *Inv2a*† (conjunct E1) and that the proposal number n is fresh (conjunct E2). *Action2b*†(a, n, v) captures acceptor a voting for $\langle n, v\rangle$ (conjunct A2) provided that it has not voted for, or promised not to vote for, any proposal numbered n (conjunct E3), and that $\langle n, v\rangle$ has indeed been proposed (conjunct E4).

Challenge 1: action-invariant cross-product. Reasoning about state-machine correctness is formulated as invariance proofs: proving that each invariant is preserved by each action. In our example above, all of the actions and invariants depend on the global variables *Proposals* and *Votes*. The example showed two invariants and two actions, requiring four (2×2) invariance proofs in total. In Lamport's complete formulation of Paxos, there are several more actions and invariants. As the protocol grows in complexity, the number of action-invariant pairs grows quadratically. Adding, or even modifying, an action or invariant becomes increasingly costly. This presents a scalability challenge for verifying Tulip, which has many more actions and invariants.

Challenge 2: modularity. A monolithic specification like the one in Figure 1 imposes both cognitive load on the developer and processing burden on the proof assistant. Ideally, we would like to group variables, invariants, and actions into modules around a common concern. Consider Figure 1 as an example. *Inv2a*† constrains which proposals are valid; *Action2a*† models the creation of a new proposal; naturally, these two, together with the variable *Proposals*, belong to a "proposers" module. Similarly, *Inv2b*† requires each acceptor to vote only for proposals in the proposal set; *Action2b*† captures an acceptor casting a vote for a given proposal; together with *Votes*, they belong to an "acceptor" module. However, this natural modular boundary is undermined by the fact that all these invariants and actions directly depend on both *Proposals* and *Votes*.

Verifying Tulip using a monolithic specification would be impractical. Tulip has a naturally modular design: each replica group handles a partition of the database; within a group, each replica server maintains a copy of that partition to achieve fault tolerance; each key maps to multiple versions for improved concurrency; the transaction system coordinates concurrent transactions. Separate network and file modules handle message passing and crash recovery. We would like to take advantage of the natural modularity, while still capturing the dependencies between the modules.

4 Approach: permissioned state machine

In this section, we introduce the *permissioned state machine* (PSM) approach to address the two challenges from the previous section, by integrating ideas from concurrent separation logic (CSL) [10, 20, 25] into TLA-style protocol reasoning.

Figure 2 shows an overview of the PSM approach. In PSM, developers start by defining the PSM specification for each module, including the state variables, permissions, invariants, and actions. Then, developers prove validity of the actions as lemmas that serve as the interface between protocol and code. Finally, developers prove the implementation code against Hoare triples using those action lemmas.

This section starts by introducing the necessary background on CSL, and then explains each layer of the PSM approach in more detail, using the same single-decree Paxos example we saw in §3.

4.1 Background: concurrent separation logic

In CSL, the behavior of a function `fn` is specified as a Hoare triple in the form of $\{P\}$ `fn` $\{Q\}$, where P is called the *precondition* and Q the *postcondition*. The precondition asserts what the function requires before its execution, while the postcondition asserts what the function establishes after its return. CSL extends Hoare logic with the notion of *resources* and their *ownership*. For instance, a common resource is the *heap points-to*, denoted as $x \mapsto v$, which gives the owning thread full access to the heap cell located at x.

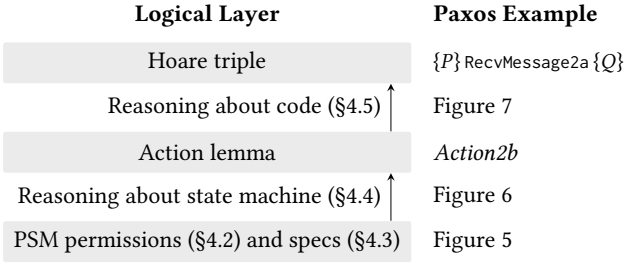


Figure 2: A layering overview of the permissioned state machine approach. Each layer above an arrow is proved using the layer below it.

Requiring full access to certain resources limits the kinds of concurrency that a specification can express. We often want to impose certain *permissions* that restrict some forms of access while making others possible. For example, by limiting accesses to read-only, we enable multiple concurrent readers. As another example, by requiring the writer never to decrement an integer, we allow infinitely many readers to observe lower bounds on that integer. *Resource algebras* [10] provide a generic way to construct resources equipped with such permissions.

The principle underlying CSL's modularity is the *frame rule*. The rule enables using a subset of resources connected with the *separating conjunction*, denoted as $*$, to reason about a code fragment, while keeping the other unused resources intact. For instance, say we have $x \mapsto v_x * y \mapsto v_y$ before invoking a function that mutates only x. In this case, we can "frame away" the unused resource $y \mapsto v_y$, pass $x \mapsto v_x$ to that function, and obtain the updated resource back on the function return. The frame rule ensures that $y \mapsto v_y$ is unchanged without the need to explicitly specify it for the function. Such "minimalism" is the key to modularity because it allows us to prove each module locally and later combine them into the overall system.

4.2 Defining permissions

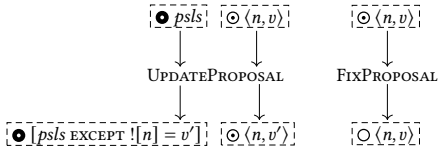
To make the main ideas of PSM concrete, we reformulate the single-decree Paxos example presented earlier in PSM terms. The first step is to add permissions on the same abstract state variables—the proposal set and the voted lists. Figure 3 summarizes the permissions we choose for the example.

For the proposal set, we use the permission $\boxed{\bullet\ psls}$ to encode the full view of the proposal set, and $\boxed{\odot\ \langle n, v\rangle}$ for the point-wise view at number n. The two permissions are owned by separate parties. $\boxed{\bullet\ psls}$ is owned by the protocol state machine and is used to construct global invariants on the entire proposal set. By contrast, $\boxed{\odot\ \langle n, v\rangle}$ is owned by the implementation in the lock invariant and connects the implementation state to the protocol state in a fine-grained fashion, allowing for implementation-level concurrency.

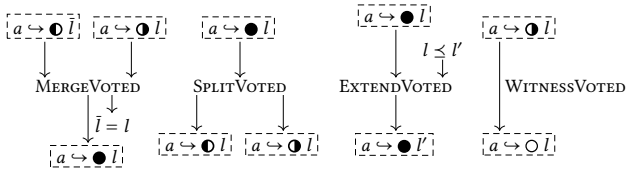
Importantly, for this pair of permissions to be sound, owning just one of them should not permit the owner to modify the proposal set, as that would otherwise invalidate the contract of the other permission. We must update both per-

Abstract state	Permission	Semantics	Owner	Stable?
Proposal set	$\boxed{\bullet\ psls}$	The proposal set is *psls*	*ProposersMach*	No
	$\boxed{\odot\ \langle n, v\rangle}$	The proposal numbered n has value v	*LockInv*(px, a)	No
	$\boxed{\circ\ \langle n, v\rangle}$	The proposal numbered n has value fixed to v	*AcceptorMach*(a)	Yes
Voted lists	$\boxed{a \hookrightarrow \bullet\ l}$	The acceptor a's voted list is l, and it can be further extended	No owner	No
	$\boxed{a \hookrightarrow \mathbin{\bullet\!\!\circ}\ l}$	The acceptor a's voted list is l	*AcceptorMach*(a)	No
	$\boxed{a \hookrightarrow \mathbin{\circ\!\!\bullet}\ l}$	The acceptor a's voted list is l	*LockInv*(px, a)	No
	$\boxed{a \hookrightarrow \circ\ l}$	The acceptor a's voted list is at least l	*ProposersMach*	Yes

Figure 3: Permissions on the abstract state of our single-decree Paxos example. This table focuses on the meaning of a permission; Figure 4 details the rules for *updates* using permissions. *ProposersMach* and *AcceptorMach*(a), defined in Figure 5, are the PSM protocol-level specifications for all proposers and for an individual acceptor a, respectively. *LockInv*(px, a) is the lock invariant of our Paxos *implementation*, which contains the facts that the implementation proof can assume upon obtaining the lock (mutex) and must re-establish before releasing it. Its parameter px is a pointer to the Go struct that implements both acceptor and proposer roles.



(a) Interaction rules for permissions on the proposal set.



(b) Interaction rules for permissions on the voted lists. WitnessVoted can accept $\boxed{a \hookrightarrow \mathbin{\bullet\!\!\circ}\ l}$ or $\boxed{a \hookrightarrow \bullet\ l}$ as input as well. Unlike other rules, WitnessVoted does not consume its input permission.

Figure 4: Permission interaction rules for the single-decree Paxos example. Each rule takes in one or more input permissions and produces one or more output permissions.

missions in synchrony to set the value of proposal n. The interaction rule UpdateProposal, shown in Figure 4(a), formalizes this requirement.

The last kind of permission on the proposal set, $\boxed{\circ\ \langle n, v\rangle}$, is similar to the point-wise view $\boxed{\odot\ \langle n, v\rangle}$, except that it additionally asserts that the value has been fixed to v. $\boxed{\circ\ \langle n, v\rangle}$ can be obtained from $\boxed{\odot\ \langle n, v\rangle}$ using the FixProposal rule shown in Figure 4(a). Converting $\boxed{\odot\ \langle n, v\rangle}$ into $\boxed{\circ\ \langle n, v\rangle}$ illustrates an important tradeoff: we forfeit the ability to further update the proposal value, but in return, since the value is fixed to v, the resulting permission is no longer exclusive—meaning it can now be owned by arbitrarily many parties at the same time. We refer to these kinds of non-exclusive permissions like $\boxed{\circ\ \langle n, v\rangle}$ as *stable permissions*. As we discuss later, stable permissions play a crucial role in PSM.

Similarly to the proposal set, we use a pair of permissions on the voted lists: $\boxed{a \hookrightarrow \mathbin{\bullet\!\!\circ}\ l}$ and $\boxed{a \hookrightarrow \mathbin{\circ\!\!\bullet}\ l}$ represent the two half-ownerships of a's voted list, with $\boxed{a \hookrightarrow \mathbin{\bullet\!\!\circ}\ l}$ owned by the protocol state machine and $\boxed{a \hookrightarrow \mathbin{\circ\!\!\bullet}\ l}$ by the implementation. Having just one of them grants read access, while owning both allows mutation. Moreover, such mutation is restricted to append-only updates. This additional restriction is useful, as it in turn enables another stable permission, $\boxed{a \hookrightarrow \circ\ l}$, which asserts that a's voted list is at least l.

Figure 4(b) shows the interaction rules governing these voted-list permissions. We can merge half-ownerships $\boxed{a \hookrightarrow \mathbin{\bullet\!\!\circ}\ l}$ and $\boxed{a \hookrightarrow \mathbin{\circ\!\!\bullet}\ l}$ into full ownership $\boxed{a \hookrightarrow \bullet\ l}$ using MergeVoted, or vice versa using SplitVoted. With full ownership, we have the right to extend the voted list, as captured by ExtendVoted. We can further use WitnessVoted to extract $\boxed{a \hookrightarrow \circ\ l}$. In §4.4 and §4.5, we illustrate the use of these interaction rules in concrete proofs.

4.3 Specifying protocols using permissions

Figure 5 shows the PSM specifications for the same Paxos invariants and actions as those shown in Figure 1. One immediate difference is that we now have two separate PSM specifications: one for the proposers, *ProposersMach*, which contains *Inv2a* and *Action2a*; and one for each individual acceptor a, *AcceptorMach*(a), which contains *Inv2b* and *Action2b*. We use the keyword Invariants to declare the invariants of each specification, including permissions it owns.

Let us first consider *ProposersMach*, as shown in Figure 5(a). Instead of referring directly to the global variable *Proposals*, the specification asserts that it has the permission $\boxed{\bullet\ psls}$, which permits the holder of the permission to conclude that *psls* is the set of all proposals. The invariant *Inv2a* has a similar structure to its TLA+ counterpart *Inv2a*†. The key difference is that, unlike *Inv2a*†, *Inv2a* does not depend on the exact value of *Votes*, but merely on some lower bounds *votes* existentially quantified within the invariant. The fact that *votes* are lower bounds of the actual votes is expressed with the permission $\boxed{a \hookrightarrow \circ\ l}$, which permits the holder to conclude that l is a prefix of acceptor a's voted list.

PSM also uses permissions to describe how actions update state. A PSM action is defined using the $\Rrightarrow$ notation, which says that the action requires the permissions on the left of $\Rrightarrow$ and returns the permissions on the right of $\Rrightarrow$, while preserving the invariants of its PSM specification (formally, $\Rrightarrow$ is the Iris view shift [10]). For example, *Action2a*

ProposersMach

VARIABLES *psls*

INVARIANTS $\boxed{\bullet\ psls}$, *Inv2a*

$Inv2a \triangleq$
$\forall \langle n, v\rangle \in psls.\ \exists votes.\ EqLatest(n, v, votes) * \divideontimes_{\langle a, l\rangle \in votes} \boxed{a \hookrightarrow \bigcirc\ l}$

$Action2a(n, v) \triangleq$
$\exists m.\ EqLatest(n, v, m) * \divideontimes_{\langle a, l\rangle \in m} \boxed{a \hookrightarrow \bigcirc\ l} * \boxed{\odot \langle n, -\rangle} \Rrightarrow \boxed{\odot \langle n, v\rangle}$

(a) PSM specification for Paxos proposers. In contrast to *Inv2a*†, *Inv2a* does not depend on the exact value of *Votes*. Rather, *votes* is existentially quantified within the invariant. It is accompanied by the stable permission $\boxed{a \hookrightarrow \bigcirc\ l}$ witnessing that *votes* are lower bounds of the actual votes. *EqLatest*, defined in Figure 1, requires the proposed value v to agree with the latest prior vote, if one exists. The symbol $\divideontimes$ denotes separating conjunction over a collection of permissions (or resources more generally).

AcceptorMach(a)

VARIABLES *voted*

INVARIANTS $\boxed{a \hookrightarrow \bigcirc\!\!\blacksquare\ voted}$, *Inv2b*

$Inv2b \triangleq \forall n, v.\ voted[n] = \texttt{Some}(v) \mathrel{-\!\!*} \boxed{\bigcirc \langle n, v\rangle}$

$Action2b(n, v, l) \triangleq$
$len(l) \le n \wedge \boxed{\bigcirc \langle n, v\rangle} * \boxed{a \hookrightarrow \blacksquare\ l} \Rrightarrow \boxed{a \hookrightarrow \blacksquare\ extend(l, n, v)}$

(b) PSM specification for an individual acceptor. Each acceptor PSM owns its voted list. Defined using the magic wand $-\!\!*$ (i.e., implication in separation logic), *Inv2b* says that if the acceptor votes for $\langle n, v\rangle$, then $\langle n, v\rangle$ must have been proposed.

Figure 5: PSM specifications for single-decree Paxos. A PSM specification is formally defined in our Rocq proof as a separation-logic resource in which variables are existentially quantified and invariants are internal resources connected by separating conjunctions ($*$). Aside from their pre- and post-conditions, actions declared within a PSM specification take and give the PSM invariants.

in Figure 5(a) takes the $\boxed{a \hookrightarrow \bigcirc\ l}$ permissions required to re-establish *Inv2a* and updates $\boxed{\odot \langle n, -\rangle}$ to $\boxed{\odot \langle n, v\rangle}$, where $-$ denotes any value.

Figure 5(b) shows the PSM specification for each individual acceptor a. The permission $\boxed{a \hookrightarrow \bigcirc\!\!\blacksquare\ voted}$ allows the holder to conclude that acceptor a's voted list is *voted*. The invariant *Inv2b* again has a similar form to its TLA+ counterpart *Inv2b*†, except in two respects. First, *Inv2b* refers to the *voted* list of the specific acceptor a, rather than to the global variable *Votes*, which stores the votes of all acceptors. Second, *Inv2b* uses the permission $\boxed{\bigcirc \langle n, v\rangle}$ as evidence that $\langle n, v\rangle$ belongs to the proposal set, thereby eliminating the need for the global assertion $\langle n, v\rangle \in Proposals$ required by *Inv2b*†. This permission allows the holder to conclude that the value of proposal n was v (and will remain so forever), but does not allow updating it.

The action *Action2b*, as shown in Figure 5(b), has two preconditions similar to its TLA+ counterpart *Action2b*†, except that it avoids global references using the techniques discussed above. The rest of *Action2b* requires passing in the permission $\boxed{a \hookrightarrow \blacksquare\ l}$ and returns $\boxed{a \hookrightarrow \blacksquare\ extend(l, n, v)}$, capturing the same transition as *Action2b*†.

PSM benefit 1: avoiding the cross-product. A key benefit of PSM's permission-based approach is that it avoids the need to consider the quadratic combination of every action and invariant. In the Paxos example, we no longer need to prove *Action2a* against *Inv2b* or *Action2b* against *Inv2a*, reducing the number of invariance proofs from four to two. The reason this is possible is that, in PSM, invariants can be defined over stable permissions: for instance, *Inv2a* is defined over $\boxed{a \hookrightarrow \bigcirc\ l}$, and *Inv2b* over $\boxed{\bigcirc \langle n, v\rangle}$. Stable permissions refer to parts of the state that must remain unchanged, so invariance holds by construction.

PSM benefit 2: modular organization. PSM also enables modular organization. Variables are assigned ownership permissions, and cross-module dependencies are expressed through stable permissions. In the Paxos example, we organize the system into two PSM specifications: *ProposersMach* and *AcceptorMach*, as we saw in Figure 5. *ProposersMach* owns the proposal set via $\boxed{\bullet\ psls}$, while *AcceptorMach* may refer to individual fixed proposals through the stable permission $\boxed{\bigcirc \langle n, v\rangle}$ in its actions and invariants. Conversely, *AcceptorMach* owns the votes via $\boxed{a \hookrightarrow \bigcirc\!\!\blacksquare\ l}$, while *ProposersMach* may refer to their prefixes through $\boxed{a \hookrightarrow \bigcirc\ l}$.

While the advantages of PSM may seem less important with the small Paxos example, they become significant when verifying Tulip. As we discuss later, the mechanized proof of Tulip involves 25 actions, more than 50 abstract state variables, and numerous invariants relating them. Keeping these in a single monolithic specification would be impractical. It would also be overwhelming to have to consider the full cross-product of each action and invariant to see if the action preserves that invariant—particularly because it increases the effort required to make changes to invariants or actions, thereby impeding incremental and modular development.

4.4 Proving validity of actions

The bulk of the effort required to prove a protocol in PSM lies in proving that the protocol's actions are valid. Mechanically, this takes the form of spelling out the steps that an action performs in order to satisfy its definition—that is, starting from the supplied permissions and arriving at the resulting permissions, as exemplified in Figure 5(b) for *Action2b*.

Figure 6 shows a diagram of the steps involved in proving *Action2b*. The proof of *Action2b* first opens *AcceptorMach*(a) to obtain the PSM invariants: $\boxed{a \hookrightarrow \bigcirc\!\!\blacksquare\ \bar{l}}$ and *Inv2b*$(\bar{l})$. It then uses a sequence of rules to manipulate the permissions (labeled "permission reasoning" in Figure 6): MERGEVOTED combines $\bigcirc\!\!\blacksquare$ with the supplied $\blacksquare$ permission to obtain the $\bullet$ permission; EXTENDVOTED extends a's voted list to l' by filling the old list with `None` up to index n and then appending `Some`(v) at index n; SPLITVOTED splits $\bullet$ back into $\bigcirc\!\!\blacksquare$ and $\blacksquare$. Finally, the proof returns the $\boxed{a \hookrightarrow \blacksquare\ l'}$ permission, reflecting the updated voted list.

In addition, the proof must also restore the PSM invariants of *AcceptorMach*(a). The main proof obligation,

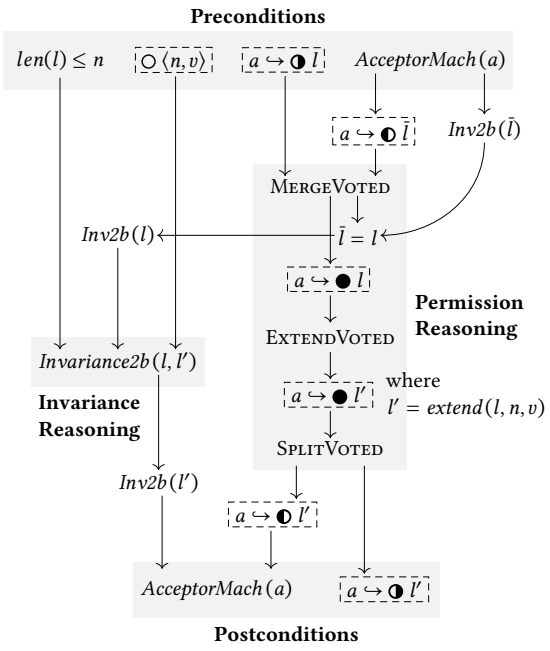


Figure 6: Proving validity of *Action2b*. The action is defined in Figure 5(b).

Invariance2b(l, l'), is to re-establish *Inv2b* for the newly extended list l'. In particular, it requires showing that $\langle n, v\rangle$ was indeed a valid proposal, which is established using the permission $\boxed{\bigcirc\langle n,v\rangle}$ supplied to *Action2b*.

Re-establishing *Inv2b* using $\boxed{\bigcirc\langle n,v\rangle}$ demonstrates a key benefit of PSM: an action proof does not require considering other invariants or actions, unless explicitly needed for the proof itself. For instance, the proof of *Action2b* does not have to consider *Inv2a* or any other invariants, except for *Inv2b*, which it needs to justify its steps. The use of stable permissions allows actions to precisely capture their requirements without global references. For example, *Action2b* requires that $\langle n, v\rangle$ be a valid proposal, represented by the stable permission $\boxed{\bigcirc\langle n,v\rangle}$, which can be obtained by first applying *Action2a* to get $\boxed{\circledcirc\langle n,v\rangle}$ followed by the FixProposal rule. In essence, the permission $\boxed{\bigcirc\langle n,v\rangle}$ captures the fact that once $\langle n, v\rangle$ was a valid proposal, it will always remain a valid proposal, and hence indirectly connects *Action2b* with *Action2a*. This allows the developer to decompose the overall proof of Paxos into smaller proofs that consider just one part of the system at a time: one proof that reasons about *Action2a* and *Inv2a* (and nothing else) to obtain $\boxed{\bigcirc\langle n,v\rangle}$, justifying that $\langle n, v\rangle$ is a valid proposal, and another proof, that of *Action2b* and *Inv2b*, described above, which does not consider *Inv2a*.

4.5 Integrating PSM into code proof

PSM allows developers to specify and verify their protocol separately from the implementation, and to then use the proven protocol actions in the proof of their concurrent implementation. This works well because PSM's notion of permissions corresponds directly to the notion of resources

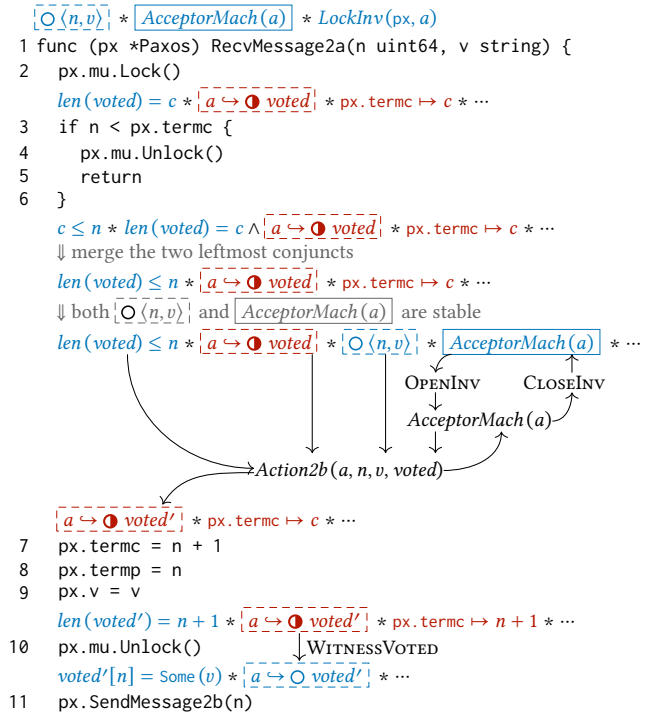


Figure 7: Proving `RecvMessage2a` using *Action2b*. Permissions established at each program point are shown in the figure. Stable permissions are shown in blue; non-stable permissions are shown in red. For clarity, details such as integer overflow and namespace management for program invariants are omitted.

in concurrent separation logic (CSL), and thus allows a CSL-based proof of a concurrent implementation to directly refer to PSM permissions and actions.

For example, consider the Go function `RecvMessage2a` that corresponds to *Action2b* in our running Paxos example, as shown in Figure 7. This function has three preconditions. The first, $\boxed{\bigcirc\langle n,v\rangle}$, is the stable permission established by the caller of `RecvMessage2a`. The second, $\boxed{AcceptorMach(a)}$, is the PSM acceptor specification sealed as a *program invariant*, representing facts that hold at every program step. The third, *LockInv*(px, *a*), is a *lock invariant* that holds except while the lock is held by some thread.

At line 2, the function starts by acquiring the lock to guard against race conditions. On acquiring the lock, the proof obtains the implementation permission $\texttt{px.termc} \mapsto c$, which asserts that the struct field `px.termc` has value c; the protocol permission $\boxed{a \hookrightarrow \text{◑ } voted}$; and the relationship between the two, $len(voted) = c$. Between lines 3–6, the code checks whether the acceptor has promised not to vote for any proposal numbered below n, and returns early if it has. Otherwise, the proof obtains $c \le n$. Together with the equality $len(voted) = c$ from the lock invariant, the proof obtains $len(voted) \le n$.

At this program point (after line 6), the proof has all the permissions required to invoke *Action2b*. The proof therefore opens the program invariant $\boxed{AcceptorMach(a)}$, passes the

required permissions to *Action2b*, recovers *AcceptorMach* (a) from its result, seals it back into the program invariant, and obtains $a \hookrightarrow ◑\ voted'$ for the updated voted list.

Between lines 7–9, the code updates the `px` struct to catch up with the permission $a \hookrightarrow ◑\ voted'$ so that the proof can re-establish the lock invariant and release the lock at line 10. In addition, using the permission interaction rule WitnessVoted, the proof extracts the stable permission $a \hookrightarrow ○\ voted'$ from $a \hookrightarrow ◑\ voted'$ as evidence that this acceptor has voted for the current proposal $\langle n, v \rangle$. Since the function released the lock on line 10, other threads might be modifying the *voted* abstract state (and the physical state in `px`) concurrently with `SendMessage2b` on line 11. However, the proof does not need to explicitly consider this, because the proof holds the stable permission $a \hookrightarrow ○\ voted'$, which is sufficient to allow the acceptor to reply by calling `SendMessage2b`.

4.6 Applicability of PSM

Using PSM requires the developer to express their protocol specification in terms of a permissioned state machine. In terms of generality, PSM can reason about any protocol that can be represented as a state machine, which is the same set of protocols that TLA+ can handle. However, the key advantage of PSM is that it improves modularity when the developer chooses suitable permissions that decompose the protocol into smaller modules. To this end, the two types of permissions that we have found particularly useful are *stable permissions* and *fractional permissions*.

Stable permissions. Distributed systems are commonly designed for a network that can arbitrarily delay and duplicate messages, so the system must be prepared to handle any transmitted message arbitrarily many times in the future. This typically requires each message to have an unambiguous stable meaning regardless of when the message is received in the future, which can be expressed using stable permissions.

Distributed systems are often built around the notion of a log, such as a set of replicas agreeing on a log of operations, or a per-key log of actions issued by applications in a key-value store. Such logs are a natural fit for stable permissions of log prefixes, since they grow only by appending.

Distributed systems sometimes have mutable state that does not grow monotonically (e.g., the contents of a distributed key-value store). However, we have found it useful to introduce an abstract state variable representing the monotonic history of operations over time, logically capturing the order of operations that have executed. A history abstract state allows developers to both use stable permissions of a history prefix (e.g., to capture the fact that the result of a key-value get observed by a client corresponds to some prefix of the history) as well as connect the abstract state to the physical mutable state (e.g., the physical key-value store corresponds to the latest version of the history).

Small changes to the protocol definitions can unlock monotonicity. For instance, Lamport's original definition of *Inv2b*† in TLA+ is formulated in terms of the complete set of network messages, rather than a voted list for each acceptor. As a result, *Inv2b*† is not monotonic with respect to the set of network messages because it requires that votes with certain proposal numbers are *not* present in the set. However, reformulating the action in terms of a monotonically growing voted list, with `None` values representing votes that will never be cast, unlocks the underlying monotonicity.

Finally, we have found a small set of data types that work well for representing monotonic states: namely, monotonic integers (such as transaction timestamps or proposal numbers in Paxos), monotonic lists (such as logs), monotonic maps (such as commit decisions), monotonic sets (such as a set of prepared shards in two-phase commit), and monotonic option types (which can change from `None` to `Some` (v) but not the other way around). §5 shows how these types of monotonic states are used in the Tulip case study.

Fractional permissions. When module boundaries are within a single machine, we often find that fractional permissions are a good fit to represent the sharing pattern. For example, fractional permissions are a good fit for connecting protocol-level reasoning to code-level reasoning, such as the $a \hookrightarrow ◐\ l$ and $a \hookrightarrow ◑\ l$ permissions connecting the acceptor protocol invariant to the implementation state protected by a mutex lock. Similarly, fractional permissions show up in our Tulip case study when connecting the implementation of the Tulip protocol to the file library, which saves the implementation's state to persistent storage.

Limitations. PSM requires a framework that supports reasoning about ownership of permissions; our prototype is built on top of Iris and Grove. For projects where switching to such a framework requires substantial effort, the benefits might not be worth it if (1) there is no aim to connect the protocol to a concurrent implementation, and (2) either the protocol in question has a small number of actions and invariants, or the protocol stack has clean layering that can be captured through coarse-grained refinement rather than PSM's fine-grained sharing. For instance, clean abstractions, such as IronFleet's IronRSL [6] or Verdi's VST [28], provide modularity without PSM-style reasoning. On the other hand, verifying replication protocols with sophisticated optimizations, such as TAPIR [31] or EPaxos [19], would likely benefit from using PSM-style reasoning. Finally, we have not explored the applicability of PSM to liveness reasoning.

5 Design and verification of Tulip

Tulip provides a library for writing transactional applications on top of a multi-version key-value store sharded across many replica groups for scalability, and replicated within each replica group for fault tolerance. Applications run on

Operation	Description
`db.Begin() -> txn`	Start a transaction on database `db`
`txn.Commit() -> ok`	Commit transaction `txn`
`txn.Abort()`	Abort transaction `txn`
`txn.Read(k) -> v`	Read current value of key `k`
`txn.Write(k, v)`	Write `v` as new value of key `k`
`txn.Delete(k)`	Delete current value of key `k`

Figure 8: Tulip client library API.

client machines and invoke the Tulip client library to begin, abort, and commit transactions, and to read, write, and delete key-value pairs within each transaction, as shown in Figure 8. The client's job is to interact with the Tulip servers to execute this transaction in a way that achieves strict serializability. Tulip proves a top-level theorem that captures strict serializability [4: §3.2]: that is, any transactional code executed under Tulip appears to execute atomically at some point between its start and end time, observing a database state where all of the transactions before this one have committed, and none of the transactions after this one have made any changes.

Each Tulip server is part of a replica group, whose job is to maintain a shard of the database (an MVCC key-value store). The replica group agrees on a consistent view of its shard using a Paxos log, which records transaction commit and abort decisions, as shown in Figure 9. Each replica also maintains state about active transactions that have not yet committed or aborted—namely, which keys those transactions have read and what writes a transaction may be attempting to commit. This state is potentially *inconsistent* across a replica group, since each replica has its own version of this inconsistent-replication (IR) log. Both the Paxos and IR logs are stored durably on disk to handle crashes.

Figure 9 illustrates the overall client-server interaction in Tulip. To execute read operations, clients directly contact all servers in the target replica group, to execute the request in a single network round-trip (e.g., c_1 in Figure 9). On receiving a read request for some key, the replica immediately executes it and replies with the result. The replica also needs to remember that this transaction read a particular version of this key, so that logically earlier transactions cannot commit a conflicting earlier write. To do this, the replica records in its IR log an entry consisting of the request and the point at which the request is executed, illustrated by the $\mathcal{R}$ with a downward-pointing arrow in Figure 9. This entry is *inconsistent* because it might not appear on every replica, or it might appear at different positions relative to the Paxos log.

When the application is ready to commit, the client initiates a two-phase commit (2PC) process. The client first computes the *participant groups*—the set of replica groups hosting the data shards to be modified by the transaction. It then issues a prepare request to each group, and if all are willing to commit, follows up with a commit request to all participant groups.

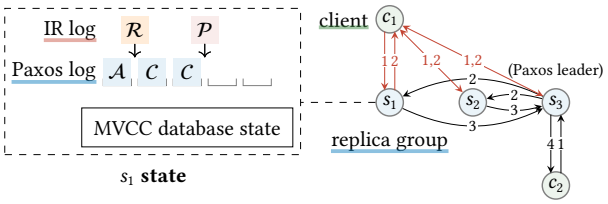


Figure 9: Client-server interaction in Tulip. Each replica server maintains a Paxos log and an IR log, where each IR log entry is logically ordered after some Paxos log entry. Both logs are durably stored on disk. As an optimization, each replica maintains an MVCC database state, which is a function of the Paxos and IR logs and can be reconstructed after a crash.

Tulip uses inconsistent replication for prepare requests, allowing the prepare phase to complete in a single round-trip in the common case. In the commit phase, the client (e.g., c_2 in Figure 9) interacts with the Paxos leader of each group, which in turn executes the commit request in a consistent order across all replicas in that group. Importantly, replicating and executing commit requests in the commit phase is not on the critical path for transaction latency: once a transaction has prepared on all participant groups, it is guaranteed to commit, even if the client crashes. As a result, the commit phase happens in the background, after the client has returned to the application.

If the client fails after preparing at least one group, but before committing or aborting the transaction, servers in the prepared groups take over as backup 2PC coordinators for the transaction, and either commit or abort it as appropriate.

In this section, we give an example of how Tulip's proof uses PSM. Appendix A gives a more complete description of Tulip's design and proof.

5.1 Verifying Tulip using PSM

Tulip makes extensive use of permissioned state machines to break up the overall Tulip system into many largely independent modules, both at the code level and at the specification and proof level. There are 10 types of modules in Tulip: 6 types for different parts of the transaction system, and 4 types for different parts of the Multi-Paxos library. The transaction-level modules are: a separate module for each replica, a separate module for the persistent on-disk state of each replica, a separate module for each group of replicas, a separate module for each key in the key-value store, one module for the Tulip network messages, and one module for the overall transaction system. The Multi-Paxos library modules are: one module for all the proposers, a separate module for each acceptor, a separate module for the persistent on-disk state of each participant machine, and one module for the Multi-Paxos network messages. The dependencies between these modules are represented by monotonic states as shown in Figure 10.

Since we do not have space to describe all of Tulip's modules, we focus on one illustrative example below, to show how Tulip's proofs use PSM. Other examples, as well as more details on the Tulip protocol, can be found in Appendix A.

Abstract state	Semantics	Owner module
Prepare map (§5.2)	Group's decision to prepare a given transaction	Each group
Validation set (§5.2)	Replica's decision to validate transaction's write-set at its timestamp	Each replica
Ranked prepare decisions (§5.2)	Replica's decision to prepare or unprepare a transaction at each rank	Each replica
Transaction result map (§A.1)	Coordinator's decision to commit or abort a given transaction	Txn system
Prepare proposal map (§A.2)	Coordinator's proposal to prepare or unprepare a transaction in a group	Each group
Value history (§A.3)	History of values for a given key across timestamps	Each key + group
Validation history (§A.3)	Replica's validation decisions for transactions modifying a given key	Each replica
Command pool (§A.4)	Set of commands submitted to the Paxos library for appending to the log	Paxos + group
Command log (§A.4)	Append-only log of commands agreed upon by a replica group	Paxos + group

Figure 10: Selected abstract state variables introduced by Tulip to enable stable permissions.

5.2 Preparing to commit in a single round-trip

The prepare phase in Tulip's 2PC protocol serves two purposes: ensuring *mutual exclusion*, meaning that at any point in time, no more than one committing transaction may modify a given key; and achieving *stable prepare decisions*, where all 2PC coordinators—including the client and any backup coordinators—are guaranteed to reach the same decision. We now describe Tulip's 2PC prepare procedure, including a fast path and a slow path.

Fast-path prepare. The client for transaction t serves as the initial coordinator and begins the prepare phase by sending $\langle \textsc{FastPrepare}, t, w_g, G\rangle$ as an inconsistent operation to all replicas in each participant group $g \in G$, where w_g is the transaction's write-set on group g (G is included in the message for coordinator recovery; see §A.2).

On receiving $\langle \textsc{FastPrepare}, t, w_g, G\rangle$, each replica first checks locally if t has already committed or aborted (which can happen when a backup coordinator becomes live and proceeds to commit or abort t). If so, the replica simply responds with $\langle \textsc{Committed}, t\rangle$ or $\langle \textsc{Aborted}, t\rangle$. The transaction client immediately reports the result and terminates on receiving these messages.

Otherwise, each replica performs the following checks to validate transaction t: for each $k \in dom(w_g)$, (1) k is not locked by another transaction, and (2) k has not been read or validated by any transaction $t' > t$. If these checks pass, t acquires the lock for each $k \in dom(w_g)$ and is said to be *validated* on this replica. Validating on a classic quorum of replicas (i.e., $\lceil \frac{n+1}{2} \rceil$ in our setup, where n is the number of replicas in the replica group) ensures mutual exclusion, as no two quorums of replicas can simultaneously grant the same lock to two distinct transactions. The replica replies $\langle \textsc{FastResult}, t, p, b\rangle$, normally with $p = b = \top$ if validation succeeds and $p = b = \bot$ otherwise; p and b may diverge if the replica previously processed another FastPrepare for t.

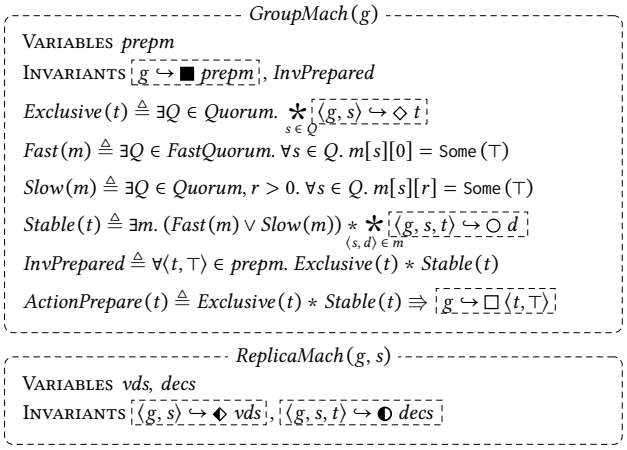
In the common case where transaction t obtains positive fast-prepare responses $\langle \textsc{FastResult}, t, p = \top, b = \top\rangle$ from some fast quorum (i.e., $\lceil \frac{3}{4} n \rceil$ in our setup), t is said to have successfully prepared on the target group.

Slow-path prepare. Tulip also allows progress with just a classic quorum of replicas; we call this the slow path for prepare. The slow path runs in parallel with the fast path: the client proceeds to the slow path once the following two preconditions are met: (1) mutual exclusion has been achieved, and (2) the fast path has not—and will not—agree to unprepare. The client fulfills the first condition by waiting until it has validated the transaction on a classic quorum of replicas. To fulfill the second condition, the client must exhibit a classic quorum in which no more than half (i.e., $\lfloor \frac{c}{2} \rfloor$ where c is the size of the classic quorum) of the replicas have agreed to unprepare. Contingent upon the two conditions above, the client sends $\langle \textsc{Prepare}, t, r = 1\rangle$ to all replicas in the target group. The parameter r denotes the *rank* of the transaction coordinator. Tulip unifies the prepare interface for transaction clients and backup coordinators: rank 0 corresponds (implicitly) to the client's fast-path prepare, 1 to the slow-path prepare, and higher ranks to backup coordinators. We provide a more detailed discussion about ranks in §A.2.

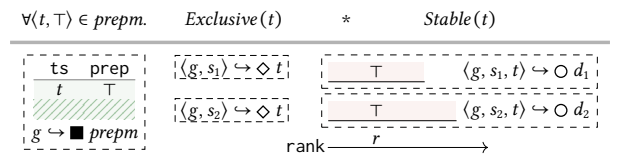
On receiving $\langle \textsc{Prepare}, t, r = 1\rangle$, each replica records transaction t as prepared at rank 1—provided it has not seen a higher-rank coordinator—and responds with $\langle \textsc{Prepared}, t, r = 1\rangle$. Once a classic quorum responds positively, the client concludes the slow-path prepare.

Verification using PSM. Recall that the two main correctness conditions to establish in Tulip's 2PC prepare phase are mutual exclusion and stability of prepare decisions. We capture these conditions with the invariant *InvPrepared* defined in Figure 11(a). The invariant is centered on the per-group state variable *prepare map*, whose value is a map from transaction timestamps to their corresponding *prepare decisions* on a certain group. Each prepare decision can either be "prepared" ($\top$) or "unprepared" ($\bot$). The invariant requires that each prepared entry in the prepare map (i.e., $\forall \langle t, \top\rangle \in prepm$) be accompanied by certain forms of stable permissions specified by the $Exclusive(t)$ and $Stable(t)$ predicates. Figure 11(b) visualizes an example set of stable permissions satisfying the two predicates separately on group g. Below, we explain what the stable permissions are and the intuition behind why they achieve their respective goals.

For mutual exclusion, $Exclusive(t)$ requires the stable permission $\boxed{\langle g, s\rangle \hookrightarrow \diamond\, t}$ for s in some classic quorum of replicas. This stable permission is defined over the per-replica *validation set*, a monotonic set of transaction timestamps.



(a) Partial PSM specifications for replica groups and replica servers. The action *ActionPrepare* is simplified to omit permissions orthogonal to Tulip's 2PC prepare, such as MVCC linearization [4] and crash recovery. *Quorum* and *FastQuorum* indicate classic and fast quorums [15], respectively.



(b) An example state satisfying the *InvPrepared* invariant. Each part of the invariant definition shown above the gray horizontal line is paired with a concrete visual representation below it. It assumes the group consists of three servers s_1, s_2, and s_3; hence, $\{s_1, s_2\}$ forms a classic quorum.

Figure 11: PSM specifications capturing correctness of Tulip's 2PC prepare.

Intuitively, t in the validation set of replica s means that t has passed timestamp validation and acquired the locks at s.

One subtle but important question arises: how can the validation set be monotonic when each transaction must release its locks on commit or abort? The trick is to assign *history semantics* to the validation set. That is, instead of defining $\langle g, s\rangle \hookrightarrow \diamond\, t$ to mean "t is currently holding the locks on replica s", we define the stable permission to mean "t has, at some point, obtained its required locks on replica s". This small change allows us to formulate stable permissions about the fact that t had acquired the necessary locks, rather than referring to the transient contents of the lock table. This validation set is purely logical, introduced only for proof purposes; the program does not physically store it.

For prepare stability, $\mathit{Stable}(t)$ requires the stable permission $\langle g, s, t\rangle \hookrightarrow \bigcirc\, l$ from some classic quorum of servers such that each list l in the quorum contains $\mathtt{Some}(\top)$ at some index $r > 0$, or a fast quorum such that each list in the quorum contains $\mathtt{Some}(\top)$ at index 0. The classic-quorum part maps to the slow path in Tulip's 2PC prepare and is specified by the *Slow* disjunct, whereas the fast-quorum part represents the fast path and is specified by the *Fast* disjunct (Figure 11(b) shows only the slow path).

The stable permission $\langle g, s, t\rangle \hookrightarrow \bigcirc\, l$ is defined over another per-replica state variable called *ranked prepare decisions*. Its value is a map from transaction timestamps to the replica's decisions to prepare or unprepare a certain transaction at each rank. The decisions are maintained as a list of optional booleans, a structure similar to the voted list we saw in the Paxos example in §4: $\mathtt{Some}(\top)$ (or $\bot$) at index r means "prepared" (or "unprepared") at rank r, and $\mathtt{None}$ at r means no prepare decision at that rank. In fact, Tulip uses a special case of Fast Paxos [15] to achieve prepare stability.

We conclude by describing the main actions used in Tulip's 2PC protocol, including *ActionPrepare*, shown in Figure 11(a). As reflected in its definition, the purpose of *ActionPrepare* is to insert an entry $\langle t, \top\rangle$ into the prepare map and extract the stable permission $g \hookrightarrow \square\, \langle t, \top\rangle$. The invariant *InvPrepared* requires that we do so in a way that ensures exclusivity and stability. Therefore, *Exclusive* and *Stable* also appear as preconditions of *ActionPrepare*. The stable permissions required to establish these requirements, including $\langle g, s\rangle \hookrightarrow \diamond\, t$ and $\langle g, s, t\rangle \hookrightarrow \bigcirc\, l$ as discussed above, come from applying replica-level actions and are associated with messages such as FastResult. Finally, by applying *ActionPrepare* and obtaining $g \hookrightarrow \square\, \langle t, \top\rangle$ for each participant group g, we can proceed to Tulip's commit phase.

Component	Impl (Go)	Proof (Rocq)	
		Code	Protocol
Transaction	882	6177	4442
Backup coordinator	674	4445	
Replica	669	5719	2501
Transaction log	76	306	678
Multi-Paxos	748	4929	4148
MVCC tuple	88	863	1862
Message	517	1186	-
Misc	302	1613	-
Common definitions	-	-	1491
General lemmas	-	-	1771
Total	3956	25238	16893

Figure 12: Summary of Tulip's development in lines of code.

6 Verification evaluation

As summarized in Figure 12, the implementation of Tulip consists of 3,956 lines of Go, and the proof consists of 42,131 lines of Rocq (both excluding comments and blanks). The implementation and proof build on the Grove framework for verifying distributed systems [25], and in particular use Grove's network, file system, and RPC libraries.

Protocol proof. Tulip's protocol proof uses 21 state variables shared across modules; 13 of them are shared via stable permissions over monotonic states (such as the ones shown in Figure 10), and 8 of them are shared via fractional permissions (such as fractional ownership of on-disk replica state) or other kinds of permissions. (We denoted fractional permissions using ◐ and ◑ in §4.) Figure 13 illustrates how Tulip's state is shared between different types of modules. Each box represents a type of module (even if that module might be then instantiated many times, such as a module representing one of the replicas). A full circle on the side of a

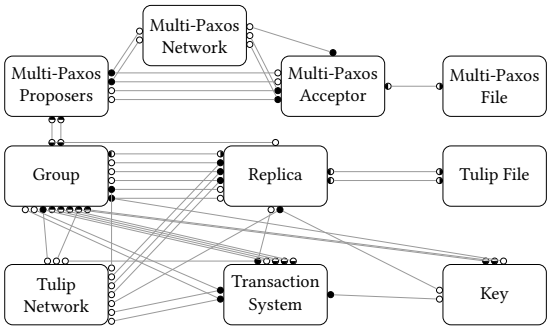


Figure 13: Diagram illustrating how state is shared among Tulip's modules.

box indicates that full ownership of a state variable belongs to that module; an empty circle indicates that some invariant in that module is defined over the stable permission of that variable; a half circle indicates half ownership.

The benefit of Tulip's modularity is shown in Figure 14, which reports the number of actions that need to access the states and invariants of different combinations of modules. Of the 25 actions in total, 13 depend on a single module, which achieves ideal modularity: proving these actions requires purely local reasoning within their own module. Even for actions that depend on multiple modules, permissioned state machine is still helpful, in two ways. First, actions depend on a small number of modules (e.g., 6 out of 25 actions depend on two modules, so their proofs need to consider just those two modules out of all 10 types of modules). Second, in many cases developers can prove the validity of actions module-by-module, without simultaneously opening multiple modules. Sometimes the developer needs to open multiple modules, but they can take just the required permissions out of the module, rather than moving all the variables and invariants into their proof context.

The use of stable permissions enabled us to verify Tulip in a largely modular fashion. Once we established the abstract states and stable permissions at the boundary of certain components, we verified their implementations independently of one another. As one example, the Multi-Paxos library provides a specification stated in terms of its command pool and agreed-upon log abstract state variables, and its proof is independent of the rest of Tulip. Although Tulip's 2PC protocol uses the library, the library's proof is not dependent on the details of the commands and predicates that 2PC needs.

Code proof. Code proofs account for about 60% of the total proof by line count. A substantial portion of the code proof consists of mechanical steps, such as symbolic execution of program statements. The most time-consuming part of the code proof is in identifying the right lock and loop invariants.

Bugs caught during verification. One bug that we discovered during verification is related to transactional reads. Initially, we designed the replica-level read to use timestamp 0 as an indicator for hitting the fast path, with the goal of returning one fewer boolean value. This design, however,

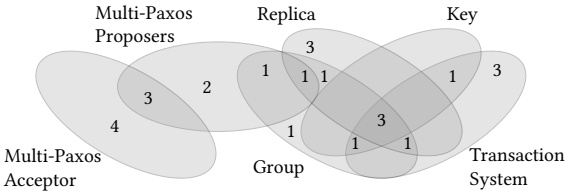


Figure 14: Each number denotes the number of actions that depend on a certain combination of modules. Network modules are updated only by single-module actions. File modules are updated in tandem with either the Tulip replica or the Multi-Paxos acceptor module.

was unable to distinguish a fast read from a quorum read that reads the very first version, and was hence incorrect.

Another bug was due to transactions releasing locks that they had not acquired. Unlike running 2PC over Paxos, where each prepared transaction must have obtained locks on all replicas by the time the transaction commits, running 2PC over IR requires transactions to obtain locks only on some classic quorum of replicas. Our initial implementation of Tulip (which was derived from an implementation of running 2PC over Paxos) missed the essential check to test whether the transaction had actually been prepared on a certain replica, violating correctness of concurrency control.

Finally, we discuss the "timestamp-inversion" bug [18] discovered in some transaction systems, including TAPIR. Timestamp inversions happen when a transaction linearizes (i.e., applies its effect to the logical state) before another smaller-timestamp transaction. This violates a core principle of timestamp-based concurrency control: linearization order should respect timestamp order. Had Tulip been affected by timestamp inversions, the proof would have gotten stuck when extending the MVCC value history.

6.1 Developer experience

In the remainder of this section, we reflect on our experience of applying PSM to Tulip.

Understanding the system. Applying PSM begins with identifying the abstract state, which requires understanding the sub-protocols that make up the system and their interactions. For instance, when verifying Tulip's 2PC prepare, as discussed in §5.2, we initially overlooked two entangled properties—mutual exclusion and prepare stability—that the 2PC protocol should achieve; moreover, stability can be modeled as a special case of Fast Paxos [15]. Once we had these insights, it became clear what abstract state we needed.

Breaking protocols into actions. In the proof planning stage, the most important task is to break the protocol into several actions, with each action fulfilling a specific purpose in the overall protocol. For instance, we break the Tulip 2PC process (the successful case) roughly into the following actions: (1) the transaction client initiates the prepare phase by contacting replicas in the participant group; (2) a replica promises to prepare the transaction; (3) once obtaining enough promises from replicas in a group, the client sets

that group as prepared (i.e., *ActionPrepare* in Figure 11); (4) once all participant groups are prepared, the client sets the transaction as committed; (5) the client submits a commit record to the underlying Multi-Paxos library. Furthermore, each action often produces one or two stable permissions that clarify the purpose of the action. For instance, $\langle g, s \rangle \hookrightarrow \Diamond\, t$, mentioned in §5.2, encodes the promise by replica s in group g that it has validated t.

Implementation-guided proof planning. In our experience, protocol proofs often take more time than program proofs, which are mostly mechanical and require less design and planning. To avoid committing too early to an incorrect PSM protocol formulation, we find it useful to first write an implementation and annotate where in the code each action would be used before actually proving the action lemmas (e.g., between lines 6 and 7 in Figure 7), which suggests how the action lemmas will be consumed. When the program proofs get stuck, we can often fix just the implementation without having to revise the PSM formulation. Occasionally, we did go back to fix the PSM formulation while doing program proofs, but the fixes were often localized to a few actions or invariants without affecting the remaining proofs.

7 Performance evaluation

To confirm that Tulip's design indeed implements a high-performance sharded transactional key-value store with sophisticated optimizations mirroring those of TAPIR, we performed several experiments to demonstrate the following:

1. Tulip achieves throughput and latency that are at least as good as TAPIR's.
2. Tulip achieves low latency in a wide-area setting, where the latency between replicas is significant.
3. Tulip also achieves low latency in a datacenter setting, where the network latency is low.
4. Tulip continues executing despite replica failures, and resumes executing on a replica after recovery.

7.1 Experimental setup

To evaluate Tulip's performance, we run experiments on AWS, measuring performance in two different settings: wide-area and single-datacenter. In the wide-area setting, we used instances in three different regions (Northern Virginia in the US, Stockholm in Europe, and Singapore in Asia Pacific). The datacenter experiments run all instances in the US region. The latency between servers in the three wide-area regions is shown in Figure 15, as measured by `ping`. The latency between servers within the same datacenter is 200–400 μs. We run all of the experiments with Tulip's storage in a ramdisk (tmpfs), in order to focus on protocol overheads, even though Tulip does have a verified crash-safe storage implementation. We use r7a/i.4xlarge EC2 instances for both clients and servers; each instance has 16 vCPUs and 128 GiB of main memory. We run all the clients on the same instance. In a given replica group (shard), each replica runs on a different instance; replicas from different groups share instances.

	US	Europe	Asia Pacific
US	<1 ms	109 ms	218 ms
Europe		<1 ms	178 ms
Asia Pacific			<1 ms

Figure 15: Round-trip latency between AWS regions used in the evaluation.

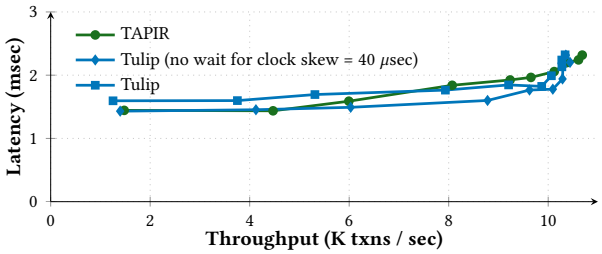


Figure 16: Throughput versus latency for the Retwis workload with Tulip clients and servers located in the same datacenter. The Retwis workload models a Twitter-like application.

We compare Tulip with a baseline system resembling Spanner that we call "2PC+Paxos," which we implemented. For the purposes of our experiments, the Paxos leader in this baseline system always runs in the US region. We also compare Tulip with TAPIR, using the code provided by the TAPIR authors at `https://github.com/UWSysLab/tapir`.

7.2 Datacenter latency

To evaluate Tulip's performance in a datacenter, we compare it to TAPIR, using the Retwis workload [17] in the configuration used by the TAPIR authors [31]. We used three servers, all in the same datacenter and availability zone. We use 24 Retwis clients, with keys and values both 64 bytes long, as in the TAPIR paper. Keys are drawn from a uniform distribution. Each run of the benchmark takes 10 seconds.

One difference between Tulip and TAPIR is that Tulip implements strict serializability, while TAPIR implements (non-strict) serializability. To provide strict serializability, Tulip waits for the maximum clock skew to elapse before returning from transaction commit, which TAPIR does not do. To provide a fair comparison, we implement and evaluate a variant of Tulip that does not perform this wait, achieving the same (non-strict) serializability guarantee as TAPIR.

Figure 16 shows the results, measuring latency at different throughputs. Tulip's non-strict serializability variant achieves effectively the same performance as TAPIR. Tulip's strict-serializability variant achieves slightly higher latency, due to the clock-skew wait; the maximum clock skew for the AWS instances used in our experiment is 40 μs.

7.3 Wide-area latency

To evaluate the latency benefits of Tulip's protocol, we considered a scenario with five total replicas: two in the US, two in Europe, and one in Asia Pacific. We evaluate a microbenchmark transaction that writes to a single key and commits. We repeatedly execute the same transaction for 30 seconds, and report the average observed latency. Both keys

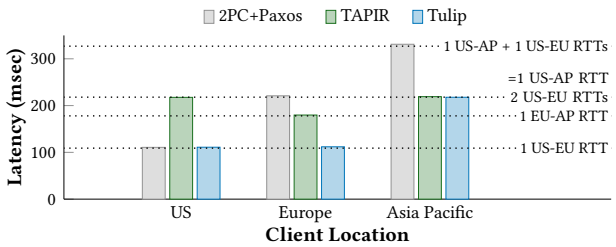


Figure 17: Latency for committing a single-key write transaction across geographically distributed datacenters.

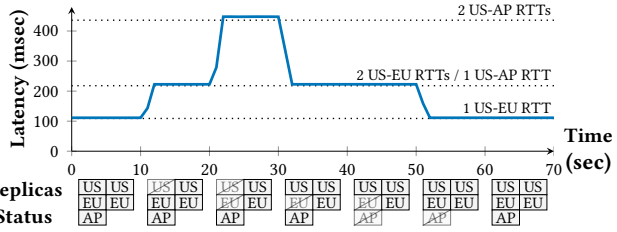


Figure 18: Latency of a single-write transaction over time with replicas crashing and recovering, as shown in the bottom status row.

and values are 64 bytes. Since only one key is involved in the microbenchmark, only one shard participates in the transaction. We compare Tulip's latency to that of 2PC+Paxos and TAPIR. For 2PC+Paxos, the leader is on a server in the US region. We measure the latency observed by clients running in each of the three regions.

Figure 17 shows the latency of committing a write transaction, as a function of the client location. Tulip achieves much lower latency for commit than 2PC+Paxos when the client is in a different datacenter than the Paxos leader. When the client is in the US region, both 2PC+Paxos and Tulip achieve the same latency, because both wait for a response from Europe to achieve quorum. When the client is in the Europe or Asia Pacific region, Tulip allows the client to directly contact other datacenters and collect a quorum, whereas 2PC+Paxos requires going through the leader in the US region; the differences correspond directly to the latencies seen in Figure 15. This shows that both our Tulip implementation and the 2PC+Paxos implementation are bounded by the underlying wide-area network latency, and Tulip's protocol reduces the overall transaction latency.

Tulip also achieves lower latency for commit than TAPIR, when the client is in the US or Europe regions. To the best of our understanding, TAPIR could in principle commit in a single US-EU round-trip, because that provides enough replicas to form a fast quorum. However, TAPIR does not seem to implement this optimization, and instead waits for all replicas to reply (hence the 1 US/EU-AP RTT latency). Tulip does implement this optimization, allowing it to commit in a single US-EU RTT. When the client is in the Asia Pacific region, this optimization does not matter since no quorum can be reached before 1 US/EU-AP RTT, and after that time, all replicas are reached.

7.4 Failure recovery

To demonstrate Tulip's performance as replicas crash and recover, we continuously execute a single-write transaction (as in the wide-area latency experiment above), and monitor its latency. We start in the same configuration as the wide-area experiments above: five total replicas, with two in the US, two in Europe, and one in Asia Pacific. The client is always in the US region. We then perform the following actions at 10-second increments: crash a US replica; crash an EU replica; recover the US replica; crash an AP replica; recover the EU replica; recover the AP replica.

Figure 18 shows the results. At $t = 10$ s, the latency goes up because the client must now wait for a response from the AP replica to form a fast quorum. At $t = 20$ s, with only 3 live replicas left, no fast quorum can be formed, forcing Tulip to execute in two round-trips, with each round-trip requiring a US-AP RTT to reach a full classic quorum. At $t = 30$ s, the US replica recovers, allowing the client to observe a fast quorum from 4 replicas in a single US-AP RTT. At $t = 40$ s, the AP replica crashes, leaving the system with just 3 live replicas again. This forces Tulip to again fall back to committing in two round-trips, with each round-trip requiring a US-EU RTT. It so happens that two US-EU RTTs are almost the same as a single US-AP RTT, which is why the latency appears not to change. At $t = 50$ s, the EU replica recovers, allowing Tulip to again commit in a single US-EU RTT. At $t = 60$ s, the AP replica recovers, which does not change the latency because a fast quorum does not require the US client to wait for a response from the AP replica.

We do not compare TAPIR's handling of replica crashes because its implementation lacks support for replica recovery and synchronization, as confirmed by TAPIR's authors.

8 Conclusion

Tulip is a transactional key-value store that supports sharding, replication, crash-safe storage, MVCC, and coordinator recovery, achieves performance competitive with that of TAPIR, and comes with a machine-checked proof of correctness (strict serializability) for its implementation written in Go. To verify Tulip, we introduced the PSM approach, which combines TLA-style state-machine reasoning with permissions. PSM allowed us to decompose Tulip's proof into 10 types of modules, with explicit permissions capturing the dependencies between the modules. Permissions eliminate the need for the developer to consider the full cross-product of all actions and invariants: most actions in Tulip's protocol proof depend on just one module.

Acknowledgments

We thank Tej Chajed, Baltasar Dinis, Jon Howell, Dongjae Lee, Derek Leung, Natalie Neamtu, Bryan Parno, the anonymous reviewers, and especially our anonymous shepherd, for feedback that improved this paper. This work was supported by NSF awards 2319167 and 2524668, and by a gift from Amazon.

References

[1] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical report, Laboratory for Computer Science, University of Bologna, Italy, 1993.

[2] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, Huntsville, Ontario, Canada, Oct. 2019.

[3] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent Go code in Coq with Goose. In *Proceedings of the 6th International Workshop on Coq for Programming Languages (CoqPL)*, New Orleans, LA, Jan. 2020.

[4] Y.-S. Chang, R. Jung, U. Sharma, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2023.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[6] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Monterey, CA, Oct. 2015.

[7] J. M. Hellerstein and P. Alvaro. Keeping CALM: when distributed consistency is easy. *Communications of the ACM*, 63(9):72–81, Aug. 2020.

[8] W. Honoré, J. Kim, J. Shin, and Z. Shao. Much ADO about failures: a fault-aware model for compositional verification of strongly consistent distributed systems. In *Proceedings of the 36th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Chicago, IL, Oct. 2021.

[9] W. Honoré, J.-Y. Shin, J. Kim, and Z. Shao. Adore: Atomic distributed objects with certified reconfiguration. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2022.

[10] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[11] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Proceedings of the 8th ACM EuroSys Conference*, Prague, Czech Republic, Apr. 2013.

[12] R. Krebbers, J. Jourdan, R. Jung, J. Tassarotti, J. Kaiser, A. Timany, A. Charguéraud, and D. Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 77:1–30, St. Louis, MO, Sept. 2018.

[13] M. Krogh-Jespersen, A. Timany, M. E. Ohlenbusch, S. O. Gregersen, and L. Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In *Proceedings of the 29th European Symposium on Programming (ESOP)*, pages 336–365, Dublin, Ireland, Apr. 2020.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[15] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2): 79–103, 2006.

[16] L. Lamport. The Paxos algorithm, or how to win a Turing award, 2024. https://lamport.azurewebsites.net/tla/paxos-algorithm.html.

[17] C. Leau. Spring Data Redis - Retwis-J, 2013. https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[18] H. Lu, S. Mu, S. Sen, and W. Lloyd. NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2023.

[19] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, Farmington, PA, Nov. 2013.

[20] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007.

[21] A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *Proceedings of the 7th ACM SIGPLAN Workshop*

on Types in Language Design and Implementation (TLDI), page 73–86, New York, NY, 2011.

[22] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Massachusetts Institute of Technology, Sept. 1978. http://hdl.handle.net/1721.1/16279.

[23] I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. In *Proceedings of the 45th ACM Symposium on Principles of Programming Languages (POPL)*, pages 28:1–28:30, Los Angeles, CA, Jan. 2018.

[24] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS)*, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] U. Sharma, R. Jung, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Grove: a separation-logic library for verifying distributed systems. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, Oct. 2023.

[26] J. Shin, J. Kim, W. Honoré, H. Vanzetto, S. Radhakrishnan, M. Balakrishnan, and Z. Shao. WormSpace: A modular foundation for simple, verifiable distributed systems. In *Proceedings of the 10th ACM Symposium on Cloud Computing (SOCC)*, pages 299–311, Santa Cruz, CA, Nov. 2019.

[27] A. Timany and L. Birkedal. Reasoning about monotonicity in separation logic. In *Proceedings of the 10th International Conference on Certified Programs and Proofs*, pages 91–104, Virtual conference, Jan. 2021.

[28] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, Portland, OR, June 2015.

[29] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, Mar. 2017.

[30] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication (extended version). Technical Report UW-CSE-14-12-01 (v2), University of Washington, School of Computer Science and Engineering, Seattle, WA, Dec. 2014. https://syslab.cs.washington.edu/papers/tapir-tr-v2.pdf.

[31] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[32] T. N. Zhang, K. Singh, T. Chajed, M. Kapritsos, and B. Parno. Basilisk: Using provenance invariants to automate proofs of undecidable protocols. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2025.

A Other aspects of Tulip's design (not peer-reviewed)

This appendix explains other important aspects of Tulip's design. Following the structure of §5.2, we first discuss the mechanics of each sub-protocol and then explain the main permissions, invariants, and actions used in its formalization. Note that the descriptions are inevitably incomplete—and at times imprecise—with respect to the actual Rocq development; our goal is to highlight some of Tulip's design decisions and explain how they support correctness proofs in PSM. Refer to https://github.com/mit-pdos/tulip for Tulip's code and https://github.com/mit-pdos/tulip-proof for its full Rocq development.

A.1 Committing a transaction

Once a client successfully prepares transaction t on all participant groups, it sends $\langle \textsc{Commit}, t, w_g \rangle$ to the Paxos leader of each group, where w_g is transaction t's partial write-set on group g. Otherwise, if preparation fails on any group, it sends $\langle \textsc{Abort}, t \rangle$. Tulip (as in prior work [11, 31]) adopts an optimization to reduce the transaction latency by eliminating commit replication from the critical path.

This optimization is enabled by the *commit requirement*, which demands that the outcome of transaction t be a function of the prepare decisions of all t's participant groups G. More specifically, t commits if every group $g \in G$ prepares; t aborts if some $g \in G$ unprepares.

An important implication of this requirement is that, after making all prepare decisions stable, a transaction client can immediately report the outcome while replicating it in the background. Even if the client fails before the outcome is successfully replicated, any backup coordinator will eventually derive the same outcome and finalize the transaction.

Verification using PSM. We capture the commit requirement by introducing the *transaction result map*, a new abstract state variable whose value is a map from transaction timestamps to their corresponding outcomes: committed ($\top$) or aborted ($\bot$). As shown in Figure 20, full ownership of this map, denoted as $\boxed{\blacktriangledown\ resm}$, is assigned to the PSM module *TxnSystemMach*. We formulate the commit requirement by

Abstract state	Permission	Semantics	Owner	Stable?
Prepare map	$g \hookrightarrow \blacksquare\ \mathit{prepm}$	Group g's prepare map is *prepm*	*GroupMach*(g)	No
	$g \hookrightarrow \square\ \langle t, p \rangle$	Group g's prepare decision for transaction t is fixed to p	*TxnSystemMach*	Yes
Validation set	$\langle g, s \rangle \hookrightarrow \text{⬖}\ \mathit{vds}$	Replica s's validation set is *vds*	*ReplicaMach*(g, s)	No
	$\langle g, s \rangle \hookrightarrow \text{⬗}\ \mathit{vds}$	Replica s's validation set is *vds*	*LockInv*(rp, g, s)	No
	$\langle g, s \rangle \hookrightarrow \lozenge\ t$	Transaction t has been validated on replica s	*GroupMach*(g)	Yes
Ranked prepare decisions	$\langle g, s, t \rangle \hookrightarrow \text{●}\ \mathit{decs}$	Replica s's ranked prepare decisions for transaction t are *decs* and can be extended	No owner	No
	$\langle g, s, t \rangle \hookrightarrow \text{◐}\ \mathit{decs}$	Replica s's ranked prepare decisions for transaction t are *decs*	*ReplicaMach*(g, s)	No
	$\langle g, s, t \rangle \hookrightarrow \text{◑}\ \mathit{decs}$	Replica s's ranked prepare decisions for transaction t are *decs*	*LockInv*(rp, g, s)	No
	$\langle g, s, t \rangle \hookrightarrow \text{○}\ d$	Replica s's ranked prepare decisions for transaction t have prefix d	*GroupMach*(g)	Yes
Transaction result map	$\blacktriangledown\ \mathit{resm}$	The transaction result map is *resm*	*TxnSystemMach*	No
	$\triangledown\ \langle t, c \rangle$	Transaction t's result is fixed to c	*GroupMach*(g)	Yes
Prepare proposal map	$\langle g, t \rangle \hookrightarrow \text{▰}\ \mathit{ppm}$	Transaction t's prepare proposal map on group g is *ppm*	*GroupMach*(g)	No
	$\langle g, t \rangle \hookrightarrow \text{▱}\ \langle r, p \rangle$	Transaction t's prepare proposal at rank r is fixed to p	*ReplicaMach*(g, s)	Yes
Value history	$k \hookrightarrow \blacktriangleleft\ \mathit{hist}$	Key k's value history is *hist*	*KeyMach*(k)	No
	$k \hookrightarrow \blacktriangleleft\ \mathit{hist}$	Key k's value history is *hist*	*GroupMach*(g)	No
	$k \hookrightarrow \triangleleft\ h$	Key k's value history has prefix h	*Network*	Yes
Validation history	$\langle g, s, k \rangle \hookrightarrow \blacktriangleright\ \mathit{vdhist}$	Replica s's validation history for key k is *vdhist*	*ReplicaMach*(g, s)	No
	$\langle g, s, k \rangle \hookrightarrow \blacktriangleright\ \mathit{vdhist}$	Replica s's validation history for key k is *vdhist*	*LockInv*(rp, g, s)	No
	$\langle g, s, k \rangle \hookrightarrow \triangleright\ l$	Replica s's validation history for key k has prefix l	*Network*	Yes
Command pool	$\text{◐}\ \mathit{cpool}$	Group g's command pool is *cpool*	*MultiPaxosMach*	No
	$\text{◑}\ \mathit{cpool}$	Group g's command pool is *cpool*	*GroupMach*(g)	No
Command log	$\text{◧}\ \mathit{log}$	Group g's command log is *log*	*MultiPaxosMach*	No
	$\text{◨}\ \mathit{log}$	Group g's command log is *log*	*GroupMach*(g)	No

Figure 19: Permissions on selected abstract state variables introduced by Tulip. For value-history ownership, g denotes the group handling key k. For command-pool and command-log ownership, *GroupMach*(g) is the caller of the corresponding Multi-Paxos instance. Stable permissions are duplicable and might have multiple owners; the table highlights a selected owner.

constructing two invariants, *InvCommit* and *InvAbort*, over the map.

Figure 20(b) and Figure 20(c) illustrate example states satisfying the two invariants, respectively. *InvCommit* requires each committed transaction t_c to be accompanied by the stable permission $g \hookrightarrow \square\ \langle t_c, \top \rangle$ for every participant group g of t_c. The permission encodes t_c being prepared on g and is the result of applying *ActionPrepare* discussed in §5.2 and shown in Figure 11(a). Likewise, for each aborted transaction t_a, *InvAbort* requires the stable permission $g \hookrightarrow \square\ \langle t_a, \bot \rangle$ for some participant group g of t_a.

The transaction result map grows monotonically: once an entry $\langle t, c \rangle$ is inserted into the map, it must stay there forever. This monotonicity captures the promise that a transaction commit or abort decision must be irreversible. On the proof side, it enables the stable permission $\triangledown\ \langle t, c \rangle$ that can be used by other PSM modules to refer to the transaction result without depending on the entire transaction result map.

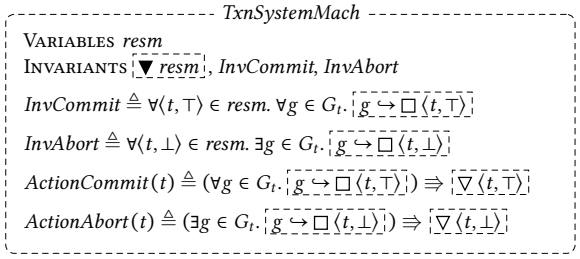
Figure 21 shows the stable permissions required to send the $\langle \textsc{Commit}, t, w_g \rangle$ and $\langle \textsc{Abort}, t \rangle$ messages, respectively. For commit, the transaction coordinator for t must establish $\triangledown\ \langle t, \top \rangle$, which can be obtained by applying *ActionCommit*, defined in Figure 20, if the condition to preserve *InvCommit* is met. The same reasoning principle works for abort as well.

A.2 Coordinator recovery

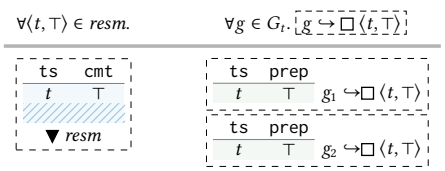
Tulip employs backup 2PC coordinators to clean up leftover locks acquired by transaction clients that fail in the middle of 2PC. Unilaterally aborting these transactions would achieve this goal, but could lead to inconsistent outcomes if the client has already reported the transaction as committed after the prepare phase completed, but *before* the commit command has actually been replicated. Moreover, we must account for the possibility of backup coordinator failures, as well as concurrent coordinators arising from network partitions. Hence, the core challenge is to achieve a stable transaction outcome in the presence of failed and concurrent coordinators.

With the commit requirement discussed in §A.1, we can reduce the problem of achieving a stable transaction outcome to that of achieving a stable prepare decision for each participant group—a problem we formulate as a special case of Fast Paxos [15], as we describe below.

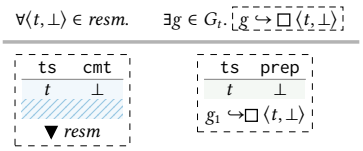
Fast Paxos introduces the notions of *classic rounds* and *fast rounds*. In a classic round, *proposers* send their proposed value to the *leader*. The leader then picks a single value that is "safe"—in the sense that it preserves the value chosen, if any, in a lower-numbered round—and asks *acceptors* to accept it. In a fast round, proposers directly propose (potentially different) values to acceptors, bypassing the leader to cut one network round-trip from the consensus latency. A value is considered *chosen* if it is accepted by a classic quorum



(a) Partial PSM specification for the transaction system.



(b) An example state satisfying *InvCommit*, with $G_t = \{g_1, g_2\}$.



(c) An example state satisfying *InvAbort*, where $g_1 \in G_t$ witnesses the existential condition.

Figure 20: PSM specification and example states capturing Tulip's commit requirement. G_t denotes the set of participant groups of transaction t.

in a classic round, or a fast quorum in a fast round. The definitions of classic and fast quorums are flexible, but must satisfy the quorum requirement [15: §3.1].

Tulip specializes Fast Paxos as follows. The very first round (i.e., rank = 0) is the only fast round; all subsequent rounds are classic. The fast quorum size is $\lceil \frac{3}{4}n \rceil$, and the classic quorum size is $\lceil \frac{n+1}{2} \rceil$, where n is the number of replicas in a replica group. Proposers, acceptors, and leaders roughly map to Tulip's transaction clients, replicas, and backup coordinators. Achieving a stable group prepare decision is then cast as choosing "prepared" ($\top$) or "unprepared" ($\bot$) using a distinct instance of Fast Paxos.

With this framing, we now describe Tulip's coordinator recovery. Tulip detects possible failures of 2PC coordinators with timeouts: if transaction t currently holds locks on its write-set, but the replica has not received a keep-alive message from the latest coordinator (whether the client or a backup coordinator) for some time, the replica spawns a new backup coordinator for t. The new coordinator is assigned rank $r = 2$ if the replica has not seen another backup coordinator for t, or $r = r_t + 1$ otherwise, where r_t is the largest rank for t seen by the replica.

The backup coordinator initiates the recovery procedure by sending $\langle \textsc{Inquire}, t, r \rangle$ to all replicas in each of t's participant groups G (this explains why we include G in the $\langle \textsc{FastPrepare}, t, w_g, G \rangle$ message). Upon receiving the message, each replica first checks whether t has been committed or aborted, or whether it has heard

Message	Stable permission
$\langle \textsc{Commit}, t, w_g \rangle$	$\blacktriangledown\langle t, \top\rangle$
$\langle \textsc{Abort}, t \rangle$	$\blacktriangledown\langle t, \bot\rangle$

Figure 21: Stable permissions associated with commit and abort messages.

from a higher-ranked coordinator; if so, it responds with $\langle \textsc{Committed}, t \rangle$, $\langle \textsc{Aborted}, t \rangle$, or $\langle \textsc{StaleCoord}, t \rangle$, respectively. For $\langle \textsc{Committed}, t \rangle$ or $\langle \textsc{Aborted}, t \rangle$, the backup coordinator simply follows the decision and finalizes the transaction on each participant group; for $\langle \textsc{StaleCoord}, t \rangle$, it immediately terminates.

Otherwise, the replica reports the transaction status using the message $\langle \textsc{TxnStatus}, t, r, r_l, p, b \rangle$, where $\langle r_l, p \rangle$ is the *latest* prepare proposal—made by the highest-ranked coordinator—and boolean b indicates whether t has been validated (i.e., has acquired locks at some point). If the replica has no such information, it sets $\langle r_l, p \rangle = \langle 0, \bot \rangle$ and $b = \bot$. The replica also promises not to accept prepare decisions from coordinators with a lower rank. Based on the TxnStatus responses, the backup coordinator must make prepare decisions consistent with what the client could have made—a process we refer to as *stabilization*, described below.

The backup coordinator waits until a classic quorum of replicas has responded, and collects the latest proposals included in their TxnStatus messages. Let $\langle r_h, p_h \rangle$ be the highest-ranked proposal among them. If $r_h > 0$ (i.e., a classic Fast Paxos round), then the backup coordinator simply follows the prepare decision p_h: it sends $\langle \textsc{Prepare}, t, r \rangle$ if $p_h = \top$, or $\langle \textsc{Unprepare}, t, r \rangle$ otherwise. If $r_h = 0$ (i.e., the only fast round), this entails that the prepare decision could not have been chosen at any positive rank below r. Hence, the backup coordinator only needs to determine the decision that may have been chosen at rank 0.

The possibly chosen decision can be determined by simply counting the prepare decisions accepted by the replicas. For instance, in a five-replica setup, if a backup coordinator receives two $\bot$ responses and one $\top$ response, then it can safely propose $\bot$. The reason is that $\top$ has not reached a majority in the responding classic quorum and therefore could not have been chosen. We provide detailed reasoning in the correctness proof at the end of this section.

After stabilization, the coordinator follows the commit requirement to commit or abort the transaction across all participant groups, thereby releasing the leftover locks.

Verification using PSM. The main correctness criterion of coordinator recovery is to ensure *prepare stability*. This means that all coordinators of a given transaction, including the client and any backup coordinators, must make a consistent decision to prepare or unprepare the transaction on a certain replica group, even in the face of failures. As discussed earlier, Tulip achieves prepare stability using a special case of Fast Paxos. Consequently, the PSM speci-

GroupMach(g)

VARIABLES *ppm*

INVARIANTS $\boxed{\langle g, t\rangle \hookrightarrow \blacksquare\ ppm}$, *InvProposal*

$SafeProposal(r, p) \triangleq \exists m.\ EqLatestFast(r, p, m) * \bigstar_{\langle s,d\rangle \in m} \boxed{\langle g, s, t\rangle \hookrightarrow \bigcirc\ d}$

$InvProposal \triangleq \forall \langle r, p\rangle \in ppm.\ SafeProposal(r, p)$

$ActionPropose(r, p) \triangleq SafeProposal(r, p) \Rrightarrow \boxed{\langle g, t\rangle \hookrightarrow \square\ \langle r, p\rangle}$

ReplicaMach(g, s)

VARIABLES *decs*

INVARIANTS $\boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ decs}$, *InvDecision*

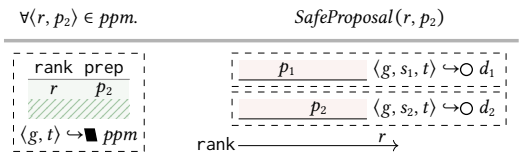
$InvDecision \triangleq$
$\forall r, p.\ r > 0 \implies decs[r] = \texttt{Some}(p) \mathrel{-\!\!*} \boxed{\langle g, t\rangle \hookrightarrow \square\ \langle r, p\rangle}$

$ActionAcceptFast(p) \triangleq \boxed{\langle g, s, t\rangle \hookrightarrow \bullet\ []} \Rrightarrow \boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ [p]}$

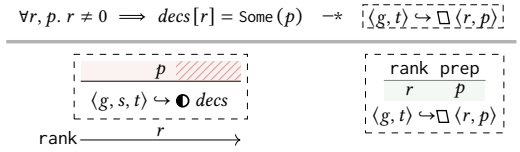
$ActionAccept(r, p, d) \triangleq$
$len(d) \le r \wedge \boxed{\langle g, t\rangle \hookrightarrow \square\ \langle r, p\rangle} * \boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ d} \Rrightarrow$
$\boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ extend(d, r, p)}$

$ActionInquire(r, d) \triangleq$
$len(d) \le r \wedge \boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ d} \Rrightarrow \boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ fill(d, r)}$

(a) Partial PSM specifications for replica groups and replicas. In both specifications, t denotes a transaction whose 2PC process has begun at the corresponding group or replica. For each t, the group module maintains a prepare proposal map (*ppm*), while each replica module maintains ranked prepare decisions (*decs*). $EqLatestFast(r, p, m)$ requires p to match the decision accepted at the largest positive rank below r among m, if any. Otherwise, if a decision is accepted at rank 0 by more than half of the quorum, p must match that decision; if neither condition holds, p may be arbitrary. The function $fill(d, r)$ fills d with None until its length reaches r.



(b) An example state satisfying *InvProposal*.



(c) An example state satisfying *InvDecision*.

Figure 22: PSM specifications and example states capturing stability of prepare decisions.

fications related to prepare stability resemble those of the single-decree Paxos example shown in Figure 5.

Figure 22(a) shows the PSM specifications. The first abstract state variable, the *prepare proposal map*, is owned by the group module and represented by $\boxed{\langle g, t\rangle \hookrightarrow \blacksquare\ ppm}$. The map associates each coordinator rank with the prepare decision made by the coordinator for transaction t.

The second abstract state variable, the *ranked prepare decisions* (already introduced in §5.2), has a similar structure to the voted list of a Paxos acceptor: for each group g, replica s, and transaction t, $\boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ decs}$ records a list *decs* of optional prepare decisions indexed by coordinator ranks. At index r, None means no decision has been accepted at rank r, and $\texttt{Some}(p)$ means p has been accepted at rank r. One half of its ownership, $\boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ decs}$, is assigned to the replica module and the other half, $\boxed{\langle g, s, t\rangle \hookrightarrow \CIRCLE\ decs}$, to the replica lock invariant.

Message	**Stable permission**
⟨FastPrepare, t, w_g, G⟩	None
⟨FastResult, t, p, b⟩	$\exists d.\ \boxed{\langle g, s, t\rangle \hookrightarrow \bigcirc\ d} \wedge d[0] = p$ $\boxed{\langle g, s\rangle \hookrightarrow \diamond\ t}$ if b = true
⟨Prepare, t, r⟩	$\boxed{\langle g, t\rangle \hookrightarrow \square\ \langle r, \top\rangle}$
⟨Unprepare, t, r⟩	$\boxed{\langle g, t\rangle \hookrightarrow \square\ \langle r, \bot\rangle}$
⟨Prepared, t, r⟩	$\exists d.\ \boxed{\langle g, s, t\rangle \hookrightarrow \bigcirc\ d} \wedge d[r] = \top$
⟨Unprepared, t, r⟩	$\exists d.\ \boxed{\langle g, s, t\rangle \hookrightarrow \bigcirc\ d} \wedge d[r] = \bot$
⟨Inquire, t, r⟩	None
⟨TxnStatus, t, r, r_l, p, b⟩	$\exists d.\ \boxed{\langle g, s, t\rangle \hookrightarrow \bigcirc\ d} \wedge len(d) \ge r \wedge d[r_l] = p$ $\boxed{\langle g, s\rangle \hookrightarrow \diamond\ t}$ if b = true

Figure 23: Stable permissions associated with prepare-related messages. When present, g identifies the relevant replica group, and s identifies the replica sending the response.

The group invariant *InvProposal*, akin to *Inv2a* in Figure 5, requires the proposed decision to match the decision accepted at the largest classic rank $r > 0$, if any, among some classic quorum of replicas. Otherwise, if a decision has been accepted at the fast rank $r = 0$ by more than half of the quorum, the proposed decision must match that decision. If neither condition holds, any decision may be proposed. For instance, Figure 22(b) illustrates an example state satisfying the invariant: the decision proposed at rank r must be p_2 because p_2 was accepted at the largest rank before r among the classic quorum $\{s_1, s_2\}$.

The replica invariant *InvDecision* has almost the same structure as *Inv2b* in Figure 5, except for the additional premise $r > 0$. The invariant asserts that any prepare decision p accepted at a classic rank $r > 0$ must have been proposed first. At the end of this section, we will prove prepare stability using these invariants.

The four actions defined in Figure 22(a), one in the group module and three in the replica module, establish the stable permissions associated with prepare-related messages, as summarized in Figure 23. We next explain how these actions model 2PC prepare and coordinator recovery.

The first replica action $ActionAcceptFast(p)$ models how a replica responds to a ⟨FastPrepare, t, w_g, G⟩ request issued by the client of transaction t. This action requires that the ranked-decision list does not yet exist, implying that the replica has not heard from any backup coordinator of transaction t. In this case, the replica program can allocate full ownership of an empty ranked-decision list, $\boxed{\langle g, s, t\rangle \hookrightarrow \bullet\ []}$, and apply $ActionAcceptFast(p)$ where $p = \top$ if the replica successfully validates the transaction's

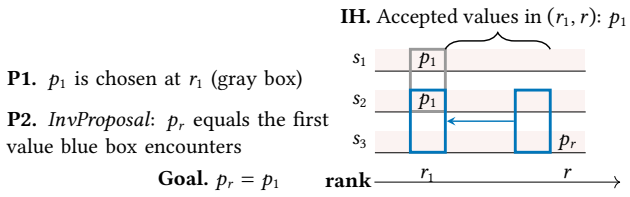


Figure 24: Illustration of the inductive argument for prepare stability.

partial write-set w_g, or otherwise $p = \bot$. Applying this action allows the replica to obtain the stable permission required to reply with $\langle \textsc{FastResult}, t, p, b \rangle$.

The group action *ActionPropose*(r, p) is invoked on the transaction client's slow path (in which case $r = 1$) or by a backup coordinator ($r > 1$). The action proposes the prepare decision p at rank r and produces the stable permission $\boxed{\langle g, t \rangle \hookrightarrow \square \langle r, p \rangle}$ required to send $\langle \textsc{Prepare}, t, r \rangle$ when $p = \top$ or $\langle \textsc{Unprepare}, t, r \rangle$ when $p = \bot$. For transaction clients, the *SafeProposal*(r, p) required by the action can be obtained from the stable permissions carried by $\langle \textsc{FastResult}, t, p, b \rangle$ responses; backup coordinators obtain it from those carried by $\langle \textsc{TxnStatus}, t, r, r_l, p, b \rangle$ responses, as discussed later.

The second replica action *ActionAccept*(r, p, d) captures a replica's response to a $\langle \textsc{Prepare}, t, r \rangle$ or $\langle \textsc{Unprepare}, t, r \rangle$ request, which is issued by either the client or a backup coordinator. This action conditions on two facts: (1) the list has length at most r, and (2) the request carries $\boxed{\langle g, t \rangle \hookrightarrow \square \langle r, p \rangle}$, produced by the group action described above. Intuitively, the first condition indicates that the replica has not received any message from a backup coordinator of transaction t with rank greater than r. The stable permission $\boxed{\langle g, t \rangle \hookrightarrow \square \langle r, p \rangle}$ allows us to re-establish *InvDecision* after accepting p at rank r. This action fills the list with None through index $r - 1$ and then appends p at index r. Applying this action allows the replica to obtain the stable permission required to respond with $\langle \textsc{Prepared}, t, r \rangle$ or $\langle \textsc{Unprepared}, t, r \rangle$.

The last replica action *ActionInquire*(r, d) is used when a replica receives an $\langle \textsc{Inquire}, t, r \rangle$ request from a backup coordinator of transaction t with rank r. Again, it conditions on the replica's ranked prepare decisions having length at most r (i.e., it has not received any message from a coordinator with rank above r). This action fills the ranked-decision list with None through index $r - 1$. Applying it allows the replica to obtain the stable permission required to respond with $\langle \textsc{TxnStatus}, t, r, r_l, p, b \rangle$, where p is the accepted decision at the largest rank r_l. If no decision has been accepted, the replica sets $r_l = 0$ and $p = \bot$. The boolean b indicates whether transaction t has been validated.

We conclude our discussion of coordinator recovery by proving prepare stability using the invariants *InvProposal* and *InvDecision* from Figure 22(a), together with *InvPrepared* from Figure 11(a).

Proof sketch. Our goal is to show that if $\langle t, \top \rangle$ appears in the prepare map, then $\top$ is the only prepare decision that can ever be chosen for transaction t (recall that a value is chosen at rank r if it is accepted by a fast quorum when $r = 0$, or by a classic quorum when $r > 0$). By *InvPrepared* we know that $\top$ has been chosen at some rank. Thus, it suffices to show that any two chosen values agree. Let p_1 and p_2 be the values chosen at r_1 and r_2, respectively, and, without loss of generality, assume $r_1 \le r_2$. We show $p_1 = p_2$.

Case: $r_1 = r_2$. This follows from the fact that any two quorums (fast or classic) have a non-empty intersection.

Case: $0 < r_1 < r_2$ (both are classic). It suffices to show that any value p_r accepted at rank $r > r_1$ equals p_1. By (strong) induction on r and the invariant *InvDecision*, we assume any value accepted in $[r_1, r)$ is p_1. The invariant *InvProposal* asserts that $p_r = p_l$, where p_l is the value accepted at r_l, the highest rank less than r with an accepted value among some quorum. We illustrate in Figure 24 the process of finding r_l as "pushing the blue box backward" from $r - 1$ until it encounters an accepted value. By the quorum intersection rule, we have $r_1 \le r_l < r$, and therefore $p_r = p_1$.

Case: $r_1 = 0$ and $r_2 > 0$ (r_1 fast and r_2 classic). The argument is identical to the previous case, except when $r_l = 0$, which we discuss below. Assume, for contradiction, $p_1 \neq p_2$. By the invariant *InvProposal*, the negation of p_2—namely, p_1—does not reach a majority in some classic quorum. This contradicts the assumption that p_1 is chosen, and a key property of classic and fast quorums: if a value is chosen by a fast quorum, then it must have been accepted by a majority within *any* classic quorum. □

A.3 Transactional read with MVCC

Tulip uses explicit timestamps both to order transactions and to version reads and writes, implementing a form of multi-version concurrency control (MVCC) [4, 22, 29]. Instead of modifying values in-place, replicas handle a write request by appending a new version for the key being modified, consisting of the transaction timestamp and the new value.

Under MVCC, reading a key at timestamp t should return the value last written by the same transaction, if such a write exists, or else the value written by the transaction t' that immediately precedes t; if no such write exists, the read returns absence. As with prepares, reads have two execution paths: *fast-read* and *quorum-read*.

Fast-read. To read key k, the client for transaction t first checks its local write-set and immediately returns the value if one exists. Otherwise, the client sends a $\langle \textsc{Read}, t, k \rangle$ request to all replicas in the group hosting k.

On receiving the request, each replica inspects its local multi-version database for the versions written to k. If it finds a version with timestamp $t' > t$, then it responds to the client with $\langle \textsc{FastRead}, t, k, v \rangle$, where v is the value of the version whose timestamp immediately precedes t, or absence if no such version exists. Once it receives such a

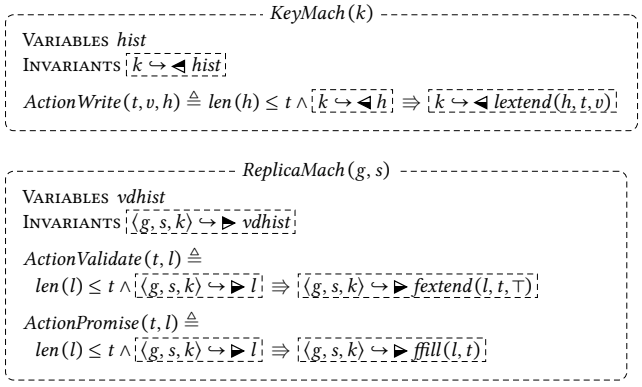


Figure 25: Partial PSM specifications capturing read correctness. The function $lextend(h, t, v)$ first fills value history h through index $t-1$ using its last value (or the "tombstone" record if the key has not been written before), and then appends v. The function $fextend(l, t, b)$ fills list l through index $t-1$ using $\perp$, and then appends b. The function $ffill(l, t)$ performs only the filling step of *fextend*.

response, the client immediately concludes the read with result v, without waiting for the remaining replicas.

Quorum-read. If a replica determines that all its local versions of k have timestamps below t, it responds with $\langle$QuorumRead, $t, t_r, k, v_r\rangle$, where $\langle t_r, v_r\rangle$ is the most recent version available, or $t_r = 0$ if none exists. Additionally, the replica creates an inconsistent log entry to record that it will not validate any transactions with timestamps in the interval (t_r, t). A subtlety arises if there exists a transaction $t_p \in (t_r, t)$ holding k's lock: the replica cannot yet rule out that transaction t_p will commit. In this case, the replica must wait until t_p has released the lock (by committing or aborting) before replying to the client.

Once the client collects a classic quorum of QuorumRead responses, it picks the value v_l with the highest timestamp among the responses as the final read result.

Verification using PSM. To capture the correctness of transaction reads, we introduce a new per-key state variable, *value history*, which serves as an abstraction of the MVCC database state [4]. Each value history is a list of values indexed by timestamps.

Figure 25 shows the per-key PSM module $KeyMach(k)$, including the half-ownership $k \hookrightarrow \blacktriangleleft hist$ of key k's value history (the other half, not shown in the figure, is owned by the group that handles key k to maintain consistency between the value history and the commit records in the Paxos log). We further define an action $ActionWrite(t, v, h)$ to model transaction t writing to the key with value v: it fills the value history h with its most recent value (or "tombstone" records if none exists) up to index $t-1$, then appends v at index t.

As discussed earlier, the fast-read path of a key is enabled for transaction t if there exists a transaction $t' > t$ that has already committed its update to the key. Using the value-history abstraction, this condition simply translates to the

Message	Stable permission
⟨Read, t, k⟩	None
⟨FastRead, t, k, v⟩	$\exists h.\ k \hookrightarrow \triangleleft h \wedge h[t-1] = v$
⟨QuorumRead, t, t_r, k, v_r⟩	$\exists h.\ k \hookrightarrow \triangleleft h \wedge h[t_r] = v_r$ $\exists l.\ \langle g, s, k\rangle \hookrightarrow \triangleright l \wedge \forall t' \in (t_r, t).\ l[t'] = \perp$
(permissions depicted)	$v_r \quad \perp \quad \perp$ $t_r \qquad\qquad t$

Figure 26: Stable permissions associated with read messages. When present, g identifies the relevant replica group, and s identifies the replica sending the response. The final row illustrates the two history prefixes carried by a QuorumRead response.

value history containing some value at index $t-1$. Moreover, since value histories also grow monotonically, we can extract the stable permission $k \hookrightarrow \triangleleft h$ as a witness that values of the key at certain timestamps are fixed. As shown in Figure 26, the ⟨FastRead, t, k, v⟩ message uses this stable permission to encode its promise.

For quorum reads, we introduce another state variable to formalize the replica's promise not to validate transactions whose timestamps fall within a certain range. For each replica-key pair, we add another monotonic list called the *validation history*. A validation history is a list of validation decisions indexed by timestamps. A value of "validated" ($\top$) at index t in the list for key k indicates that transaction t, whose write-set includes k, has been validated on the replica; a value of "unvalidated" ($\perp$), on the other hand, means that t has not been—and will not be—validated on the replica.

As illustrated in Figure 25, the half-ownership $\langle g, s, k\rangle \hookrightarrow \blacktriangleright vdhist$ of the validation history belongs to the $ReplicaMach(g, s)$ module. The other half is owned by the lock invariant $LockInv(\mathsf{rp}, g, s)$ of the replica program to connect the abstract state with the implementation state. We also define actions to update validation histories in the following two cases. First, upon successfully validating transaction t on the replica, $ActionValidate(t, l)$ is applied for each key in t's write-set: it extends the validation history l with $\perp$ up to $t-1$, and appends $\top$ at index t. Second, when processing a ⟨Read, t, k⟩ message in the slow path, $ActionPromise(t, l)$ extends k's validation history l on the replica with $\perp$ up to $t-1$.

The extension made by $ActionPromise(t, l)$ allows a replica to establish the stable permissions required to send a ⟨QuorumRead, t, t_r, k, v_r⟩ message, as shown in Figure 26. The stable permissions have two parts: $k \hookrightarrow \triangleleft h$ asserts that the value history contains value v_r at timestamp t_r, while $\langle g, s, k\rangle \hookrightarrow \triangleright l$, the stable permission for validation histories, serves as the promise that no transactions in the timestamp range (t_r, t) will be validated by the replica. These stable permissions are at the heart of the correctness argument for quorum reads, as demonstrated below.

The key to proving the correctness of quorum reads at timestamp t is to show that t_l—the largest timestamp among the collected QuorumRead responses—immediately pre-

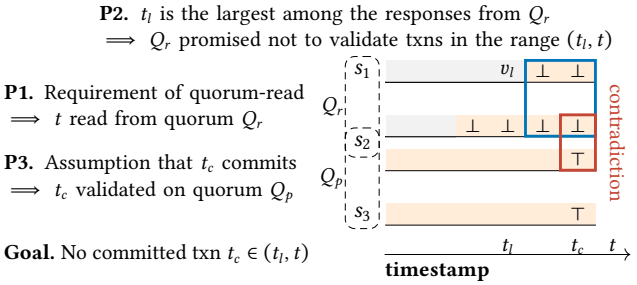


Figure 27: Illustration of the read-consistency proof for a quorum read.

cedes t. This amounts to showing that no transaction in the range (t_l, t) could have committed.

Proof sketch. Assume, for contradiction, the existence of a committed transaction $t_c \in (t_l, t)$. Recall that the stable permission of $\langle \text{QUORUMREAD}, t, t_r, k, v_r \rangle$, as shown in Figure 26, asserts that the value history contains v_r at timestamp t_r, and that no transactions in the range (t_r, t) can be validated on the replica that replies with the message (i.e., the validation history for k contains $\bot$ in that range).

Figure 27 illustrates an example in which quorum-read responses are collected from a quorum Q_r, with $s_1 \in Q_r$ responding with the latest version $\langle t_l, v_l \rangle$. From the fact that t_l is the largest timestamp among the responses, we know that replicas in Q_r must have all promised not to validate transactions within the range (t_l, t), as depicted by the blue box in Figure 27.

Moreover, a committed transaction must be validated on some classic quorum. From the assumption that t_c is committed, we obtain a quorum Q_p on which t_c is validated.

Using the majority rule, we can exhibit a replica (e.g., $s_2 \in Q_r \cap Q_p$ in Figure 27) that, contradictorily, promised not to validate t_c but also validated t_c. □

A.4 Multi-Paxos library

Intuitively, the primary abstract state exposed by the Multi-Paxos interface is an append-only log. The API provided by Tulip's internal Multi-Paxos library, however, does not have a direct `Append(v)` operation. Instead, the API provides two functions: `Submit(v)`, which allows the caller to request that v be added to the log, and `Lookup(i)`, which allows the caller to learn what value appears at position i in the log. This API design aligns more closely with typical usage of the library, where users submit a command in one thread and apply replicated commands in another.

Verification using PSM. As shown in Figure 28, we specify Multi-Paxos with two new abstract state variables: the *command pool*, which reflects the set of commands that callers have submitted, and the *command log*, which represents the set of commands that have been agreed upon in a Multi-Paxos group. We represent both as fractional permissions: one half of the ownership is maintained internally by the

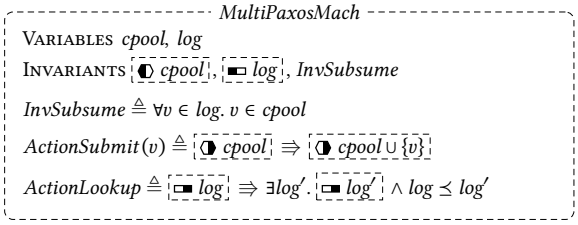


Figure 28: Partial PSM specification for one Multi-Paxos instance.

Multi-Paxos module, and the other half by the caller module—the group module in Tulip's case.

Further, we define two actions to update the abstract state: *ActionSubmit*(v) adds the command v to the command pool and is used when callers invoke `Submit(v)`. *ActionLookup* is used when callers invoke `Lookup(i)`. The action, counter-intuitively, mutates the command log; moreover, the caller cannot control the new value—it only knows that *some* commands have been appended to the log. This specification design offers the Multi-Paxos library implementation great flexibility in deciding what commands end up appearing in the log, allowing it to account for duplicate, missing, and out-of-order commands due to network and server failures.

However, one important caveat remains: library users often need to associate arbitrary invariants with the submitted commands, but the gap between the command pool and the log means that the caller cannot *use* the invariants by the time a command is read via `Lookup(i)`. We resolve this gap by enforcing the invariant *InvSubsume*, which requires that every command in the log also be present in the command pool. Put differently, only commands submitted by the users can be added to the agreed-upon log. Importantly, this invariant enables callers to *transfer* arbitrary invariants from the command pool to the command log.

As an example usage of the Multi-Paxos library, consider Tulip's 2PC commit, as described in §A.1. Recall that when a transaction client is ready to commit a transaction with timestamp t with partial write-set w_g, it sends a $\langle \text{COMMIT}, t, w_g \rangle$ request to the Multi-Paxos leader of group g. On receiving the request, the leader calls `Submit` to initiate replication of a new commit command.

To preclude the bug in which a leader replicates a commit command for a transaction when it should not, we define a user-level invariant that requires every commit command in the command pool to be associated with $\boxed{\nabla \langle t, \top \rangle}$, the same stable permission that came with the $\langle \text{COMMIT}, t, w_g \rangle$ request. Once the commit command is agreed upon, a replica might retrieve it from the log via `Lookup` and apply it. To prove the correctness of applying the commit command, the replica needs the stable permission $\boxed{\nabla \langle t, \top \rangle}$ asserting that the transaction is indeed committed. Even though the user-level invariant ties the stable permission to commands in the command pool, rather than the log, the library invariant *InvSubsume* allows the proof to conclude that every user-level invariant must also hold for the commands in the log.