Grove: a Separation-Logic Library for Verifying Distributed Systems

Upamanyu Sharma Ralf Jung[†] Joseph Tassarotti[▽] M. Frans Kaashoek Nickolai Zeldovich
MIT CSAIL [†] *ETH Zurich* [▽] *New York University*

Abstract

Grove is a concurrent separation logic library for verifying distributed systems. Grove is the first to handle time-based leases, including their interaction with reconfiguration, crash recovery, thread-level concurrency, and unreliable networks. This paper uses Grove to verify several distributed system components written in Go, including vKV, a realistic distributed multi-threaded key-value store. vKV supports reconfiguration, primary/backup replication, and crash recovery, and uses leases to execute read-only requests on any replica. vKV achieves high performance (67-73% of Redis on a single core), scales with more cores and more backup replicas (achieving about 2× the throughput when going from 1 to 3 servers), and can safely execute reads while reconfiguring.

1 Introduction

Large-scale applications run on many servers, and face a wide range of challenges typical of distributed systems such as concurrency, crashes, network outages, loosely-coupled clocks between servers, etc. This means that the developer has to consider a large number of subtle corner cases and interactions, which in turn makes it difficult to ensure that the application correctly handles all such cases. Formal verification is an attractive approach to rigorously establish correctness of such systems, and in principle could help developers ensure that they correctly handle all of the corner cases.

One particularly challenging and cross-cutting aspect of distributed systems, which has not been addressed in prior work on verification, is the use of leases. Leases [16] are a widely used technique in distributed systems. A lease is a promise that some aspect of the system will not change for some duration of time (e.g., the primary server will not be replaced for the next 5 seconds). For instance, leases are used to ensure there is at most one Paxos leader trying to run the replication protocol in Spanner [10]; and GFS [13], Chubby [3], and DynamoDB [12] have similar mechanisms. Leases allow a leader to execute read-only operations quickly, without having to contact replicas to confirm that it is still the leader. Leases are challenging to use correctly because they interact with crash recovery and reconfiguration (e.g., reconfiguration must wait until leases expire before it can choose a new primary server) and node-local concurrency (e.g., executing a read-only operation may require first checking that a lease is valid, but in the time between the check and the operation itself, the lease may have expired, and other threads may have executed additional writes).

This paper presents Grove, a library based on concurrent separation logic (CSL) [34] for reasoning about distributed systems, where state is split between nodes, crashes discard a node's memory state, network messages can be lost or duplicated, and nodes have loosely synchronized clocks. In CSL, a proof decomposes a system's state into parts called resources that are logically owned by different threads. Synchronization primitives, like mutexes, are used to transfer ownership between threads. Grove generalizes this notion of resources and ownership to reason about distributed systems; in particular, Grove introduces *time-bounded invariants* to reason about leases, extends Crash Hoare Logic [5, 8] to reason about crashes in distributed systems, provides abstractions for reasoning about append-only logs and monotonic epoch counters, and provides a verified RPC library. This makes Grove the first to support verification of distributed systems that use leases, including their interaction with crash recovery, reconfiguration, concurrency, and unreliable networks.

To demonstrate Grove's approach, we developed a number of distributed system components written in Go (libraries, systems, and applications), and specified and verified them using Grove. As we explain in §3, these components make extensive use of Grove's ownership and resources. To name some examples: the proof of consistency for a replicated log in a primary-backup replication library uses ownership of logical append-only lists; the proof of crash recovery in a durable storage library uses ownership of durable files; proofs about RPCs that may re-execute many times use duplicable ownership (which can be thought of as knowledge), since they cannot transfer ownership of unique resources; and proofs about read operations that use leases to avoid coordination in a state-machine replication library uses time-bounded invariants to prove the state being read is not stale.

By using CSL, Grove enables modular reasoning: developers can verify each component of a distributed system separately, and reason about code line-by-line, rather than explicitly considering all possible interleavings. Nevertheless, these proofs still compose into a complete proof of the entire distributed system. For instance, our case study builds a repli-

SOSP '23, October 23–26, 2023, Koblenz, Germany
ACM ISBN 979-8-4007-0229-7/23/10.
http://dx.doi.org/10.1145/3600006.3613172

Component	Code	Spec and proof	Description
`bank`	[38: §2.3.4]	[38: §4.7]	Uses vKV and lock service to execute bank transactions
`lockservice`	[38: §2.3.3]	[38: §4.7]	Distributed lock service implemented on top of vKV
`cachekv`	§2.3.2	§4.4	Uses leases for linearizable key-value caching on client
vKV	§2.3.1	§5.1	Handles key-value state and operations (`Get`, `Put`, `CondPut`)
`exactlyonce`	[38: §2.2.8]	[38: §3.6], [38: §3.7]	Tracks and handles duplicate operations for exactly-once semantics
`clerk`	[38: §2.2.7]	§5.1	Issues operations to replicated state machine
`storage`	[38: §2.2.4]	[38: §3.8]	Stores application state and handles state transfers
`configservice`	§2.2.2	[38: §4.6], [38: §5.3]	Tracks the changing set of replica servers and issues epoch leases
`paxos`	[38: §2.2.5]	[38: §4.6]	Simple Paxos-based fault-tolerant replication for `configservice`
`reconfig`	§2.2.2	§4.2	Adds or removes replicas, using the config service
`replica`	§2.2.1, §2.2.3	§3.2.2, §3.2.3, §3.3, §4.1, §4.2, §4.3, [38: §4.5]	Stores and replicates operations between primary and backups
Lease abstraction	—	§3.4, §3.5	Time-bounded invariant abstraction to reason about leases
`rpc`	§2.1	§3.3	Unreliable request/response communication

Figure 1: Components verified using Grove as case studies. Some components are presented only in the extended version of this paper [38].

cated key-value service (called vKV) out of multiple independent components (RPC library, primary-backup replication, state-machine replication, durable storage, configuration service, etc), and builds an example bank application on top of vKV and a distributed lock service. The proof of the bank considers only the specifications of the underlying vKV and lock service, and does not look at their implementation. At the same time, the composed proofs ensure there are no subtle bugs due to surprising interactions between the components. As we show in §6, the proof-to-code ratio is about 12×, on par with other distributed systems verification efforts, which shows that handling leases, reconfiguration, concurrency, etc, with Grove does not come at the cost of inflated proof effort.

Grove's support for leases, concurrency, reconfiguration, and crash recovery is crucial for verifying high-performance distributed systems. §7 shows that vKV achieves good performance (67–73% of the throughput of Redis in a single-core unreplicated configuration), scales well with the number of cores and the number of backup replicas (going from 463,491 to 816,252 req/sec for a 95%-read YCSB workload when using 1 and 3 servers respectively), and is able to serve read requests quickly and safely during reconfiguration.

To summarize, the contribution of this paper is Grove, which generalizes concurrent separation logic (CSL) to support distributed systems with RPCs, leases, replication, reconfiguration, and crash recovery. The paper provides lessons, insights, and techniques at several different levels. For a general systems audience, Grove demonstrates that ownership-based reasoning (using CSL) is valuable for distributed systems, by showing what kinds of distributed systems can be verified, how verification catches specific subtle bugs (the "what if" scenarios in §4), and how CSL leads to modular development (§5). For a verification audience, Grove presents techniques and ideas for how to extend CSL to reason about distributed systems issues, such as RPCs, leases, and replication, as we describe in §3. These ideas may be helpful to researchers building frameworks for verifying distributed systems. Finally, the source code of Grove and its case studies is publicly available,[1] for experts that may want to adopt Grove's lower-level techniques for encoding distributed systems in the Iris separation logic [22, 23, 27].

One limitation is that Grove is only able to verify safety properties, ensuring that a system never returns the wrong results. Grove cannot verify liveness properties, such as ensuring that the system will respond or otherwise make progress.

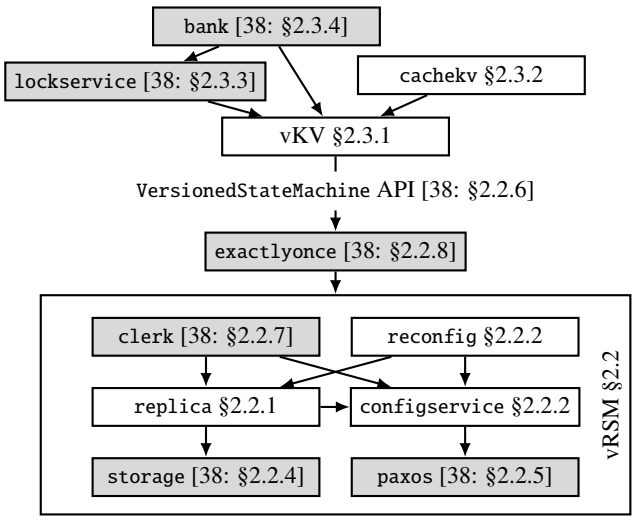

Figure 2: Case study components. An arrow $A \to B$ means A uses B. Gray components are described only in the extended version of this paper [38].

2 Motivating case studies

Grove's goal is to enable verification of distributed system components in a way that allows composing them into a single proof for the entire system. To illustrate the verification challenges that Grove aims to address, this section presents a number of components typically seen in distributed systems, shown in Figure 1, spanning from RPC and storage at the lowest level, to libraries for replicated state machines,

[1] Grove is available at `https://github.com/mit-pdos/perennial` and the case studies are at `https://github.com/mit-pdos/gokv`.

key-value stores, and locking, to application-level code such as a bank example. Distributed systems challenges, such as concurrency, crashes, clocks, etc, show up in many of these components, and a key benefit of Grove is that it provides a consistent framework for handling these issues in the specifications and proofs of each component, which in turn allows combining these components into larger verified systems. The components fit together to build vKV, a replicated key-value store, as well as applications on top of it, as shown in Figure 2.

These components use sophisticated techniques to achieve high performance and strong correctness guarantees. For instance, they use threads on each machine to execute RPCs in parallel; store data durably on disk (using a separate thread for performance) and recover their state after a crash; batch disk writes and pipelines requests for improved performance; achieve linearizability even in the presence of retransmission, crashes, and in-flight client requests while adding or removing servers through reconfiguration; and use leases to coordinate the execution of read-only requests at each replica with reconfiguration.

2.1 RPC library

An important building block for distributed systems is RPC, which allows a client to invoke a procedure on a remote server. For instance, a client invocation of `rpcClient.Call("f", args)` invokes `f(args)` on the server to which `rpcClient` is connected. The `rpc` library provides *unreliable* RPCs, meaning that one invocation by a client can result in the server running the corresponding function one, zero, or many times. This is because the underlying network may drop, reorder, or duplicate packets. Applications typically do not directly invoke RPCs; rather, applications use various *clerks*, which wrap RPCs with additional handling (such as adding request IDs, retrying, etc).

2.2 Replicated state machine library

The focal point of our case study is a replicated state machine library called vRSM, as shown in Figure 3. vRSM replicates a state machine supplied by the application (the exact interface is shown in the extended version of this paper [38: Figure 10]). §2.3 discusses how applications use this interface. vRSM is implemented in several components, which this subsection describes. The components each handle a different aspect of state machine replication, allowing, for instance, durability to be implemented separately from the replication protocol.

2.2.1 `replica` server: replicating writes

The `replica` component manages copies of the state machine being replicated. A `replica` server is either a primary and handles write requests from clients, or else is a backup (we discuss the handling of reads later in §2.2.3). Upon receiving an operation, a primary server applies it locally and then replicates it to all backup servers before replying to the client, as shown in Figure 4. To replicate an operation, the primary spawns threads to send RPCs concurrently to each backup and

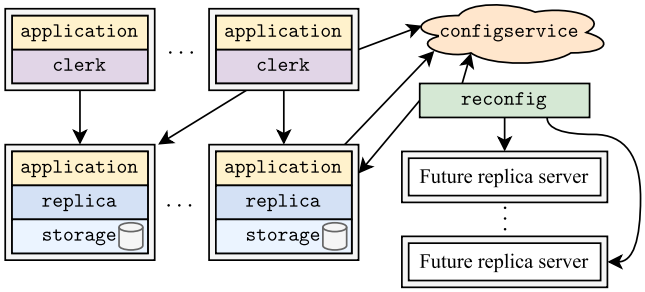

Figure 3: A running vRSM system. Double borders represent machines. Arrows represent RPCs. The cloud represents the replicated configuration service. `reconfig` represents an operator performing reconfiguration.

then waits for all the threads to finish (using a Go `WaitGroup`) to know that the operation is committed (i.e. applied by all `replica` servers). Backup replicas handle these RPCs by also applying the operation locally. `s.stateLogger` takes care of managing the RSM state, as we describe in [38: §2.2.4].

```
1  func (s *PrimaryServer) Apply(op) Result {
2    s.mutex.Lock()
3    nextIndex := s.nextIndex
4    e := s.epoch
5    s.nextIndex += 1
6    res := s.stateLogger.LocalApply(op)
7    s.mutex.Unlock()
8
9    wg := new(WaitGroup)
10   for j := 0; j < len(s.backupClerks); j++ {
11     wg.Add(1)
12     go func(j int) {
13       s.backupClerks[j].ApplyAsBackupRPC(e, nextIndex, op)
14       wg.Done()
15     } (j)
16   }
17
18   wg.Wait()
19   return res
20 }
```

Figure 4: Simplified primary server code, with error handling omitted.

This protocol requires the primary to replicate the operation to *all* servers before replying to a client, so if even a single backup is unavailable, the replication protocol is blocked. To unblock the system, an operator or an automatic failure detector can remove unresponsive servers (and add new ones) by invoking the `reconfig` component, described next.

2.2.2 `reconfig` using `configservice`

The `reconfig` component allows adding or removing `replica` servers by making use of sequentially numbered *epochs* and a `configservice` component. An epoch typically corresponds to a configuration—that is, a set of servers with one designated as the primary. We call such epochs *live*, even if such an epoch has been superseded by another one. However, some epochs may not have a corresponding configuration, if that epoch never started running (e.g. because a node running `reconfig` crashed); we call such epochs *reserved*. The `configservice` keeps track of the latest epoch number and the most recent configuration (a list of server addresses), which may be from an earlier epoch if the current epoch is not live.

Clients can invoke operations concurrently with reconfiguration, which runs the risk of a client's operations being applied in an old configuration after the new configuration has already started, thereby missing these operations in the new config. To prevent this, reconfiguration first *seals* one of the servers from the old epoch. A sealed server no longer modifies its state until it enters a new epoch, at which point it becomes unsealed. Sealed servers may still handle read requests. Sealing allows the reconfiguration process to get a stable checkpoint of the system state and ensure all of the servers in the new configuration have consistent state before entering the new epoch.

```
1  // Reserve a new epoch number for reconfiguration, and
2  // return the current configuration (set of servers).
3  func (ck *Clerk) ReserveEpochAndGetConfig() (uint64, []Address)
4
5  // Return current configuration, used by clients to
6  // determine what servers to talk to.
7  func (ck *Clerk) GetConfig() []Address
8
9  // Set new configuration, making epoch live, as long as no
10 // higher-numbered epoch has been reserved.
11 func (ck *Clerk) TryWriteConfig(epoch uint64,
12                                 config []Address) Error
13
14 // Get a lease for specified epoch, as long as it's the current
15 // epoch, returning the new lease expiration time.
16 func (ck *Clerk) GetLease(epoch uint64) (Error, uint64)
```

Figure 5: Interface provided by the configuration service clerk.

```
1  func Reconfigure(newServers []Address) {
2    newEpoch, oldServers := configClerk.ReserveEpochAndGetConfig()
3
4    // get state from a server from old config
5    oldClerk := MakeClerk(oldServers[Rand() % len(oldServers)])
6    oldState := oldClerk.GetStateAndSeal(newEpoch)
7
8    // make clerks to all of the new servers
9    var newClerks = make([]Clerk, len(newServers))
10   for i := 0; i < len(newServers); i++ {
11     newClerks[i] = MakeClerk(newServers[i])
12   }
13
14   // set state on all the new servers
15   wg := new(WaitGroup)
16   for i := 0; i < len(newClerks); i++ {
17     wg.Add(1)
18     go func(i int) {
19       newClerks[i].SetNewEpochState(newEpoch, oldState)
20       wg.Done()
21     }(i)
22   }
23   wg.Wait()
24
25   // write new addresses to config service
26   err := configClerk.TryWriteConfig(newEpoch, newServers)
27   if err == nil {
28     // activate the new primary server
29     newClerks[0].BecomePrimary(newEpoch)
30   }
31 }
```

Figure 6: Simplified `reconfig` code, with most error handling omitted.

Reconfiguration involves coordination between the configuration service, the old servers, and the new servers. Figure 5 shows the API for the configuration service, and Figure 6 shows the code for reconfiguration, invoked to change to a new set of servers specified by the `newServers` argument. Not shown is the monitoring logic that decides when to call this function or which new servers to choose; correctness (safety) is independent of that logic, and Grove does not prove liveness. Reconfiguration consists of the following steps:

1. Ask the configuration service to atomically create a new epoch and return the new epoch's number as well as the latest previous configuration (line 2).
2. Seal a replica server from the previous configuration and fetch its key-value mappings (line 6).
3. Initialize state on all new servers with the state from the old replica, informing them of the new epoch (line 19).
4. Make the new epoch live at the configuration service, by sending it the new configuration (line 26).
5. Enable the primary in the new configuration, which allows the primary to start processing write requests (line 29).

In case of a network partition, it is possible that both sides of the partition will try to initiate reconfiguration. One might worry that this would lead to two copies of the system with diverging states. This possibility is ruled out with the help of the configuration service, which accepts only the highest-numbered new epoch in its `WriteConfig` RPC handler, together with the replicas' `SetNewEpochState` handler, which rejects state from lower-numbered epochs. As a result, the reconfiguration process that has the higher new epoch from `GetNewEpochAndConfig()` will win.

2.2.3 `replica` server: lease-based reads

Any `replica` server (primary as well as backup) can serve linearizable reads without communicating with other servers by using leases, as shown in Figure 7. Leases avoid the possibility of one server returning stale reads if reconfiguration happens and the new configuration has executed additional writes not seen by this server. Specifically, every replica runs a background thread that contacts the configuration service to obtain or extend a lease that promises the configuration service will not change the current epoch number (and thus not reconfigure) until lease expiration (e.g., 1 second from the time the lease is issued). All servers can serve read requests because this lease is a promise about the epoch number, rather than anything specific to a particular server's state.

When a `replica` server receives a read-only operation, and its lease is still valid, it computes the response from its local state. The `replica`'s local state includes all committed operations since committed operations must be acknowledged by all servers. However, the state may also include ongoing write operations that have not yet been committed. To ensure that the client's observed read does not roll back due to a crash or reconfiguration, the replica waits for all the previous writes that the read depends on to be committed before sending the result to the client. As part of executing the read operation, `LocalRead`'s job is to determine which prior requests the read depends on, returning the appropriate `idx` value as shown in Figure 7; it is always safe to return `idx = s.nextIndex`. If

reconfiguration happens during `waitForCommitted`, the server tells the client to retry.

```
1  func (s *Server) ApplyReadonly(op) Result {
2    s.mutex.Lock()
3
4    if s.leaseExpiry > GetTimeRange().latest {
5      e := s.epoch
6      idx, res := s.stateLogger.LocalRead(op)
7      s.mutex.Unlock()
8
9      if s.waitForCommitted(e, op, idx) {
10       return res
11     } else {
12       return ErrRetry
13     }
14   } else {
15     s.mutex.Unlock()
16     return ErrRetry
17   }
18 }
```

Figure 7: Simplified code for handling read-only operations.

Since the clocks on different nodes might be slightly out of sync with each other, Grove provides a TrueTime-like API [10] for accessing the current time, `GetTimeRange()`. This function returns a pair of timestamps, `earliest` and `latest`, which provide lower and upper bounds for the current time.

2.3 Applications on top of vRSM

2.3.1 vKV

vKV is implemented on top of vRSM and the `exactlyonce` library. The server-side part of vKV is an implementation of the state machine interface expected by vRSM. The client-side part of vKV is a clerk implemented on top of the `exactlyonce` clerk, with the API shown in Figure 8. By building on top vRSM, the implementation of vKV itself is simple: it consists of (de)serialization methods to turn key-value operations into byte slices and a few functions to read and update an in-memory map. In addition to storing a map of keys to values, vKV also stores a map from keys to the index of the last operation that modified that key, which allows vKV to take advantage of vRSM's versioned state machine interface ([38: §2.2.6]) to improve the performance of reads.

```
1 func (ck *Clerk) Put(key, val string)
2 func (ck *Clerk) CondPut(key, expect, val string)
3 func (ck *Clerk) Get(key string) string
```

Figure 8: Interface provided by the vKV client clerk.

2.3.2 Lease-based client-side caching

As another example of using leases, `cachekv` is a lease-based client-side caching library that works by storing both data and lease expiration times in vKV. Figure 9 shows `GetAndCache` function, which returns the value of the specified `key` and caches it internally for `cachetime` time. It uses `CondPut` to atomically increase the lease duration, which ensures that a concurrent modification did not change the value since the `Get` on line 4. Similarly, `CacheKv`'s `Put` function (not shown) uses `CondPut` to ensure the value is only changed if the lease is expired. Finally, `CacheKv`'s `Get` function first tries reading from `k.cache` and only invokes the `Get` on vKV if the value is not cached. The client-caching library is simple, but exemplifies how leases can be used for cache consistency [16].

```
1  func (k *CacheKv) GetAndCache(key string,
2                                cachetime uint64) string {
3    for {
4      old := k.kv.Get(key)
5      new := old
6
7      newExpiration := max(GetTimeRange().latest+cachetime,
8        old.leaseExpiration)
9      new.leaseExpiration = newExpiration
10
11     // Try to update the lease expiration time on the backend
12     resp := k.kv.CondPut(key, old, new)
13     if resp == "ok" {
14       k.mu.Lock()
15       k.cache[key] = cacheValue{v: old.v, l: newLeaseExpiration}
16       k.mu.Unlock()
17       return old.v
18     }
19   }
20 }
```

Figure 9: Simplified code for getting a lease on a key and caching it.

3 Grove

To formally verify distributed systems such as the case studies described in the previous section, Grove adopts the ideas of concurrent separation logic (CSL) [23, 34]. CSL enables modular specifications and proofs: a developer can take two verified components, each with their own specification, and use both of them in their application without worrying that the combination breaks either component's proof. In the context of distributed systems, this allows Grove developers to separately specify and verify different services that run on different machines but that will eventually be used together (e.g., a configuration service, a key-value store, and a lock service), as well as different libraries that will run on the same machine (e.g., a clerk that talks to vKV, a clerk that talks to the lock service, etc).

In the rest of this section, we first introduce Grove's execution model (§3.1), followed by how Grove generalizes separation logic and resource ownership to distributed systems (§3.2), and Grove's library of reasoning principles. The extended version of the paper provides additional details, such as Grove's handling of exactly-once operations [38: §3.6] and crashes [38: §3.8].

3.1 Execution model

Grove models distributed systems as a collection of nodes, each running a multithreaded program written in Go. Each node has its own memory heap (accessed in Go using loads and stores) as well as durable storage (accessed by reading from and writing to files using `read()` and `write()`).

Crash recovery. Each node has a `main()` function that runs when the node starts up for the first time as well as when the node restarts after a crash. When a node crashes, it loses the

contents of its memory heap and restarts with an empty heap, but retains its durable state.

Nodes crash independently of one another. A few nodes might crash while others keep running, or all of the nodes might crash at the same time. Crashes can happen at any point, including when a node is still recovering from an earlier crash. For instance, a node's `main()` function might have a recovery phase during which it loads durable state into memory or communicates with other nodes to restore its state; crashes can occur even during this phase.

Unreliable network. Nodes communicate over an unreliable network. The low-level network API has a notion of a `Connection`, resembling a connected UDP socket. The API provides two functions, `conn.Send(msg)` and `conn.Receive()` that respectively send and receive messages over that connection. Grove models the network as unreliable: `conn.Send(msg)` is not guaranteed to deliver messages in order, and messages may be dropped or duplicated.

Clocks. There is a global clock, which advances monotonically and represents a notion of wall-clock time. Every node exposes a TrueTime-like API [10], `GetTimeRange()`, which returns a pair of timestamps that represent an interval (lower and upper bounds) that, according to our model, must contain the global clock value. This assumes that node clocks are synchronized to within known bounds (on the order of less than a second, for the purposes of vKV's use of leases).

3.2 Separation logic for distributed systems

Grove generalizes concurrent separation logic (CSL) [23, 34] to reason about distributed systems. CSL uses Hoare logic-style specifications for pieces of code (e.g., functions) of the form $\{P\}$ `f()` $\{Q\}$ meaning the precondition for running `f()` is the assertion P and the postcondition is Q. To prove such a spec, a developer applies proof rules to reason line-by-line about `f()`, starting with a state matching P, and showing that the final state matches Q.

This section reviews the background on CSL and introduces key abstractions that Grove provides on top of CSL, along with how they are used in vKV's proof.

3.2.1 Ownership reasoning

In separation logic, assertions not only describe what is true about a system's state, but also what parts of the state are logically *owned* by the thread executing the code at that point. For example, the assertion $x \mapsto v$ (pronounced "x points to v") says that memory location x stores a value v *and* that the thread running that function owns the location x, in the sense that, as long as this thread continues to own this assertion, no other thread can access location x. Such ownership constraints form the basis for modular reasoning: for instance, the fact that no other thread can access location x allows a developer to reason about this function without considering other concurrently executing code.

Grove's library brings this ownership-based modular reasoning to distributed systems. Grove provides per-node heap points-to resources: $x \mapsto_j v$ denotes ownership of location x with value v on node j's heap. Grove also provides resources for network state and file contents, as we discuss later. In a distributed systems setting, this enables the developer to verify the code running on one node without worrying about what code might be running on other nodes at the same time.

Separation logic additionally introduces a new logical connective, $*$, called *separating conjunction*. The assertion $P * Q$ holds in a state s if both P and Q are true in s, and furthermore, s can be split into two *disjoint* resources satisfying P and Q, respectively. In conventional CSL, disjointedness means separate subsets of a program's memory heap. Grove uses the separation conjunction to account for separation across different nodes as well.

3.2.2 Ghost resources

In addition to physical resources like the heap, separation logic allows proofs to use *ghost resources*, a modern form of auxiliary variables [21, 25]. Ghost resources talk about the state of the system at a more abstract level. In concurrent separation logic, ghost resources represent ghost state—state that is not materialized by the actual running code, but is useful for specification and proof. Just like physical resources, ghost resources can be owned. While the evolution of physical resources is entirely determined by the code (e.g., based on how the code modifies memory or file contents), ghost resources are controlled by the proof.

Ghost resources are especially useful for reasoning about distributed systems because they can span nodes and allow developers to reason about the system at a higher level of abstraction. Ghost resources are more powerful than regular abstract state, because these resources can be owned, which in turn provides constraints on how different threads can modify the ghost state, and thereby enables modular reasoning.

Epochs. Using ghost resources, Grove provides an epoch abstraction, which is used in the proof of vKV to keep track of and reason about the current configuration. Grove provides two resources for representing epochs. The first, $\mathsf{CurrentEpoch} \mapsto e$, states that the current epoch number is exactly e. This resource is owned by the configuration service: it is the only component that can approve a reconfiguration. The second, $\mathsf{CurrentEpoch} \geq e$, states that the current epoch is at least e. This resource is *duplicable*, meaning that many threads can have it at the same time. In a way, this resource represents *knowledge* of the fact that the epoch is at least e, rather than any exclusive ownership of some part of the state. The fact that $\mathsf{CurrentEpoch} \geq e$ is duplicable implies that epoch numbers are monotonically increasing (i.e., the resource promises that the current epoch number cannot decrease). Many vKV components, including the primary and the backup replicas, make use of this resource to represent

knowledge that a new epoch exists. This means that a server can reject operations from earlier epochs, such as a stale `SetNewEpochState()`.

Logs. Grove provides a log abstraction using ghost resources, encoded as an append-only list. The proof of vKV encodes the main logical state of each replica server using this log abstraction, representing the operations that the node has applied so far. There are three kinds of ghost resources provided by Grove that talk about the state of an append-only log:

The *points-to* resource $a \overset{\text{list}}{\mapsto} \ell$ denotes ownership of an append-only list named a with current value ℓ. The only way to update this points-to resource in the proof is to go from ownership of $a \overset{\text{list}}{\mapsto} \ell$ to $a \overset{\text{list}}{\mapsto} \ell + \ell'$, i.e. to append at the end.

The *lower bound* resource $a \overset{\text{list}}{\sqsupseteq} \ell$ denotes knowledge that the list a has prefix ℓ. This is similar to the lower-bound epoch resource $\mathsf{CurrentEpoch} \geq e$ described above, and just like it, $a \overset{\text{list}}{\sqsupseteq} \ell$ is duplicable. Other parts of the proof cannot possibly violate lower bound resources.

Finally, the *read-only* resource $a \overset{\text{list}}{\mapsto}_{\square} \ell$ denotes knowledge that a has value ℓ and can never be updated. To establish this read-only resource, a proof has to give up ownership of $a \overset{\text{list}}{\mapsto} \ell$ and in exchange get ownership of the read-only resource. After this, no part of the proof can possibly have ownership of $a \overset{\text{list}}{\mapsto} \ell$ so the list can never be updated again.

vKV's proof represents operations accepted by each server as of epoch number e with append-only lists: server $j \in \{0, 1, \ldots, n\}$ owns the resource $\texttt{accepted}_j[e] \overset{\text{list}}{\mapsto} \ell$.[2] Server j can only gain knowledge (not ownership) about other servers' $\texttt{accepted}_k[e]$ list and does this through RPCs (discussed in §3.3). All servers also own heap resources for their in-memory representation of this abstract state, but the proof does not involve sharing these heap resources across nodes. Servers only talk about other servers in terms of ghost resources for their $\texttt{accepted}_j[e]$ list. Read-only resources are used to represent sealed replicas.

The proof also has a global points-to $\texttt{committed} \overset{\text{list}}{\mapsto} \ell$ that represents the committed list of operations. When a primary server commits an operation, the proof updates the `committed` points-to resource. However, when reconfiguration happens, the new primary server will need to do the same. Thus, the `committed` points-to resource cannot be permanently owned by any one node. To share this resource between nodes, the proof uses a separation logic invariant, which we explain next.

3.2.3 Invariants

Concurrent separation logic allows for resources to be shared through *invariants*, which can talk about the resources relevant to only a small part of the system without needing to know about the entire system's state.[3] These invariants maintain ownership of resources that must always be available. The assertion $\boxed{P}$ denotes an invariant that maintains ownership of P. When reasoning about code, proofs can temporarily "open" $\boxed{P}$ to get ownership of P, but are required within one physically atomic step of the code to return ownership of P in order to "close" the invariant. An invariant is created by starting with ownership of P and giving it up to establish $\boxed{P}$. The proposition $\boxed{P}$ asserts *knowledge* of the invariant, as opposed to direct ownership of the resources P; many threads can hold $\boxed{P}$ at the same time.

Multiple invariants that each talk about separate parts of a larger system can be freely combined. As an example, a per-node invariant can describe how resources are shared between the node's threads; this is how invariants are used in traditional single-machine CSL ([23, 34]). At the same time, a separate invariant can connect the logical state of all replicas to ensure that the replicas agree on the log of accepted operations. Finally, yet another invariant can cover the configuration service and how the reconfiguration logic ensures that only one set of servers is active at a time.

Example. In the vKV proof, each node has a local invariant I_{node_j} that maintains ownership of local heap resources and $\texttt{accepted}_j[e] \overset{\text{list}}{\mapsto} \ell$ to help reason about node-local concurrency, using Grove's log ghost resource.

Separately, to reason about how nodes coordinate with each other, the proof has a "replication invariant" I_{rep} defined as

$$\boxed{\exists \ell, \exists e, \texttt{committed} \overset{\text{list}}{\mapsto} \ell * \left(\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} \ell\right) * \cdots * \left(\texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} \ell\right)}$$

Here, $*$ is the "separating conjunction" operator that combines ownership of multiple disjoint resources. The invariant maintains ownership of the committed list of operations and, for every replica server, knowledge that the server has accepted all the committed operations. This invariant encodes the primary/backup protocol: in order for an operation to be committed, all the servers must have accepted it. Not shown is a part of this invariant that says epoch e corresponds to a configuration consisting of servers 0 through n.

When the primary commits an operation `op`, the proof opens the replication invariant to update the `committed` points-to from ℓ to $\ell + [\texttt{op}]$. In order to close the invariant after the update, the primary needs knowledge of the lower-bound resources $\texttt{accepted}_j[e] \overset{\text{list}}{\sqsupseteq} \ell + [op]$ from all servers j. To get these lower bound resources from the backups to the primary, the proof uses Grove's RPC reasoning principles.

3.3 Reasoning about RPCs

Building on Grove's network model, the `rpc` verified Go RPC library provides reasoning principles that allow developers to

[2] The labels are such that server 0 is the primary and the rest are backups.

[3] Different variants of CSL come with different flavors of invariants. Here, we are explaining invariants as they work in Iris [23].

reason about RPCs much like how they would reason about local function calls in separation logic. Key to this RPC specification is its use of duplicable assertions. Formally, an assertion P is called duplicable if P implies $P * P$, meaning that it's possible to create a copy of any resource in P. For instance, $a \overset{\text{list}}{\mapsto} \ell$ is not duplicable because one thread's ownership of this resource precludes any other thread from owning it. On the other hand, knowledge such as $a \overset{\text{list}}{\sqsupseteq} \ell$ is duplicable.

The notion of duplicability allows stating Grove's RPC specification: for any function `f` with specification $\{P\}$ `f` $\{Q\}$, the specification for invoking `f` through an RPC is $\{P\}$ `rpcClient.Call("f")` $\{Q\}$, as long as P is duplicable. Duplicability of P is crucial because the RPC library may retransmit its request multiple times before it receives a response and each execution of `f` will consume one instance of P. Note that the specification does not, strictly speaking, require Q to be duplicable, because Grove obtains a fresh copy of Q from each invocation of `f()` on the server.

Example. As an example, consider the `ApplyAsBackup` RPC in vKV, issued by the primary to backup servers when replicating a new operation. The postcondition of `ApplyAsBackupRPC(e, index, op)` to server j is the assertion $\texttt{accepted}_j[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]$. This represents a promise that server j accepted `op` in its log. As the primary collects more of these lower-bound resources, it will eventually have enough to commit the operation using the replication invariant I_{rep}.

When it comes to choosing the precondition of `ApplyAsBackupRPC(e, index, op)`, one might naively pick "ownership of resources to apply operation `op` once." But, such a precondition is not duplicable, as required by RPCs. A recurring pattern when specifying RPCs with Grove is rephrasing such preconditions to not involve any exclusive ownership of resources, but instead talk about knowledge. The (correct) precondition for the `ApplyAsBackup` RPC is "knowledge that `op` is the operation at position `index`." The full spec (ignoring errors) for an `ApplyAsBackupRPC` is:

$$\{\text{knowledge that } \texttt{op} \text{ is operation at } \texttt{index}\}$$
$$\texttt{server}_j\texttt{.ApplyAsBackupRPC(e, index, op)}$$
$$\left\{\texttt{accepted}_j[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]\right\}$$

This precondition allows the primary to retry RPCs to ensure that every backup has learned about the operation.

3.4 Reasoning about leases

To reason about leases, the Grove library provides the notion of a *time-bounded invariant*. The invariant contains some resources R representing what the lease L promises to maintain until its expiration, denoted by $\boxed{R}_L$, and a separate resource representing the expiration time *exp* of the lease, denoted by $L \overset{\text{expires}}{\mapsto} \mathit{exp}$. L is a logical identifier for the lease, and does not show up in execution.

As an example, vKV uses a lease to promise that the configuration epoch number will remain the same, which ensures that no reconfiguration will take place for the duration of the lease (which in turn allows replicas to handle read-only requests on their own). When the configuration service hands out such a lease, it creates a time-bounded invariant $\boxed{\mathsf{CurrentEpoch} \mapsto e}_L$, along with a resource indicating when the lease expires, $L \overset{\text{expires}}{\mapsto} \mathit{exp}$. It then gives out the invariant and a duplicable version of the lease expiration resource, $L \overset{\text{expires}}{\geq} \mathit{exp}$; having a duplicable lower bound on the expiration time, as opposed to the exact expiration time, simplifies lease renewal.

There are four rules for time-bounded invariants. First, a time-bounded invariant can be *created* by giving up ownership of some resource R, and specifying a time at which it will expire. For instance, vKV's configuration service does this when issuing a lease in response to a `GetLease()` RPC, giving up its ownership of $\mathsf{CurrentEpoch} \mapsto e$.

Next, if a time-bounded invariant *expires*, according to the $L \overset{\text{expires}}{\mapsto} \mathit{exp}$ resource, its resources can be reclaimed. vKV's configuration service does this as part of reconfiguration: `TryWriteConfig()` waits for lease expiration, and gets back ownership of $\mathsf{CurrentEpoch} \mapsto e$, which it can then increment to $\mathsf{CurrentEpoch} \mapsto e + 1$.

Third, a time-bounded invariant can be *extended*: in vKV, the configuration service owns $L \overset{\text{expires}}{\mapsto} \mathit{exp}$ if there is an existing lease, and if another `GetLease()` RPC arrives, the configuration service extends the lease by advancing the expiration time to $L \overset{\text{expires}}{\mapsto} \mathit{exp} + \Delta$ (and sends a duplicable $L \overset{\text{expires}}{\geq} \mathit{exp} + \Delta$ to the caller).

Finally, the resources inside of the time-bounded invariant can be accessed by *opening* the invariant, as long as the time-bounded invariant has not expired. Opening a time-bounded invariant comes with the same obligations as opening a regular invariant—that is, the proof gets ownership of the resources from the invariant, but must return them back to close the invariant after at most one atomic step. In vKV, the $\mathsf{CurrentEpoch} \mapsto e$ resource inside the lease invariant is accessed by the primary-backup replication library (in contrast with the previous three operations, which all happen on the configuration service).

3.5 Reasoning about clocks

Consider the lease expiration check in vKV shown in Figure 7. This pattern is tricky to reason about: by the time `s.stateLogger.LocalApplyReadonly()` runs, the lease may no longer be valid, if there was a long delay right after `if` statement's check. As a result, whatever invariant the lease was protecting might no longer be true by the time the developer wants to use it in their proof.

To address this proof challenge, Grove's specification for `GetTimeRange()` allows the developer to perform arbitrary

proof steps (such as opening and closing invariants and updating ghost resources) at the instant when `GetTimeRange()` executes. In the context of these proof steps, the developer also gets access to a CurrentTime $\mapsto t$ resource which represents the current time, and a promise that the return value r of `GetTimeRange()` satisfies $r.\texttt{earliest} \leq t \leq r.\texttt{latest}$. (Grove implements this using logical atomicity [20].)

One subtlety is that, at the instant that `GetTimeRange()` executes, the code has not yet executed the comparison checking if the lease is still valid (i.e., comparing to `leaseExpiry`). As a result, once the proof gets the CurrentTime $\mapsto t$ resource, the developer must explicitly consider two cases at the instant of `GetTimeRange()`: either the subsequent check will succeed or it will fail. In the case where the subsequent check will succeed, the developer can use CurrentTime $\mapsto t$ to then open the time-limited invariant and access the CurrentEpoch $\mapsto e$ resource inside it. (§4.3 has a more detailed discussion of how this allows proving linearizability for reads.) When the proof eventually considers the actual comparison in the `if` statement, only one of the `if` branches will be viable in each of the two proof cases.

4 How Grove rules out bugs

This section sketches the specification and proof for several components from §2. It also poses some tricky scenarios for the components and explains either (1) why the scenario would result in buggy behavior and where the proof would get stuck because of the bug, or (2) why the scenario is subtly safe and how the proof covers it. A common theme is that the proofs center around choosing the right kinds of resources and do not need to break into cases for the different scenarios.

4.1 Primary replica server

Specification. An important part of vRSM's primary/backup replication protocol is embodied in the primary server's `Apply` function, whose job is to add a new request to the log on the primary as well as all backup replicas. Figure 4 shows the simplified code for this function, and in the rest of this subsection, we will walk through the proof of its correctness.

The spec we aim to prove for `Apply` is that its postcondition is $\exists \ell$, $\texttt{committed} \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]$. This is a formal way of stating that, after `Apply(op)` is done, the client's `op` definitely shows up in the committed log somewhere. The `op` might appear in the log after a number of other operations, which may have come from other clients. Similarly, `op` might not be the latest operation in the log either, if other operations arrived after it; however, the use of $\sqsupseteq$ allows the postcondition to ignore subsequent parts of the log. The spec does allow the operation to be added multiple times; a stronger exactly-once spec can be obtained on top of this spec via the `exactlyonce` library.

Proof. The first line acquires the primary server lock, which serializes concurrent calls to `Apply`. In the proof, the postcondition of `s.mutex.Lock()` provides ownership of the primary's points-to resource, $\texttt{accepted}_0[e] \overset{\text{list}}{\mapsto} \ell$. While holding the lock, the primary establishes the order in which this `op` will execute (namely, `nextIndex`), and applies `op` to its local state. At this point, the proof updates the thread's ownership of $\texttt{accepted}_0[e] \overset{\text{list}}{\mapsto} \ell$ to $\texttt{accepted}_0[e] \overset{\text{list}}{\mapsto} \ell + [\texttt{op}]$, and gets the knowledge resource $\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]$ before releasing the lock. To release the lock, the proof must give up ownership of $\texttt{accepted}_0[e] \overset{\text{list}}{\mapsto} \ell + [\texttt{op}]$, but retains knowledge of $\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]$, which will be useful later on.

Next, the primary invokes `ApplyAsBackupRPC` concurrently on all of the backup servers, passing in `nextIndex` to ensure that `op` is added to the backup's log at the right position. Using the RPC spec shown at the end of §3.3, the primary gets a promise that the jth backup accepted the operation, in the form of the lower bound resource $\texttt{accepted}_{j+1}[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]$. The RPC spec is established in a separate proof.

The proof of `Apply` collects these postconditions of the n calls to `ApplyAsBackupRPC` to get the resources $(\texttt{accepted}_1[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]) * \cdots * (\texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}])$. At this point, the proof has enough lower bound resources to be certain that the operation is committed. The proof opens the invariant I_{rep} defined in §3.2.3 and temporarily gets ownership of $\texttt{committed} \overset{\text{list}}{\mapsto} \ell$.[4] The proof then updates it to $\texttt{committed} \overset{\text{list}}{\mapsto} \ell + [\texttt{op}]$. With this points-to, the proof gets the lower-bound resource $\texttt{committed} \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]$, which is needed for the postcondition of `Apply`.

Before the proof can complete, it must close the invariant I_{rep} by returning ownership of the committed points-to. To close the invariant I_{rep}, the proof gives up the resources

$$(\texttt{committed} \overset{\text{list}}{\mapsto} \ell + [\texttt{op}]) *$$
$$(\texttt{accepted}_0[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]) * \cdots * (\texttt{accepted}_n[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]),$$

which matches I_{rep}'s body with $\ell + [\texttt{op}]$ in place of ℓ.

Since the lower-bound resource for the committed list matches the desired postcondition, the proof is complete.

What if a backup concurrently applies more operations? If backup j applies an operation concurrently, it will end up growing its $\texttt{accepted}_j[e]$ list (that is what happens, for instance, during `ApplyAsBackupRPC`). But, since the points-to is append-only, the lower bound resource $\texttt{accepted}_j[e] \overset{\text{list}}{\sqsupseteq} \ell + [\texttt{op}]$ that the primary received from calling `ApplyAsBackupRPC` is still valid, which captures the fact that the operation `op` appears in backup j's log, even if it's not the latest. Since the operation is still in all the backups' logs, it is safe for the primary to reply `OK`.

[4] Strictly speaking, the invariant refers to some l', but the proof can handle the case $l' \neq l$ through a combination of available facts and invariants.

4.2 Reconfiguration

A separate challenging aspect of vRSM lies in reconfiguration. This subsection will walk through the proof of `Reconfigure`, as shown in Figure 6. This function is invoked on an operator's machine when the operator wants to change the set of servers, perhaps adding new machines to replace failed ones. The spec is the following:

$\{$all `newServers` are valid replica servers$\}$
`Reconfigure(newServers)`
$\{\top\}$

This specification states that calling `Reconfigure()` requires all of the servers specified in `newServers` to be already running the vRSM replica software (although the servers might have been just installed, containing no key-value state), so that the reconfiguration logic knows it is safe to use them as a replica. `Reconfigure()` will contact both the old servers and the new servers, transferring the state to the new servers before registering them with the configuration service.

The postcondition in `Reconfigure()`'s specification appears to be weak, in the sense that it does not promise that `Reconfigure()` will make progress. This is because Grove is limited to safety properties—ensuring that vRSM never returns the wrong result—as opposed to liveness, such as guaranteeing that a client will receive a response. However, the $\top$ postcondition *does* actually guarantee an absence of all safety bugs during reconfiguration, such as corrupting the state sent to the new servers, losing some operations applied concurrently by the old servers, etc. This is because `Reconfigure()` does not own any interesting resources to start with, which precludes it from tampering with any resources held by the rest of vRSM. Any resources that `Reconfigure()` obtains must come from invariants, such as I_{rep}. However, Grove requires that the proof correctly re-establishes any invariant that is opened, thereby ensuring that the invariants are maintained throughout the execution of `Reconfigure()`.

Proof. The overall structure of `Reconfigure()` is to get a new epoch, then choose one of the old servers to seal and get a copy of the old state from, then send this state to all of the new servers, register the configuration, and activate the new primary. The proof relies on the fact that Grove's append-only list resource, used to represent vRSM's log, is indexed by epoch.

A key aspect to the proof lies in the resource returned to `Reconfigure()` by the `GetStateAndSeal` RPC. The postcondition of `GetStateAndSeal(newEpoch)` is:

$$\exists \ell,\ (\texttt{oldState} \text{ corresponds to log of operations } \ell) * \texttt{accepted}_j[e] \overset{\text{list}}{\mapsto}_{\square} \ell.$$

Once the proof receives ownership of the read-only resource $\texttt{accepted}_j[e] \overset{\text{list}}{\mapsto}_{\square} \ell$ for the old epoch, it can conclude that the old configuration will not apply any more operations. This is because replica j must have given up ownership of its $\texttt{accepted}_j[e] \overset{\text{list}}{\mapsto} \ell$ to produce the read-only resource, which precludes it from extending ℓ with more operations. After reconfiguration completes, a new append-only list resource $\texttt{accepted}_j[e+1]$ will be allocated (assuming the new epoch is $e+1$), which allows vRSM to append new operations.

As in the primary/backup replication proof, Grove allows the developer to prove the correctness of `Reconfigure()` with modular reasoning, without having to consider explicit interleavings with other parts of the system. Nonetheless, the proof does rule out bugs due to subtle interactions, as follows:

Why can't the old primary commit additional operations? The developer might worry that after `Reconfigure()` fetches the old state, the old primary could execute additional operations. This would mean that `Reconfigure()` will send an incomplete state to the new servers, losing an operation. This possibility is ruled out in vRSM's proof due to the read-only resource sent back by the `GetStateAndSeal` RPC. Having this read-only resource implies that the old primary cannot add any more operations to $\texttt{committed} \overset{\text{list}}{\mapsto} \ell$, since that would require adding the operation to every old replica, which would in turn contradict the read-only resource.

What if the log contains operations that were never committed? It is possible that the log obtained by `Reconfigure()` contains some operations that were never committed in the old configuration, for example if the primary sent the operation to some but not all backups before failing. However, it is nonetheless safe to commit those operations in the new config. This is because the primary first adds an operation to its own log before sending it to the backups. vRSM captures this in an invariant by stating that every operation in a backup's $\texttt{accepted}_j[e]$ must also appear at the same position in the primary's $\texttt{accepted}_0[e]$, making it safe to commit. A client clerk will learn the outcome of its operation when it resends its request to the servers in the latest configuration.

4.3 Lease-based reads

The key challenge for lease-based reads lies in ensuring that the result returned by `ApplyReadonly()`, as shown in Figure 7, reflects a properly linearized execution: the result cannot be stale (i.e., missing writes that have already finished), and cannot be rolled back (i.e., reflect uncommitted writes that might be lost due to a crash or reconfiguration).

vRSM proves that the value is not stale by using the lease. The read operation will be executed against the state of the local server, represented by $\texttt{accepted}_j[e]$. At the instant of the `GetTimeRange` invocation on line 4, the proof opens the lease invariant to obtain $\mathsf{CurrentEpoch} \mapsto e$, and opens the replication invariant I_{rep} to obtain the fact that all of the operations in `committed` are also in $\texttt{accepted}_j[e]$ (which has not been superseded by any higher epoch e').

To prove the second part (that the returned value cannot observe writes that will be rolled back), vRSM waits until all of the writes preceding the read's linearization point to the same key have been committed, in `waitForCommitted`. There are two cases to consider. First, there may be no pending writes, e.g., if at the instant of the `GetTimeRange` invocation, `idx` is already committed. In this case, the proof linearizes the read operation immediately at the instant of `GetTimeRange`. The second possibility is that there are some pending writes to commit. In this case, the proof maintains a set of ghost state updates (based on the helping pattern [4, 5]) that must be logically applied when the preceding write is committed (which happens in the primary server's proof of `Apply()`). Note that this case distinction is only possible in the proof; the code does not know which case it's in when running `GetTimeRange`, so `waitForCommitted` waits for `idx` to be committed in both cases (and, in the first case, returns right away). If reconfiguration happens in the meantime, and the epoch number changed before `idx` could be committed, the read result might not be valid, and the server tells the client to retry.

What if the server pauses for a long time after checking the lease? A typical concern with leases is the freshness of the lease check. What if the server running `ApplyReadonly()` does the lease check on line 4 of Figure 7, but then pauses for a long time (e.g., due to garbage collection) before actually executing the read operation on line 6? By that time, reconfiguration may have taken place, choosing a new primary, and that primary has executed more writes.

Such delays cannot violate correctness. Even if a new primary executes writes, the read will be linearized *before* those writes. This is allowed because the read request arrived before reconfiguration (since the lease check passed after the request was sent). The lock held by `ApplyReadonly()` ensures that the server's state does not change between the lease check and the execution of the read operation.

In the proof, the read operation is linearized at the instant of the `GetTimeRange()` invocation in Figure 7, if the returned latest time is less than the lease expiration time. This allows the proof to open the lease invariant to establish that the epoch is still e, and thus the in-memory state of that replica j that will be accessed by `LocalRead`, corresponding to $\texttt{accepted}_j[e]$, will be committed if `waitForCommitted()` succeeds.

4.4 Client cache consistency with leases

The top-level spec for the key-value caching library `cachekv` is that `Get` and `Put` behave like a linearizable key-value service, with `GetAndCache` working functionally like a `Get`.

To prove the linearizability of its lease-based caching, the library uses a ghost map resource, which has a $k \mapsto_{\text{kv}} v$ assertion representing the fact that the current value of k is v. The library maintains two invariants: one global invariant across all instances of the key-value caching library that share the same state, and one local lock invariant for each node's own cache, protected by a mutex lock on that node. The global invariant maintains, for each key, a time-bounded invariant $\boxed{k \mapsto_{\text{kv}} v}_L$ and ownership of the expiration time $L \overset{\text{expires}}{\mapsto} \mathit{exp}$, corresponding to the expiration time encoded in the underlying key-value pair. The expiration time may be in the past if the most recent lease on k has already expired. On each node, the library's lock invariant maintains $\boxed{k \mapsto_{\text{kv}} v}_L$ and a lower bound on the lease expiration time, $L \overset{\text{expires}}{\geq} \mathit{exp}$, corresponding to the expiration time stored in its local cache.

When a client executes `Get`, the library acquires its per-node lock and checks the lease expiration time for the requested key. If the key is cached and the lease is not expired, the library opens the time-bounded invariant to prove that the cached value is the current value for that key. `Put` waits until the lease expires, at which point its proof can reclaim the $k \mapsto_{\text{kv}} v$ resource from the lease, update it to the new value v', and then put it back into a new lease with the same (expired) expiration time, to re-establish the global invariant that every key corresponds to some lease. The proof of `GetAndCache` extends the lease in the global invariant, and makes a copy of the lease and the corresponding expiration time lower bound in its local node invariant.

Why can't a `Put` change a currently cached value? The proof of `Put` has to update $k \mapsto_{\text{kv}} v$ to maintain the invariant. To do this update, the proof needs ownership of the points-to. If a client currently has that key cached (and not expired), then the time-bounded invariant containing that key's points-to has not expired yet, and the `Put` proof cannot reclaim it.

Why can't `GetAndCache` decrease the lease expiration time instead of extending it? This would result in a bug because a `Put` might change the value while another client still believes it has an up-to-date cache. The proof would get stuck when re-establishing the global invariant with the decreased lease expiration time, because that would require updating ownership of the expiration time $L \overset{\text{expires}}{\mapsto} \mathit{exp}$ to a smaller number $\mathit{exp}' < \mathit{exp}$, which is not possible: the lease extend rule from §3.4 only allows the expiration time to increase.

5 Developing Grove proofs

A major benefit of Grove's use of concurrent separation logic is that it allows proving each function—and even each line of code—in isolation, without explicitly considering the interleavings due to concurrency, crashes, recovery, etc. This not only allows incrementally proving a single library, but also enables combining the proofs of multiple components or functions into a single proof for the composed system.

Composability of proofs is a powerful property that is not true in the general case. For example, if a system is running both a key-value service and a lock service, the key-value library might inadvertently send a message to the lock service that causes it to release a lock, thereby invalidating its proofs. As another example, if the developer adds a new RPC method

to a key-value server, and this RPC incorrectly updates data structures used by other RPCs, adding this RPC would invalidate the proofs of existing RPC methods. Concurrent separation logic enforces ownership rules to ensure that all verified code is "well-behaved" in a way that avoids problems like the above, and allows for sound composition.

The rest of this section presents several case studies that illustrate the benefits of concurrent separation logic in Grove.

5.1 Proving top-level spec for vKV

The top-level theorems for vKV are specifications for the top-level functions:

1. The replica server `main()` function is crash-idempotent [5, 37]. This covers the execution of any code invoked by `main()`, including any RPC handlers that `main()` sets up.
2. `Reconfigure(newServers)` is always safe to run, if `newServers` are all valid replica servers.
3. `MakeClerk(configAddrs)` correctly initializes a clerk, if `configAddrs` are the addresses of the `configservice`.
4. A `clerk`'s `Put(k,v)`, `Get(k)`, and `CondPut(k,e,v)` functions behave as though accessing a local in-memory key-value map with linearizable operations.

Proving these theorems involves proving specs for functions in vKV one-by-one, using specs for lower-level components to verify higher-level code, culminating in a proof of the top-level functions. A client application that uses vKV (e.g. the `bank`) can then be verified by applying these theorems to reason about `Put` and `Get` calls.

5.2 Evolving vKV to add leases

We originally built and verified the vRSM library and vKV without leases. This original version executed `Get` operations as a read-write operation; that is, by replicating the `Get` operation to the primary and all backups (including waiting for the replicas before replying to the client).

We later decided to improve performance of read-only operations by adding leases. This involved several changes: (1) adding `GetLease` to the configuration service and making sure other RPCs wait for any outstanding leases to expire before advancing the epoch number; (2) adding `ApplyReadonly` to the vRSM library as well as a helper thread on each replica that extends its lease with the `GetLease` RPC; (3) propagating the number of committed operations from the primary to the backups; (4) introducing `VersionedStateMachine` as the vRSM library's interface to allow some reads to happen without waiting for ongoing writes to finish; and (5) bypassing the exactly-once operations library for read-only operations.

In the proof before adding leases, the configuration service always owned the epoch number and could always advance it. With leases, ownership may reside in a time-bounded invariant, so the proof now must establish ownership by using the fact that the code checks for lease expiration, which allows the proof to use the time-bounded invariant expiration rule from 3.4. To reason about linearizability of lease-based reads from replica servers, the proof of the primary/backup replication and reconfiguration protocol remained the same, but we added a new proof on top that shows that ownership of the current epoch means any replica's state is at least as up-to-date as the committed operations.

Component	Code	Spec and Proof
`bank`	99	799
`lockservice`	19	133
`cachekv`	86	569
vKV	233	1,574
`exactlyonce`	127	2,272
`clerk`	146	935
`storage`	227	3,057
`configservice`	200	2,797
`paxos`	492	5,600
`reconfig`	65	817
`replica`	578	8,093
Time-bounded invariants	–	168
`rpc`	163	1,263
Network library	120	Trusted
Filesystem library	50	Trusted
Total	2,605	–
Total verified	2,435	28,077

Figure 10: Lines of Go code and Coq spec/proof for the verified components.

6 Implementation

Grove is implemented by extending Perennial [5], which is based on Iris [22, 23, 27] and Coq [39]. Grove inherits reasoning principles for concurrent Go code from Perennial, inherits general support for interactive separation logic reasoning and ghost resources from Iris, and adds support for distributed systems with new reasoning principles for the network, clock, and independent node crashes. Grove's extensions to Perennial involved 1,597 lines of Coq proof for new reasoning principles, along with other hard-to-quantify minor changes throughout Perennial. Grove comes with a distributed composition soundness theorem, which proves correctness of Grove's reasoning principles by showing that they imply a simple statement about the behavior of the distributed system under Grove's execution model.

Figure 10 shows the breakdown of code and proof for the different components. The top-level specification of the bank, which builds on most of the other components, is 52 lines. The specification for vKV, consisting of the four parts described in §5.1, is 49 lines. We confirm that the proof is complete using `Print Assumptions` in Coq. Across the different components, verification required 12× the lines of proof as lines of code, which is comparable to other concurrent and distributed systems verification projects: IronFleet's overhead is slightly lower [18] (and IronFleet also includes a proof of liveness, though it does not handle thread concurrency, leases, crashes,

or reconfiguration), but GoJournal's is slightly higher [6]. One conclusion is that verifying a complete distributed system, such as vKV, which handles node-local concurrency, crash recovery, leases, and reconfiguration, did not come at the cost of an inflated proof overhead.

7 Evaluation

To demonstrate that Grove is capable of verifying realistic high-performance distributed systems, this section experimentally demonstrates that the vKV prototype, which we verified using Grove, is able to achieve high performance. We also demonstrate that leases are particularly important for achieving high performance for reads in vKV.

Experimental setup. To evaluate vKV's performance, we use 8 CloudLab servers, with up to 3 for replicas, 4 for clients, and 1 for the configuration service. Each machine has an Intel Xeon CPU E5-2630v3 2.4GHz processor with 8 cores, 64GB of RAM, an Intel 200GB 6Gb/s SSD (SSDSC2BX200G4R) for storage, and an I350 Gigabit network card.

We generate requests using YCSB [9] with uniformly random keys and 128-byte values. Clients run in a closed loop, issuing a new request as soon as the previous request completes; for each data point, we warm up the system for 20 seconds and then measure the performance for 1 minute. To measure throughput, we keep increasing the number of clients until the total throughput of all of the clients stops growing.

Baseline performance. To demonstrate that vKV achieves good performance, we compare with Redis. Redis is a widely-used high-performance key-value server, written in C. Redis targets somewhat different goals than vKV (it is designed to run on a single core, it does not support synchronous replication or live reconfiguration, etc), but it nonetheless provides a reference point in terms of absolute performance for a key-value store. To make Redis comparable to vKV in terms of its guarantees, we run Redis with the `appendfsync always` option to ensure it made changes durable before replying, and we run vKV on a single core (disabling all other cores in Linux) and with no backup replicas. Note that Redis does not implement exactly-once semantics for its operations (if a write gets retransmitted, it may end up being executed twice), whereas vKV stores a 16-byte request ID for each operation.

Figure 11 compares the performance of vKV with that of Redis. We report the mean of 10 runs; Redis's standard deviation is 1–2%, and vKV's is 7–11%, due to the high variance of the Go runtime when running on a single core. When running on multiple cores, vKV achieves higher throughput—e.g., 5.1× on 8 cores for YCSB 5% writes, with minimal performance variability. The results show that vKV's throughput is 67–73% of Redis's, and its request latency is comparable.

Reconfiguration. To demonstrate that vKV can recover from server failures by reconfiguring the system to add new servers, all while continuing to correctly handle client requests, we run a two-server configuration of vKV. At 10 seconds into the experiment, the primary server is killed, and reconfiguration starts (changing to a new primary and a new backup server). We use a variation of the YCSB workload, with 100 clients always issuing writes, and 100 clients always issuing reads (rather than each client issuing a mix). This is because, during reconfiguration, writes block if one of the servers is sealed (which would ultimately cause all clients to block if they were issuing reads and writes), but reads can proceed (so clients that never issue writes can proceed).

Benchmark	Redis	vKV
Throughput for YCSB 100% writes	99,066 req/s	67,360 req/s
Throughput for YCSB 50% writes	107,028 req/s	75,174 req/s
Throughput for YCSB 5% writes	118,594 req/s	87,634 req/s
Read latency under low load	81 us	126 us
Write latency under low load	538 us	603 us

Figure 11: Throughput and latency of vKV compared to Redis.

Figure 12 shows the observed throughput by the read and write clients over time during this experiment. The results show that vKV can continue serving reads while reconfiguring. When the primary is initially killed, read throughput dips while clients with outstanding read requests sent to the primary wait to discover their connection is closed and while the remaining backup server marshals its key-value state to be sent to the new servers. After the backup is done marshaling its state, and after the clients connect to the backup and retransmit their requests, reads recover some of the throughput. Reads do not recover to their original throughput because of stuck reads: for a client that tries to read one of the keys whose write was in flight when the primary was killed, `waitForCommitted` returns only after reconfiguration, because the old primary did not commit those writes before being killed (and the backup doesn't know yet that those writes will not be committed by the primary). As more read clients get stuck, read throughput starts declining again. After the state is transferred to new servers (copying 1M key-value pairs, each 128 bytes long), the system switches to the new configuration and resumes executing reads and writes (including all previously-stuck operations). Most of the reconfiguration time is spent marshalling the state and sending it to new servers via the reconfiguration process, (∼4 seconds in total).

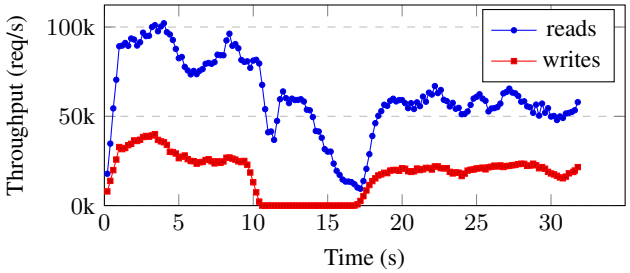

Figure 12: Throughput over time (averaged over 0.5 second time slices), with the primary crashing at 10 seconds, followed immediately by a reconfiguration to a new primary and backup.

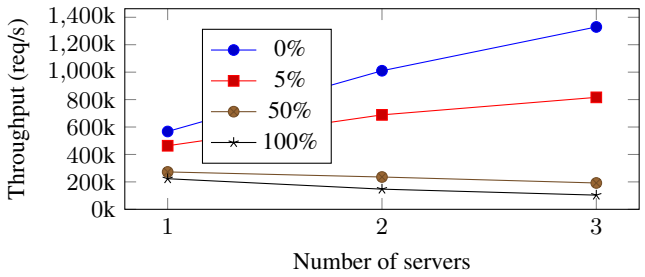

Figure 13: Peak throughput of vKV with increasing number of servers, labeled by the percentage of write operations.

Read performance with leases. Figure 13 shows vKV's throughput for different workloads as more replicas are added. For write-heavy workloads (50% or 100% writes), adding replicas reduces performance because writes encounter more overhead at the primary server, and there are not enough reads handled by other replicas to offset the costs. For read-heavy workloads, adding replicas improves performance—e.g., for YCSB 5% and 0% writes, 3 servers achieve 1.7× and 2.3× the throughput of a single server, respectively.

8 Related work

Grove is the first to support verifying distributed systems with thread- and node-level concurrency, crash recovery with durable state, time-based leases, and reconfiguration. Verifying all of these aspects in a single framework is critical because subtle bugs can occur due to interactions between these features. vKV, a realistic replicated key-value store, demonstrates the benefits of Grove's modular reasoning by proving the correctness of its primary/backup replication, durable storage, reconfiguration, concurrency, and leases. vKV's design is not novel, but rather a case study of what it takes to build a fault-tolerant primary-backup replication system. In doing so, it captures key challenges in state-of-the-art (unverified) distributed systems with primary/backup replication and a configuration service for reconfiguration, such as Chain Replication [40], FaRM [11], Boxwood [32], Bigtable [7], Megastore [2], FoundationDB [42], Kafka [24], and Tuba [1].

Concurrent separation logic for distributed systems. Broadly similar to our work, Disel [36] and Aneris [14, 28] also use concurrent separation logic in the context of distributed systems. However, neither Disel nor Aneris provide support for reasoning about time-based leases or recovery from crashes, and they have not been used to verify a system with reconfiguration. These restrictions limit the distributed systems they can reason about. For example, Gondelman et al. [15] use Aneris to verify an eventually-consistent primary-backup key-value store. However, that system does not support reconfiguration, so if the primary fails, the system cannot process any further writes. Furthermore, writes are only lazily copied to replicas for availability, and thus reads from replicas may return stale values. vKV uses a combination of reconfiguration and leases for availability when a primary fails, while also guaranteeing that reads from replicas are up-to-date.

State-machine refinement. An alternative approach to verifying distributed systems is to prove refinements from a high-level protocol description down to executable code, as in IronFleet [18], Verdi [41], and IronSync [17]. However, these systems do not reason about time-based leases, reconfiguration, or node recovery. State machines also make it challenging to compose larger systems out of smaller components, which features extensively in our case study. IronSync [17] shows how to bring some benefits of ownership-based reasoning to state-machine approaches, but at a coarse granularity.

Distributed system abstractions. Adore [19] proposes an abstraction for reasoning about reconfiguration for replicated state machine protocols, such as Raft. vKV's primary/backup replication and reconfiguration uses a configuration service to simplify the protocol, but verifies many of the same issues, such as concurrent request execution during reconfiguration. vKV also handles interactions between reconfiguration and crashes, recovery, leases, and thread-level concurrency, which the Adore abstraction does not directly address.

Protocol reasoning. TLA$^+$ [29, 30] provides a modeling language for concisely describing distributed protocols, which can then be model-checked or interactively verified. In other tools, constraining the modeling language used for expressing protocols enables automatic or semi-automatic proofs of correctness, such as ByMC [26], Ivy [33, 35], and I4 [31] and its follow-ons. Although protocol verification can ensure the absence of bugs in the protocol design, many bugs in distributed systems only manifest at the level of implementations, and so fall outside the scope of protocol verification. Grove aims to verify implementations of systems to address these bugs.

9 Conclusion

Grove is a library for verifying distributed systems using concurrent separation logic (CSL). Grove generalizes CSL to support distributed systems with RPCs, leases, replication, reconfiguration, and crash recovery. We demonstrate Grove by implementing and verifying a range of distributed system components, such as primary-backup replication, locking, client caching, and a configuration service. Verifying these components in Grove eliminates broad classes of bugs, and comes with a 12× proof-to-code ratio, in line with previous efforts to verify concurrent and distributed systems. vKV, a key-value store built out of these components, supports primary-backup replication and reconfiguration, achieves 67-73% the throughput of Redis on a single core, and scales read throughput with more replicas due to its use of leases.

Acknowledgments

Thanks to the anonymous reviewers, Jay Lorch, members of the MIT PDOS group, and our shepherd, Chris Hawblitzel, for feedback that improved this paper. This work was supported by NSF awards CCF-2123864 and CCF-2318722.

References

[1] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, October 2014.

[2] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, Asilomar, CA, January 2011.

[3] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[4] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 2015 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 241–250, Donostia-San Sebastián, Spain, July 2015.

[5] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai 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, October 2019.

[6] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–439, Virtual conference, July 2021.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[8] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, October 2015.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, Indianapolis, IN, June 2010.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 54–70, Monterey, CA, October 2015.

[12] Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *Proceedings of the 2022 USENIX Annual Technical Conference*, Carlsbad, CA, July 2022.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[14] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. Distributed causal memory: modular specification and verification in higher-order distributed separation logic. In *Proceedings of the 48th ACM Symposium on Principles of Programming Languages (POPL)*, Virtual conference, January 2021.

[15] Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. Verifying reliable network components in a distributed separation logic with dependent separation protocols. In *Proceedings of the 28th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Seattle, WA, September 2023.

[16] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, Litchfield Park, AZ, December 1989.

[17] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2023.

[18] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Monterey, CA, October 2015.

[19] Wolf Honore, Ji-Yong Shin, Jieung Kim, and Zhong 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.

[20] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–282, Austin, TX, January 2011.

[21] Cliff B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In A. W. Roscoe, Cliff. B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 167–187. Springer, 2010.

[22] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, January 2015.

[23] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[24] Apache Kafka. `https://cwiki.apache.org/confluence/display/kafka/kafka+replication`, 2013. Accessed: 2023-04-10.

[25] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, December 1999.

[26] Igor Konnov and Josef Widder. ByMC: Byzantine model checker. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation*, pages 327–342, Limassol, Cyprus, November 2018.

[27] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 205–217, Paris, France, January 2017.

[28] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars 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, April 2020.

[29] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[30] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X.

[31] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019.

[32] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–120, San Francisco, CA, December 2004.

[33] Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV)*, pages 190–202, Los Angeles, CA, July 2020.

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

[35] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. In *Proceedings of the 32nd Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 108:1–108:31, Vancouver, Canada, October 2017.

[36] Ilya Sergey, James R. Wilcox, and Zachary 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, January 2018.

[37] Upamanyu Sharma. Modular verification of distributed systems with Grove. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 2022.

[38] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Grove: a separation-logic library for verifying distributed systems (extended version). arXiv:2309.03046 [cs.LO], September 2023. Available at `https://arxiv.org/abs/2309.03046`.

[39] The Coq Development Team. *The Coq Proof Assistant, version 8.17.1*, June 2023. URL `https://doi.org/10.5281/zenodo.8161141`.

[40] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[41] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas 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.

[42] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, pages 2653–2666, Virtual conference, June 2021.