Modular Verification of Distributed Systems with Grove

by

Upamanyu Sharma

B.S., University of Michigan (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
August 26, 2022

Certified by .....
M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Nickolai Zeldovich
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Ralf Jung
Post-doctoral Researcher, Computer Science and Artifical Intelligence Lab
Thesis Reader

Certified by .....
Joseph Tassarotti
Assistant Professor, New York University
Thesis Reader

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Modular Verification of Distributed Systems with Grove

by

Upamanyu Sharma

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Grove is a new framework for machine-checked verification of distributed systems. Grove focuses on modular verification. It enables developers to state and prove specifications for their components (e.g. an RPC library), and to use those specifications when proving the correctness of components that build on it (e.g. a key value service built on RPC).

To enable modular specification and verification in a distributed systems, Grove uses the idea of ownership from separation logic. Using Grove, we built a verified unreliable RPC library, where we captured unreliability in the formal specification by using *duplicable ownership*. We also built a verified exactly-once RPC library, where we reasoned about ownership transfer from the client to server (and back) over an unreliable network by using the *escrow* pattern.

Overall, we developed and verified an example system written in Go consisting of the RPC libraries, a sharded key-value store with support for dynamically adding new servers and rebalancing shards, a lock service, and a bank application that supports atomic transfers across accounts that live in different shards, built on top of these services. The key-value service scales well with the number of servers and the number of cores per server. The proofs are mechanized in the Coq proof assistant using the Iris [12] library and Goose [5].

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Nickolai Zeldovich
Title: Professor of Electrical Engineering and Computer Science

Thesis Reader : Ralf Jung
Title: Post-doctoral Researcher, Computer Science and Artifical Intelligence Lab

Thesis Reader : Joseph Tassarotti
Title: Assistant Professor, New York University

Acknowledgments

This thesis was made possible through the guidance of several people, to whom I am grateful.

First, I want to thank my advisors Nickolai and Frans for their continued guidance, specifically in this project, as well in as many unrelated discussions about systems, verification, security, etc. Next, the work in this thesis was only completed because of Ralf and Joe's help in initially learning Iris, in formulating plans of attack for proofs, and in writing actual proofs. Most importantly, all four of these people helped guide major project decisions (such as "what to build and verify") and offered a great deal of help and direct feedback for the writing of this thesis. I want to thank the PDOS group for useful feedback and discussion about this work, which sharpened the presentation of our ideas. Especially, I want to thank Tej Chajed, who helped me learn Iris, and whose Perennial work is the basis for this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Building distributed applications is difficult because many clients interact concurrently with many servers over an unreliable network. This leads to a large number of possible application behaviors that all have to be considered. Formal verification provides techniques to systematically reason about all possible behaviors, and there has been significant interest in applying formal methods to make distributed applications more robust [1, 8, 10, 17, 20, 22, 25, 29, 32].

This master's thesis presents Grove, a new framework for verifying distributed systems. Grove focuses on modular verification of distributed systems, which means being able to prove specifications about individual components and then using those specifications to verify systems built on top, whether they be other services or applications. As an example, Grove can be used to specify and verify a key-value server, which in turn can be used in a black-box fashion to verify other services, such as a sharded key-value service or a lock service. These services can then be used by an application developer to verify a distributed application such as a bank with atomic transfers across accounts. This mirrors how distributed systems are built in practice, and avoids the need for the developers of one component to reason about the internals of other components that they are building on, which may be designed and implemented by other developers.

Stating precise specifications of distributed system components and using them to verify the rest of the system is challenging for several reasons:

Coordination. Reasoning about correctness of distributed systems often requires reasoning about coordination of operations across threads running on many machines. For instance, an application built on top of a key-value store might use a distributed lock service to coordinate access to the shared key-value pairs. When one thread acquires a lock, it can safely access the key-value pairs protected by that lock, by assuming that no other thread—either on the same machine or on other machines—will do so concurrently. To formalize such reasoning, Grove adopts the notion of *ownership* from separation logic [28]. Threads can own logical objects, such as a key-value pair, which gives them the right to access it. By ensuring that only one thread can own an object at a time, ownership can provide exclusive access to objects. Moreover, transfer of ownership between threads running on many machines, such as moving around ownership of a key-value pair when a lock is acquired or released, allows reasoning about dynamic coordination in a distributed system. Grove reasons about ownership using the Iris separation logic framework [12].

Packet loss. Network messages can be lost and retransmitted, leading to complex behaviors, such as a request being executed multiple times. For example, a key-value service might use an unreliable RPC library to ask a shard-coordinator which server is responsible for storing a particular key. If the request is lost, it will be retransmitted until a reply arrives. Explicitly exposing the network-level packet loss and retransmission to the client requires the application developer to reason about a large number of possible scenarios. Grove provides a concise specification for unreliable RPCs that captures the notion of an operation that might be executed multiple times using the idea of *duplicable ownership*. For example, an RPC that looks up the shards-to-server mapping on the coordinator server does not does not require any exclusive ownership: it is acceptable for multiple lookups and shard mapping updates to execute concurrently, since the lookup result is only a hint to the client. Grove captures this pattern by representing the shard mapping using such a duplicable kind of ownership, allowing clients to concurrently query it.

Verifying Exactly-Once RPCs. Applications often benefit from stronger semantics than unreliable RPCs, such as exactly-once RPCs, which is typically achieved

using sequence numbers and a table for tracking duplicates. However, reasoning about ownership transfer for exactly-once RPCs built on top of an unreliable network is challenging: it is not clear which exact network message will convey ownership from one machine to another. For example, when a client receives a response to a request for acquiring a lock, it should receive ownership of the key-value pair protected by that lock. However, if the client does not get a response to its first request, it must send another one. At this point, a naïve verification plan might be stuck, because the server no longer has ownership of the key-value pair to send back to the client—it sent it with the first (lost) response.

Grove addresses this problem using the idea of *escrows* [31], where ownership transfer logically occurs later, when a network message has been successfully delivered and processed.

Verifying Clerks. To simplify the life of application developers, distributed services often provide client libraries that conveniently expose network operations as simple function calls. It is also common for these client libraries (or *clerks*) to contain some non-trivial parts of the logic implementing a distributed service. For example, the `Get` clerk for a sharded key-value service first issues a lookup RPC to the coordinator to find the server responsible for a given key, then issues a `Get` RPC to that server, potentially retrying if the shard has been migrated in the meantime. Grove allows the developers of the key-value service to prove specifications for such client-side clerks, freeing the application developer from having to reason about individual RPCs, retry logic, etc. This can be challenging because clerks face many kinds of concurrency: multiple threads running on the same client machine, concurrent operations in the network, and requests running concurrently on servers. This leads to a large number of interleavings for the multiple requests issued by clerks, such as shard migration between a coordinator lookup and the `Get` RPC.

Grove extends the Iris concurrent separation logic [11, 12] to formalize distributed systems, which enables Grove to reason about concurrency within the clerk while still providing a concise specification for the clerk to the client. Using separation logic also enables multiple verified components to be composed to achieve a complete proof of

correctness.

Grove uses Goose [5] and Perennial [4] (a Go verification framework based on Iris) to allow developers to write their code in Go. Using Grove, we built and verified unreliable RPC library as well as an exactly-once RPC library, with precise specifications and proofs of their correctness on top of an unreliable network. To further demonstrate that Grove's ideas work well, we developed example applications that capture core aspects of distributed systems: GroveKV is a distributed sharded key-value service with support for dynamically adding new servers and rebalancing the shards appropriately. On top of GroveKV, we built a lock service and a simple bank application that uses the lock and key value services to implement atomic transfers.

Using Grove, we are able to prove the correctness of the example applications. Our evaluation also shows that the implementation achieves reasonable performance: it is able to sustain a throughput of 80k req/s on a single core in comparison to 125k reqs/s for the Redis key-value store [27].

To summarize, the main contributions of this thesis are:

- An extension of the Perennial [4] concurrent separation logic (CSL) framework to reason about Go code across multiple machines that communicate over an unreliable network (chapter 3).
- A specification and proof for an unreliable RPC library, using duplicable ownership to capture unreliability (chapter 4).
- A verified exactly-once RPC library, which makes use of the escrows proof technique to reason about ownership transfer over an unreliable network (chapter 5).
- A verified example distributed system (a key value service) and application (bank) as a case study of verifying systems with Grove (chapter 6), along with a performance evaluation of the verified key-value service (chapter 8).

The most closely related work is Aneris [8, 17]. They also use concurrent separation logic for verifying distributed systems. Unlike Aneris, this thesis focuses on verifying RPCs and on applying this verification approach to a realistic, runnable system.

One limitation of Grove is that it does not deal with crash recovery or liveness reasoning. We believe that Grove can be extended to handle crash recovery in future work, but we have not yet done so. The applications we have developed so far are also quite simple. However, we believe they are realistic in terms of showing that one can verify applications with Grove that perform well and that involve core challenges that often show up in distributed systems, such as dynamic reconfiguration, retransmission, and concurrent requests.

Our Go code can be found at `https://github.com/mit-pdos/gokv` (the `memkv` package contains our key-value service, and `bank` has our bank example).

Our proofs can be found in the Perennial repository at `https://github.com/mit-pdos/perennial/tree/master/src/program_proof/grove_shared` (for the RPC libraries) and at `https://github.com/mit-pdos/perennial/tree/master/src/program_proof/memkv` (for GroveKV and the bank).

Chapter 2

Verification Goal

This section presents the example system as summarized in Figure 2-1. This system consists of multiple layers, from an unreliable RPC library to key-value and lock services, and ultimately a top-level application (a bank). Our goal is to verify all of the components by giving them precise specifications, and by using these specifications to verify code at higher layers. To enable verification while also achieving reasonable performance, all of the code in our example system is implemented in the subset of Go that Goose [5] supports.

Network. Our example system is built on top of Grove's network API, which provides a typical socket-like API, with procedures for `Send`, `Receive`, etc. The implementation of the network API is trusted (as indicated by the shading), which is not a strong assumption since Grove models the network as asynchronous, allowing for arbitrary reordering, duplicating, and dropping of network messages.

uRPC. On top of the networking API we implement a library for unreliable remote procedure calls, uRPC. It allows a server to provide operations remotely over the network and allows clients to invoke these operations like local calls. However, unlike local calls, a single RPC invocation can cause the server to execute an operation multiple (or zero) times.

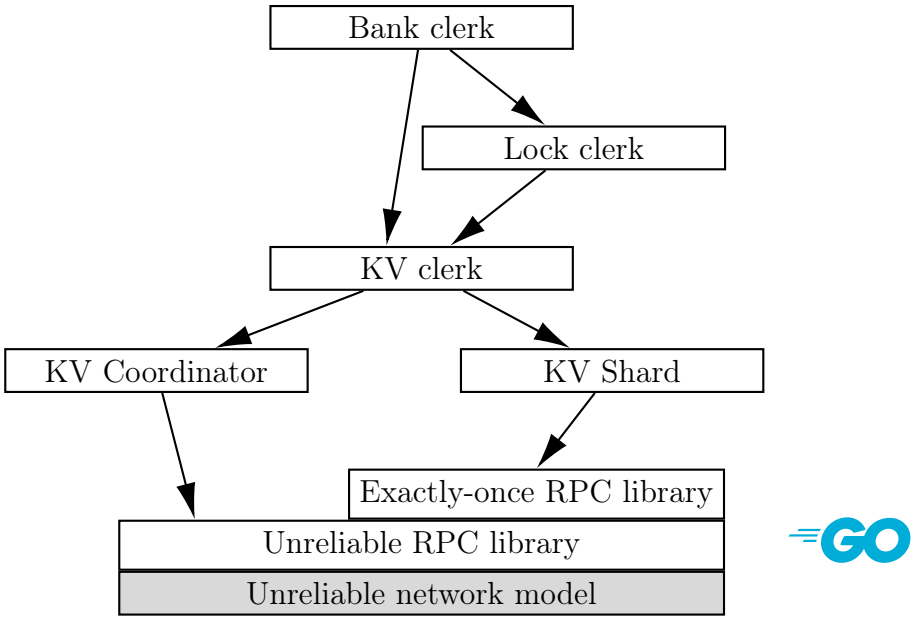

Figure 2-1: Layers of the example implementation. The network layer is trusted; all layers above the network are verified using Grove. Not shown are clerks for the coordinator and shard services, which are part of `KVClerk`.

Exactly-once RPCs. On top of uRPC, we implement eRPC, an exactly-once RPC library. To realize RPCs that execute exactly once, we use a typical reply-table protocol. Each client is given a unique client ID. The client combines its ID with increasing sequence numbers to associate each logical operation with a unique request ID. The server maintains a table of the latest request ID it has seen from each client. If a new request arrives, the server executes it and updates the table, noting the response for this request. If the server sees that request again, it replays the same response back to the client. For memory efficiency, this scheme only supports one outstanding request per client ID.

GroveKV. GroveKV consists of a single coordinator server, a variable number of shard servers, and a client library for interacting with the servers. Shards are sets of keys that can be migrated between servers; new servers can be added while the service is running.

The client library provides *clerk* objects, which maintain a bit of client-side state and implement procedures `Put`, `Get`, `MGet`, and `ConditionalPut`. Under the hood,

```
type KVClerk struct {
  coordCk  *CoordinatorClerk
  shardMap []*ShardClerk
}

func (ck *KVClerk) Get(key uint64) []byte {
  sid := shardOf(key)
  for {
    shardServer := ck.shardMap[sid]
    v, err := shardServer.Get(key)
    if err == EDontHaveShard {
      ck.shardMap = ck.coordCk.GetShardMap()
    } else {
      return v
    }
  }
}
```

Figure 2-2: Go pseudo-code for a simplified `Get` client implementation.

```
func (ck *KVClerk) GetTwo(k1, k2) (v1, v2) {
  done := make(chan bool)
  go func() { v1 = ck.Get(k1); done <- true }()
  go func() { v2 = ck.Get(k2); done <- true }()
  _ = <-done
  _ = <-done
}
```

Figure 2-3: Go pseudo-code for `GetTwo`, a simplified version of `MGet` that fetches two keys in parallel.

the client library uses the coordinator to determine which shard server the data is stored on (caching this mapping to avoid communicating with the coordinator unnecessarily), and uses exactly-once RPCs to ensure that each operation (such as `Put`) is executed exactly once. The GroveKV clerk accesses the coordinator server and shard servers through their own respective clerks. For example, the code for the `Get` operation in the GroveKV clerk, which invokes the `CoordinatorClerk` and the `ShardClerk`, is shown in Figure 2-2.

A clerk can also issue concurrent operations. For example, Figure 2-3 shows the clerk code for `GetTwo`, a simplified version of GroveKV's `MGet` which fetches any number of keys in parallel. The clerk will automatically obtain new client IDs to be

able to perform as many requests concurrently as the application requires.

Lock service. On top of the key-value service, we implemented a simple lock service with `Lock(k)` and `Unlock(k)` functions, for acquiring and releasing a lock named by key `k`. This lock service is a purely client-side abstraction—the backend system is the exact same as for the key-value interface. The lock service is a bit simplistic (it does not implement blocking acquire, and instead repeatedly tries to acquire the lock using `ConditionalPut`), which could be fixed by extending the GroveKV API.

Bank. To further evaluate the usability of client specifications in Grove, we implemented a simple bank application using the lock service and GroveKV. The bank has a `Transfer` procedure for atomically transferring between two accounts, which maybe stored at different shard servers. The bank uses GroveKV to store account balances and the lock service to synchronize access to accounts for `Transfer`. The bank also has an `Audit` procedure which checks that the total amount in the bank is unchanged. The goal is to verify that the `Audit` check never fails, providing an end-to-end proof that the specs of the components of the example system are meaningful.

Chapter 3

The Grove Verification Framework

This section describes the Grove framework that we built using Iris [12] to verify the code described in §2.

3.1 Overview

Figure 3-1 gives an overview of Grove's verification approach. The top of the diagram shows the shared state that Grove supports: a shared network and several memory heaps, one for each machine (shared among the threads of that machine). A Grove execution proceeds as a sequence of steps, whereby some thread from some machine executes an atomic instruction (accessing its heap or the shared network). An execution can arbitrarily interleave threads both within and across machines. Verifying a distributed system with Grove captures *all* possible behaviors of that system

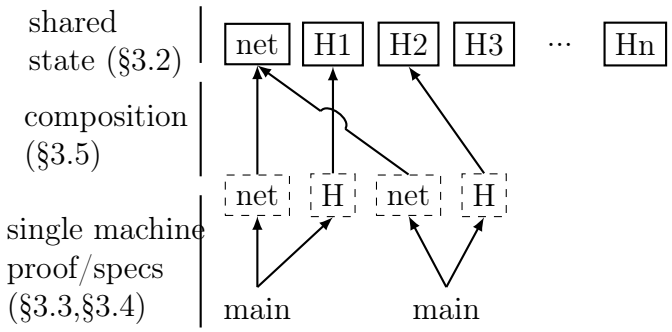

Figure 3-1: Overview of Grove's verification approach.

according to this execution model. We discuss this model in more depth in §3.2.

Formal reasoning in Grove requires using Hoare logic to establish specs for the `main` procedures running on each machine of the system. This means proving pre/-postcondition specifications of the form $\{P\}\, e\, \{Q\}$ ("Hoare triples") for procedures e. Grove provides the developer with Hoare triples for primitive operations, such as accessing shared memory in the heap (see §3.3), and sending and receiving messages on the network (see §3.4). The developer uses those specifications to build proofs of specifications for larger and larger procedures, using standard Hoare logic composition rules. Finally, the developer proves a single Hoare triple for the entire code running on a particular machine, which we call $\texttt{main}_j$. (There is a single `main` procedure per machine, but that procedure can dynamically spawn threads to achieve machine-local concurrency.)

The Hoare triples for procedures running on a single machine are parametrized by a heap ID, so they can be instantiated to reason about the same procedure running on different machines. This parametrization allows the developer to use Grove's composition theorem to put together the correctness theorems for multiple machines into a single theorem about the distributed execution of all of them (see §3.5). This composition theorem connects the Hoare logic statement about each machine's $\texttt{main}_j$ with the heap of the jth machine.

3.2 Execution model

To reason about distributed systems, we need to start with an accurate model of how these systems behave. The systems we are interested in consist of shared-memory concurrent procedures communicating over an asynchronous network, i.e., packets can be arbitrarily dropped, reordered, or duplicated.

As is common, Grove models distributed systems as state machines. The state of a distributed system consists of the shared state shown in Figure 3-1 (a global network and per-machine heaps), as well as the local state of each thread (the stack and program counter, so to speak—though we represent them more abstractly via

the remaining code that each thread will execute). Each thread is associated with one particular heap in the system (i.e. one thread cannot directly access the heaps of two different machines, as one would expect).

A step of the system non-deterministically picks any thread and executes a single instruction of that thread. This might update the thread-local state, update the heap associated with the thread (e.g. `store`ing to memory), and/or update the state of the network (e.g. `Send`ing a packet). Interleaving these steps across all threads across the entire system captures all the possible concurrency both within and across machines.

3.3 Distributed Hoare triples

When reasoning about the heap, non-distributed separation logics like Iris enjoy rules such as

$$\{p \mapsto v\}\ \texttt{*p}\ \{\textbf{ret}\ v,\ p \mapsto v\}$$

for reading memory. The **ret** clause says that the expression `*p` will evaluate to value v under the given precondition. That postcondition uses the heap *points-to assertion* $p \mapsto v$, which says that pointer p currently points to a value v. It further expresses *ownership* of said pointer, meaning that no other thread is currently able to access that location in memory.

In a distributed system, talking about "the" heap is ambiguous. Every machine has its own separate heap that only it can read or write to. A heap points-to $p \mapsto v$ in a distributed setting needs to clarify *on which heap* the pointer p lives.

To resolve this ambiguity in Grove, all heap points-to resources and Hoare triples are indexed by a *heap ID*. For instance, the Grove rule for reading from the heap is

$$\{p \mapsto_h v\}\ \texttt{*p}\ \{\textbf{ret}\ v,\ p \mapsto_h v\}_h$$

where the subscript h is a heap ID. Triples with subscript h are triples that are true only for code running against heap h. A heap points-to $p \mapsto_h v$ is only usable by code running on heap h. For a fixed heap, Grove inherits Hoare logic rules from Iris's

ability to reason about non-distributed concurrent procedures.

However, most of the time, the developer just wants to refer to "the heap of the current machine". Thus, Grove proofs and specifications typically leave the heap ID implicit. We write $\{P\}\, e\, \{Q\}$ to mean $\forall h, \{P_h\}\, e\, \{Q_h\}_h$. In other words, when proving a triple, a developer proves that it holds no matter what specific heap ID happens to be associated to the thread. This leads to the structure outlined in Figure 3-1, where each triple is parameterized by the heap it refers to.

An additional benefit of parameterizing triples by a heap ID this way is that Grove can re-use the Perennial [4, 6] framework libraries for reasoning about Go code with Goose [5]. Perennial already parameterized triples by a heap *generation number*, in order to reason about crashes and recovery. This approach lets us share proofs between single-machine Perennial and multi-machine Grove, greatly reducing the maintenance burden.

Although Grove's Hoare triples specify the execution of a procedure on a single machine, that procedure's execution can interact with other machines through the network. The Hoare-logic proof for a procedure can reason about the messages sent and received from other machines using Grove's network reasoning principles, which we describe next.

3.4 Reasoning about the network

Grove provides a simple network interface that allows sending and receiving messages, along the lines of UDP (but without limits on message sizes). Messages are always sent on a bi-directional connection between two *endpoints*. Grove models this network interface using the notion of *channels* between endpoints. To capture asynchrony, a channel represents a set of messages that have been sent; sending a message is modeled by adding the message to the channel, and receiving a message picks any message non-deterministically from the channel (or no message at all). In this model, an endpoint consists of two references to channels: one for incoming messages and one for outgoing messages. Figure 3-2 shows an example with one connection between

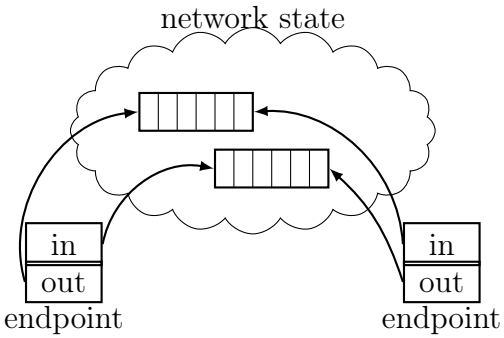

Figure 3-2: Network state representing a connection between two endpoints. The state consists of two channels, one for each direction of communication. An endpoint refers to two channels in the network state: one incoming and one outgoing.

$$\{\texttt{conn.out} \mapsto_{\text{chan}} M\}$$
$$\quad \texttt{conn.Send(msg)}$$
$$\{\texttt{conn.out} \mapsto_{\text{chan}} M \cup \{\texttt{msg}\}\}$$

$$\{\texttt{conn.in} \mapsto_{\text{chan}} M\}$$
$$\quad \texttt{conn.Recv(msg)}$$
$$\{\exists m \in M, \textbf{ret } m, \texttt{conn.in} \mapsto_{\text{chan}} M\}$$

Figure 3-3: The Hoare triples for `Send` and `Receive`. Here, `conn` is a network connection that refers to two channels: `conn.in` for incoming messages and `conn.out` for outgoing messages. M represents the set of messages in a particular channel.

two endpoints and their two associated channels. When many clients connect to the same server, they will all have individual channels for the server-to-client direction, but a single channel will collect all messages sent to that server from anyone.

To let developers reason about the network, Grove provides Hoare logic specifications for the network API. Figure 3-3 shows the specifications for the `Send` and `Receive` operations. The specifications are written in terms of the *channel points-to* assertion. This is similar to a heap points-to assertion, but talks about the contents of a channel in the network model. The channel points-to assertion $\texttt{conn.out} \mapsto_{\text{chan}} M$ represents exclusive knowledge that the only messages that have been sent on the `conn.out` channel are precisely those in the set M.

The spec for `Send` says that in order to send a packet, a thread must own the

channel points-to for the outgoing channel. The spec for `Receive` says that any previously sent packet can be received. Not shown is the case where `Receive` can time out and return nothing.

In Grove's specifications, sending and receiving a message requires ownership of the same channel: the sender must own the channel points-to when invoking `Send`, and the receiver must own the same exact channel points-to when invoking `Receive`. This reflects the fact that sender and receiver must coordinate to agree on the protocol spoken over this channel. The Iris separation logic allows such coordination on shared state via *invariants* [11]. The invariant might say that each message in the channel obeys a certain predicate; this will force the sender to prove that its message satisfies the predicate and will allow the receiver to assume the same. For example, the invariant for an RPC library might require that all messages are correctly marshalled RPC request.

3.5 Distributed Composition

Grove's Hoare triples capture how a procedure executes on a single machine (identified by its heap). These specifications enable local callers to reason about what happens if they invoke this procedure. However, Hoare triples are local: they do not directly reason about code running on other machines, or other threads on the same machine. As a result, it is possible to write conflicting specifications by making contradicting assumptions about shared state (i.e., by using different invariants). For instance, it is possible to write (and prove) Hoare triples for a key-value client and server that speak different versions of a network protocol. Although both can be individually proven, the client and server cannot talk to one another because they do not agree on a network protocol. This could happen, for instance, if the client and server serialize/deserialize requests in an incompatible way. As another example, a problem would arise if, say, the client depends on stronger semantics than what the server provides (e.g. the client expects linearizability, but the server is unsafe for concurrent accesses).

To prevent this, Grove requires developers to demonstrate the consistency of the specifications of individual machines when they are composed into a distributed system. Specifically, Grove provides a distributed composition theorem, shown in Figure 3-1, which takes as input a set of proven Hoare triples for the `main` functions running on each machine, and produces a theorem about the execution of the distributed system according to Grove's execution model (section 3.2).

However, using this theorem requires picking a `main` function for each machine in the system. What does this mean when a core verified component in GroveKV is a *library* for interacting with our verified distributed key-value service? That library does not have a `main` function that we could plug into the distributed composition theorem. Instead, it has functions like `Get` and `Put`. How can we know that the Hoare triples proven about these functions are meaningful?

The idea is that while we cannot plug those Hoare triples directly into the distributed composition theorem, we *can* write a simple test program that calls `Get` and `Put` and states assertions to check that everything behaves as expected. The developer first proves the correctness of their test procedure (as yet another Hoare triple), and then composes the test procedure with the Hoare triples for the rest of the system (including the key-value client library and the key-value server) using the distributed composition theorem. The result is a statement that the composed system will never fail any assertions. Although the composed system is a closed world that does not have any external inputs or outputs, the theorem establishes that the Hoare triples of `Get` and `Put` are meaningful. This is a standard approach in the verification community for stating the soundness of a program logic.

More precisely, the distributed composition theorem says:

> Suppose that the triples $\{P_1\}\,\texttt{main}_1\,\{\mathsf{True}\}$, …, $\{P_n\}\,\texttt{main}_n\,\{\mathsf{True}\}$ hold and that it is possible to establish $P_1 * P_2 * \cdots * P_n$ starting from a set of points-to relations for k initial network channels ($a_1 \mapsto_{\text{chan}} \varnothing, \ldots, a_k \mapsto_{\text{chan}} \varnothing$).
>
> Then a distributed system with n machines in which the jth machine

runs the procedures $\texttt{main}_j$ will always execute safely, meaning it will be memory- and thread-safe and no `assert` statements will fail.

Furthermore, all invariants I on the state of the network that are established in the proof will hold on all reachable states of the distributed system.

Note how the preconditions P_i of the individual machines are joined together using the *separating conjunction* $*$, which expresses that the initial resources handed to different machines must be *disjoint*: it is not possible to give two machines exclusive ownership of the same resource.

For our example from chapter 2, we wrote a top-level test program for the bank application that computes the total balances across all accounts and asserts that it adds up to the correct amount. We use the distributed composition theorem to prove that that assertion always holds, which demonstrates that all specs involved in the proof are consistent and meaningful.

Chapter 4

Verifying uRPC

As shown in Figure 2-1, the first (verified) component in our distributed system stack is uRPC, a concurrent, unreliable remote procedure call (RPC) library. uRPC has a server side, which executes functions as RPC requests arrive, and a client side, which sends RPC requests and matches up response packets with their originating client invocation. The server handles RPCs concurrently, and the client supports concurrent RPC invocations—hence the need to correctly match up response packets with invocations.

The library is unreliable in the sense that it does not try to cope with packet duplication or loss, so an RPC can be run many or zero times in response to a single client invocation. In fact, even if we were to use "reliable" transport protocol like TCP, it is possible for a connection to be lost in the middle of an RPC invocation, and we may still need to resend the request with a new connection. RPCs served over uRPC are meant to be safe to retry, so if the client gets no response, they can simply try again.

4.1 Formal specification of uRPC

Consider a `Put` function on a key-value server with specification

$$\{k \mapsto_{\text{kv}} w\}\ \texttt{Put(k, v)}\ \{k \mapsto_{\text{kv}} v\}^{1}.$$

Here, $k \mapsto_{\text{kv}} v$ is a points-to-like assertion to reason about the current value of key k. Suppose we want to serve `Put` on a uRPC server. An application can invoke this RPC as `urpcClient.Call(“Put”)` with uRPC's client library. What is a useful specification that we can prove for such an RPC invocation?

Naively, we might hope to prove the specification

$$\{k \mapsto_{\text{kv}} w\}\ \texttt{urpcClient.Call(“Put”)}\ \{k \mapsto_{\text{kv}} v\}.$$

In other words, we might hope that the triple for the function on the server yields an identical triple for the client. After all, the appeal of RPCs is that they feel like local function calls.

However, network unreliability means that this spec is wrong. To see why, suppose the client calls `Put` to set k to 0. If the packet is duplicated and delayed, the duplicated request to set k to 0 might arrive again at the server long after the original request completed. Setting k to 0 a second time is not the specified behavior of a `Put`. This issue is visualized in Figure 4-1. There is a difference between running `Put` locally and invoking it over uRPC, so we cannot expect them to have the exact same spec.

In fact, `Put` should not even be served directly over uRPC. Instead, the only functions it makes sense to directly serve over uRPC are ones that are safe to run many times. The precondition must let us run the server function arbitrarily many times.

Formally, in order for a function to be safe to serve over uRPC, we require that its precondition be *duplicable*. An assertion P is duplicable if P implies $P * P$. As shown

[1] One can imagine more generally, $\{P\}\ \texttt{f()}\ \{Q\}$; the `Put` spec is our representative example for this section.

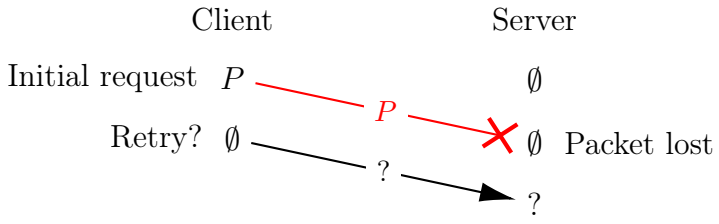

Figure 4-1: The problem with sending (exclusive) resources to the server for an RPC. Client first tries to send P along with its request message in order to run some $\{P\}\,\texttt{f}\,\{Q\}$ on the server. The client may resend the request. On the second request, the client no longer has the resources P to send and is stuck.

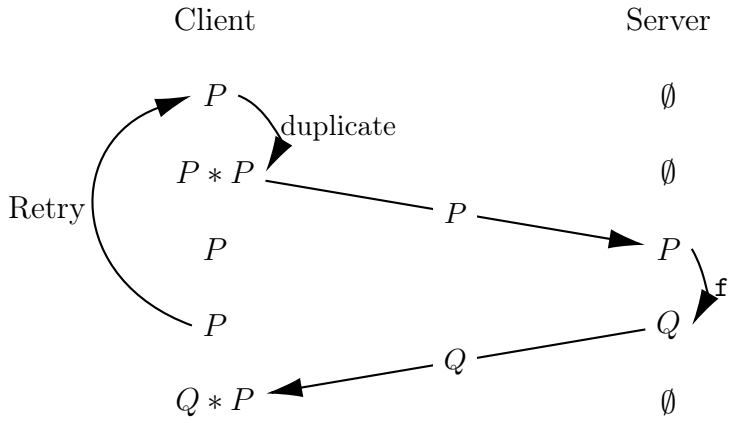

Figure 4-2: First, the client duplicates P into $P * P$, and the server the extra copy of P to safely run the RPC handler. If the client wants to retry (or the request is duplicated), there is always P sitting around from which a new copy can be created.

in Figure 4-2, if a client invokes an RPC with duplicable P, we can keep peeling off copies of P that make it safe to run the function as many times as demanded by packet duplication.

Now, we have arrived at a provable specification for uRPC:

> Let f be a function with spec $\{P\}\,f\,\{Q\}$. Suppose also the precondition P is duplicable. Then, a server can safely host f as an RPC using uRPC, and if a client connects to this server with a uRPC client `urpcClient`, then the following triple holds true:
>
> $$\{P\}\,\texttt{urpcClient.Call(“f”)}\,\{Q\}$$

This specification precisely captures the notion that an operation might be executed multiple times. For instance, the RPC in GroveKV's coordinator for getting the current server for each shard is safe to run directly over uRPC. The precondition is just True, reflecting that a client can always fetch the shard mapping. True is inherently duplicable so the RPC can be run as often as desired.

RPCs like `Put`, cannot be run them directly over uRPC, since that would require that the precondition $k \mapsto_{\text{kv}} v$ be duplicable, but $k \mapsto_{\text{kv}} v$ is not duplicable because it represents *ownership*. To achieve a client-side `Put` with the same spec as a local `Put`, we need to do some extra work both in the code and proof to ensure that `Put` runs *exactly* once. This will be the subject of §5.

4.2 Proving uRPC's specification

uRPC is a concurrent RPC library, and involves multiple threads and coordination via mutexes and condition variables. The proof of the top-level uRPC spec reasons about this node-local concurrency using Iris's standard Hoare-logic rules.

The proof also uses the reasoning principles for network primitives from Figure 3-3 and involves maintaining invariants about the messages in the network exchanged between the server and any clients. A key invariant about the network messages states that for any RPC request sent to the server, there is a copy of P available so the function for that RPC can always be run again.

uRPC is the only component that directly uses Grove's network interface. All higher layers use RPCs as their foundation for communication. So, from here on out, we can forget about the underlying network interface and its reasoning principles, and instead rely on the specification for uRPC.

One interesting feature about the uRPC spec is that the *post*condition need not be duplicable. For example, the exactly-once RPC library includes a `GetFreshCID` RPC served over uRPC which returns a client ID (CID) not used by anyone else. The postcondition of the `GetFreshCID` RPC is exclusive (hence non-duplicable) ownership of that client ID. If a particular reply message is lost, then the ownership of that

client ID is also lost. Even so, one reply message is never mistakenly used to fulfill two `GetFreshCID` requests, so two clients will never believe themselves to own the same client ID.

We achieve non-duplicable post-conditions using *escrow*, a proof technique that features prominently in our proof of exactly-once RPC and which is the topic of the next chapter. We will also briefly discuss in the next chapter how escrow is used in uRPC.

Chapter 5

Exactly-once RPCs and Escrows

We have seen how the uRPC library specifies unreliable function calls. Moving up the system layers in Figure 2-1, we now consider how to implement and verify the *exactly-once* RPC library (eRPC) on top of uRPC. On the implementation side, we follow a standard *reply table* approach, as already sketched in chapter 2. To illustrate how eRPC works, we discuss in this chapter a concrete instance of the eRPC code for `Put`, shown in Figure 5-2. In the real library, in place of the map update on line 5, eRPC takes a callback to allow the user to provide the function to be run for an eRPC invocation.

The challenge lies in verifying the desired specification for such an exactly-once RPC: we want to prove $\{k \mapsto_{\text{kv}} w\}$ `s.Put(k, v)` $\{k \mapsto_{\text{kv}} v\}$; i.e., this time we actually *will* obtain the same specification for the RPC as for a regular function call. But how is that possible, given that the underlying uRPC library requires a duplicable precondition? How can we send non-duplicable assertions such as $x \mapsto n$ to the server using a library that only allows sending duplicable assertions? The answer lies in the *escrow pattern*.

5.1 The escrow pattern

Originally developed for the weak-memory concurrent separation logic GPS [31] based on earlier work in verified cryptography [2], the intuitive idea of the escrow pattern is

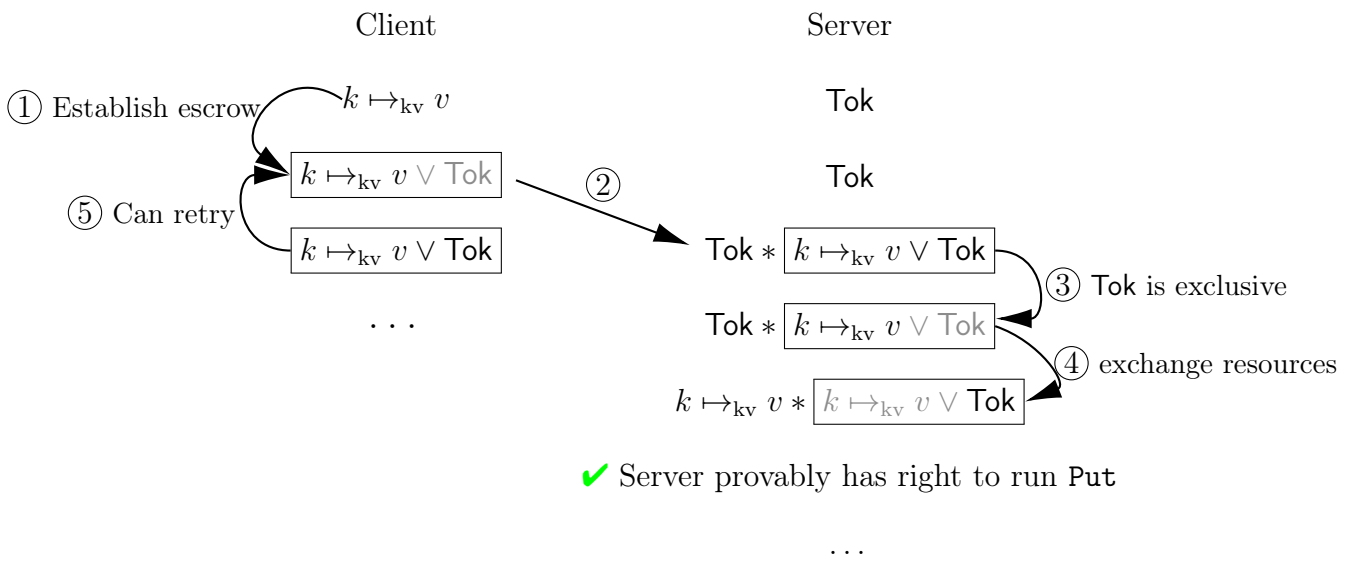

Figure 5-1: To send $k \mapsto_{\text{kv}} v$ for a `Put`, the client puts it in an escrow invariant, and sends knowledge of the invariant to the server.

to *indirectly* transfer ownership of some non-duplicable assertions by "depositing" the assertion in a "known location" (the *escrow*), and then only transferring (duplicable) *knowledge* that the deposit has happened. For this to work, the other party needs to have the sole right to take things out of the escrow. To illustrate the escrow pattern, consider a machine that wants to transfer ownership of $k \mapsto_{\text{kv}} v$ to another machine while only attaching duplicable assertions to messages to account for message duplication.

We can accomplish this by using invariants and an *escrow token* Tok. This token is an exclusive assertion: one with the property $\text{Tok} * \text{Tok} \implies \text{False}$. The escrow token represents the right for the server to get the client's assertions. The client and server need to agree in advance on which token is supposed to be used for a given request. As we will see in the next section, exactly-once RPCs will use one token for each client ID and sequence number, which are all initially owned by the server.

The escrow proof pattern is shown in Figure 5-1. The client starts with ownership of $k \mapsto_{\text{kv}} v$ and the server with Tok. To transfer the points-to, the client establishes the escrow invariant $E_{\text{inv}} = \boxed{k \mapsto_{\text{kv}} v \lor \text{Tok}}$ by enclosing its points-to in the invariant

as the left disjunct (step ① in the Figure). At this point, we say that $k \mapsto_{\text{kv}} v$ is *under escrow*. The client can now send messages to the server informing it that E_{inv} holds (step ②). The *knowledge* that $k \mapsto_{\text{kv}} v$ is under escrow is duplicable, so duplicating those messages is not a problem.

When the server receives this message, it knows E_{inv} holds: $k \mapsto_{\text{kv}} v$ has been put under escrow, and is ready for the taking. Crucially, the server owns the exlusive Tok, which serves as a proof that $k \mapsto_{\text{kv}} v$ not been taken yet (step ③). Formally, the server opens up the invariant E_{inv}, which can be in two cases. In the first case, the invariant contains $k \mapsto_{\text{kv}} v$, and the server simply keeps it. To re-establish the invariant, the server proves $k \mapsto_{\text{kv}} v \lor \mathsf{Tok}$ by picking the right disjunct and giving up ownership of the token (step ④). The second case, in which the invariant does not contain $k \mapsto_{\text{kv}} v$ but contains Tok, is impossible: the token cannot be both owned by the server and stored in the escrow because $\mathsf{Tok} * \mathsf{Tok} \implies \mathsf{False}$.

The end-to-end effect of this exchange is that the server started with ownership of Tok and knowledge of E_{inv}, and ended up with ownership of $k \mapsto_{\text{kv}} v$, while only duplicable knowledge was transferred from client to server. If the client wants, it can retry (step ⑤) by resending knowledge of E_{inv}.

5.2 Verifying exactly-once RPCs

We now apply this general escrow pattern to verify exactly-once semantics of reliable RPCs implemented on top of uRPC. Figure 5-2 shows such an implementation of the `Put` RPC.

The implementation of exactly-once RPCs works as follows: each request has a unique identifier by combining a client ID (`CID`) and an increasing sequence number (`Seq`); we call this pair the *request ID*. The first time that `PutRPC` is run on the server for a particular request ID, the server can be sure that the operation has not been run before, so it can do so *exactly once*. Before replying to the caller, it records the fact that the operation has now been run in `lastSeq`. If the same request ID is seen by the server again, the server can safely do nothing and repeat the old reply to the

```
func (s *KVServer) PutRPC(args *PutArgs) {
  s.mu.Lock()
  if s.lastSeq[args.CID] >= args.Seq {
  } else {
    s.kvs[args.Key] = args.Value
    s.lastSeq[args.CID] = args.Seq
  }
  s.mu.Unlock()
}

func (c *KVClerk) Put(key uint64, val []byte) {
  args := PutArgs{CID:c.cid, Seq:c.seq,
    Key:key, Value:val}
  c.seq = c.seq + 1
  for {
    err := ck.urpcClient.Call("PutRPC", &args)
    if !err {
      break
    }
  }
}
```

Figure 5-2: Simplified fragment of a key-value service built on top of uRPC.

client. (If an even older sequence number is seen, we know the reply is irrelevant since clients never have more than one outstanding request.) `Put` does not return anything, which simplifies the example since no reply needs to be cached.

In Grove, we reason about the `Put` using escrow. For every single request ID that it has not been seen, the server owns an escrow token $\mathsf{STok}_{CID,Seq}$. The clerk starts an exactly-once RPC by putting the assertions needed to execute the operation, in this case $k \mapsto_{\text{kv}} w$, under escrow:

$$R_{\text{inv}} = \boxed{k \mapsto_{\text{kv}} w \lor \mathsf{STok}_{CID,Seq}}$$

The client passes knowledge of this escrow invariant to the server when it invokes `PutRPC`. There are two cases for invocations of `PutRPC` on the server.

Fresh request. The first case is that the request ID passed into `PutRPC` has not been seen before (the `if` branch in `PutRPC`). In this case, the server owns $\mathsf{STok}_{CID,Seq}$.

The server can exchange this token for the points-to $k \mapsto_{\text{kv}} w$ that is under escrow. With the points-to in hand, the server can physically update its key-value map and end up with $k \mapsto_{\text{kv}} v$, while maintaining the invariant that the physical key-value map is coherent with the $k \mapsto_{\text{kv}} v$ assertion

To finish the request, the server needs to send $k \mapsto_{\text{kv}} v$ back to the client together with the reply. The reply could be duplicated like the request, so we again use the escrow pattern for this ownership transfer. Similar to the server, the client also owns a client-side escrow token $\mathsf{CTok}_{CID,Seq}$ for each sequence number associated with its client ID.

The server puts the updated points-to under escrow:

$$S_{\text{inv}} = \boxed{k \mapsto_{\text{kv}} v \vee \mathsf{CTok}_{CID,Seq}}$$

Before replying to the client, the server updates its `lastSeq` table to record the last sequence number that it has seen. We already mentioned that the server owns $\mathsf{STok}_{CID,Seq}$ for all requests it has not yet seen; moreover the server has knowledge of an escrow like S_{inv} for the last request that might still be outstanding. This invariant holds again once `lastSeq` has been updated. Finally, the server sends back knowledge of S_{inv} to the client.

Duplicate request. The second case is that the request ID passed into `PutRPC` has been seen before. If this is the last request, the server will already have a copy of S_{inv} saved from the first time that the request was seen. The server does not have to do anything, it just sends another copy of S_{inv} (representing knowledge of the fact that the request was executed) to the client. The proof has to also cover the case in which this request is even older; in that case we can show that the response is irrelevant since the client will only ever care about one pending request.

Client-side reasoning. We now briefly consider what happens on the client-side in `Put`. The first two lines bundle up the arguments for the request as well as the client ID and sequence number, which is incremented each time. The client has ownership

of each client-side token $\mathsf{CTok}_{CID,Seq}$ that it has not used yet for sequence numbers `c.seq` and higher.

Next, the client sets up the request escrow R_{inv} by putting $k \mapsto_{\text{kv}} w$ into the invariant. This is the precondition required by the uRPC. It is duplicable, so we can repeat this request arbitrarily often in a loop, until we get a response.

If the unreliable RPC ever gets a response (after enough retries), that will convey knowledge of S_{inv} from the server. Since we own the client-side token corresponding to this request, we can open the escrow and take out the points-to fact $k \mapsto_{\text{kv}} v$ that the server sent back, which is exactly the postcondition we need to return from `Put`.

5.3 Escrow in uRPC

We mentioned at the end of chapter 4 that we also use escrow to achieve a non-duplicable postcondition for uRPC. Now that we have talked about escrow, we can briefly describe how. First, note that reply messages in uRPC are matched up with their invocation by associating each request with a uRPC request ID. When the server responds to an RPC with a given request ID, the response message includes this request ID so the client can correctly match up the response with the invocation.

Recall that a message (whether it happens to be considered a request or reply message by higher-level code) can only carry duplicable assertions with it for the receiver to learn about when they receive it. The assertion associated with the reply message for uRPC is actually an escrow invariant $\boxed{Q \lor \mathsf{Tok}_{ID}}$, where Q is the real postcondition of that RPC. Tok_{ID} is a token owned by the client library for the request with ID as its ID. When the uRPC client library receives a reply for an unfinished RPC, it will still own Tok_{ID}, and can use it to get Q from the escrow invariant. Then, it returns Q to the caller of the RPC and is done.

Chapter 6

Verifying distributed systems with Grove

This section outlines the verification of GroveKV, the sharded key-value server described in chapter 2. By using the exactly-once RPC specification described in the previous section, much of the verification is similar to standard single-node reasoning in concurrent separation logic (CSL). In particular, verifying each GroveKV operation is split into verifying a server-side procedure, and then verifying a client-side clerk that manages local state and calls the RPC. Finally, we use the specification for GroveKV to verify a lock service and a simple bank client.

6.1 GroveKV specification

GroveKV's top-level library is the `KVClerk`, which provides the user with a interface to the key-value service that hides concerns about concurrency and network unreliability. Figure 6-1 shows its specification. The angle-bracket notation $\langle P \rangle\ f\ \langle Q \rangle$ will be explained shortly, but can be initially thought of as the same as $\{P\}\ f\ \{Q\}$ (in fact, the angle-bracket version is a stronger spec).

The `KVClerk` specifications uses a *key-value points-to*, $k \mapsto_{\text{kv}} w$ which represents exclusive ownership of key k and knowledge that its current value is w, much like the usual in-memory points-tos for heaps. The specification for `Put` says that if one

$$\langle k \mapsto_{\text{kv}} w \rangle \; \texttt{kvck.Put}(k, v) \; \langle k \mapsto_{\text{kv}} v \rangle$$

$$\langle k \mapsto_{\text{kv}} v \rangle \; \texttt{kvck.Get}(k) \; \langle \textbf{ret } v,\; k \mapsto_{\text{kv}} v \rangle$$

$$\langle k_1 \mapsto_{\text{kv}} v_1 * k_2 \mapsto_{\text{kv}} v_2 \rangle$$
$$\quad \texttt{kvck.GetTwo}(k_1, k_2)$$
$$\langle \textbf{ret } (v_1, v_2),\; k_1 \mapsto_{\text{kv}} v_1 * k_2 \mapsto_{\text{kv}} v_2 \rangle$$

$$\langle k \mapsto_{\text{kv}} v_{old} \rangle$$
$$\quad \texttt{kvck.ConditionalPut}(k, v_{exp}, v)$$
$$\left\langle \begin{array}{r} \exists b.\, \textbf{ret } b,\; (v_{old} = v_{exp} \wedge b) * k \mapsto_{\text{kv}} v \vee \\ (v_{old} \neq v_{exp} \wedge \neg b) * k \mapsto_{\text{kv}} v_{old} \end{array} \right\rangle$$

Figure 6-1: Specification for the KV service. Here, `kvck` is a `KVClerk` to the key-value service,

calls `Put`(k, v) starting with ownership of $k \mapsto_{\text{kv}} w$, then $k \mapsto_{\text{kv}} v$ is returned in the postcondition. The specification for `Get` says that the value returned matches the points-to $k \mapsto_{\text{kv}} v$ in the precondition.

`GetTwo` issues two `Get` RPCs in parallel (Figure 2-3). The specification for `GetTwo` hides this internal parallelism, and just says that the operation returns the value of two keys, as though doing a `Get` for each of them. GroveKV provides a more general `MGet` and corresponding specification for fetching arbitrarily many keys concurrently.

The specification for `ConditionalPut` is a slight modification of the `Put` spec, in which the target key is only updated if the previous value is the desired one.

The specifications for these operations are essentially the same as might typically be given to an in-memory hashmap in a concurrent separation logic like Iris. Hence, standard Hoare logic reasoning that would apply to such an in-memory hashmap works just the same for the distributed key-value service.

6.1.1 Linearizability and logical atomicity

GroveKV is a linearizable key-value service, and the specifications capture this using *logical atomicity* [13]. A logically atomic spec is written $\langle P \rangle\; f \;\langle Q \rangle$, and implies the normal spec $\{P\}\; f \;\{Q\}$.

Before we explain what logical atomicity is, note that the normal (non-logically atomic) specification is insufficient, for instance, to verify our lock service (section 6.4). If the `ConditionalPut` spec required $k \mapsto_{\text{kv}} v_{old}$ in a normal Hoare triple, then while one `ConditionalPut` runs, no other function would be able to access key k. However, the whole point of the lock service is that many clients race to do a `ConditionalPut` on the same key.

To make it possible for clients to concurrently access a key in a linearizable way, a logically atomic specification ensures that the transition from the precondition P to the postcondition Q happens *atomically* at the linearization point. To achieve this, a logically atomic spec is given a special assertion called the *atomic update*. An atomic update from P to Q gives the user of the update access to the resources P, but requires that the user immediately return the resources Q in at most one physical step. This ensures that there is no point in time at which P is violated but Q is not yet true.

A logically atomic spec enables concurrent accesses becausee it allows the user to open an invariant around a logically atomic function call. An invariant must remain true after every single physical step, so they can normally only be opened (i.e. temporarily broken) around a physically atomic step (such as a single memory read or write). A logically atomic function allows one to use an invariant across a function call that consists of many physical steps and is thus not physically atomic. It does this by only actually opening the invariant at the moment that the atomic update goes from P to Q.

So, for instance, ownership of a $k \mapsto_{\text{kv}} v$ can be placed in an invariant, the invariant shared with many threads, and the threads can all concurrently run `ConditionalPut` on key k.

With Grove, to prove logically atomic specs for RPCs, we pass the atomic update via the exactly-once RPC proof library to the server, and then pass the post-condition the caller. Putting the atomic update into the escrow invariants used by exactly-once RPC is straightforward because Iris (on top of which Perennial and Grove are built) is a *higher-order* logic [12].

6.2 Verifying `Put`, `Get`, etc.

GroveKV is sharded, which means that different shards (which are simply sets of keys) can be managed by different servers. Internally, shard servers in GroveKV provide their own client interface called `ShardClerk`. That interface includes functions similar to the top-level `KVClerk`, except that `ShardClerk` operations only operate on a particular server, so they can fail if the server does not own the requested key. For instance, the specification for the `ShardClerk` `Put` says:

$$\langle k \mapsto_{\text{kv}} w \rangle$$
$$\texttt{err} := \texttt{shardck.Put}(k, v)$$
$$\langle (\texttt{err} = \texttt{nil} * k \mapsto_{\text{kv}} w) \lor (\texttt{err} \neq \texttt{nil} * k \mapsto_{\text{kv}} v) \rangle$$

This means that either the `Put` will succeed and the key-value points-to will have the new value, or the server will return an error indicating that it does not own the shard for that key, in which case the key-value points-to is unchanged.

Proving this specification is straightforward with eRPC. We simply prove a local spec for the `Put` handler on the `ShardServer`, and then by plugging that into the eRPC library, we get the same spec on `ShardClerk`.

Clerks for multiple shard servers. The `KVClerk` uses the specs for `ShardClerk` in a loop: the `KVClerk` tries doing an operation against the server it believes owns the key, and consults the coordinator to try a different server if that operation returns an error. The code for this is shown in Figure 6-2. Shard lookup requests are sent to the coordinator as unreliable RPCs, because the coordinator merely returns a hint about where the shard might be, so no ownership transfer needs to take place. The postcondition for a shard lookup is a duplicable fact saying that the returned server is a valid shard server that can be queried. Clients thus do not have to know upfront about all shard servers, they can discover servers dynamically from the coordinator.

Each `ShardClerk` can only handle a single request at a time, because it is associated with one eRPC client ID, which allows for one outstanding operation. To

```
func (ck *KVClerk) Put(key uint64, value []byte) {
  for {
    shardID := shardOf(key)
    shardServer := ck.shardMap[shardID] // this server should hold 'key'
    shardClerk := ck.shardClerks.GetClerk(shardServer)

    err := shardClerk.Put(key, value)

    if err == ENone {
      break
    }
    // if error, update the shard mapping and try again
    ck.shardMap = coordinator.GetShardMap()
  }
}
```

Figure 6-2: Top-level `Put` operation using `ShardClerk`

support running multiple requests concurrently, `KVClerk` will create new `ShardClerk`s as needed, by requesting more unused client IDs.

6.3 Verifying shard migration

Besides distributing the current shard mapping, the coordinator is also responsible for dynamically migrating shards.

To facilitate shard migration, shard servers have a `MoveShard` and `InstallShard` operation. The coordinator calls `MoveShard(shardID, dst)` on a shard server to tell it to move one of its shards to another server `dst`. The source shard server will then call `InstallShard(shardID, kvData)` on the destination server to send it the latest key-value data for the specified shard. The internal specification for `InstallShard` is proved just like `Put`; instead of updating a key-value entry, we simply add a full shard to a server. Also just like the `Put`, proving the spec for `InstallShard` is straightforward with eRPC. We simply prove a normal local specification for `InstallShard` on the server, and then get the same spec through eRPC for the client.

In this protocol, at most one server thinks that it owns a shard at a time (possibly none if it is being moved). So, in the proof, each server maintains as an invariant

that all of its key-value data must match the current key-value points-tos. When a server responds to a request, it can be certain that it has the latest value for a key.

6.4 Verifying the lock service

The lock service is implemented as a client-side abstraction on top of GroveKV. The specification for the lock service is a standard (non-distributed) CSL lock specification. The assertion `isLock`(k, P) says that the lock k is associated with the lock invariant assertion P. The specification for `Lock` gives ownership of the lock invariant, $\{$`isLock`$(k, P)\}$ `Lock(k)` $\{P\}$. Conversely, releasing the lock requires giving up ownership of the invariant, $\{$`isLock`$(k, P) * P\}$ `Lock(k)` $\{$`True`$\}$.

The `Lock` function is implemented as a spinning `ConditionalPut`, and `Unlock` does a `Put`. Because the GroveKV specification from Figure 6-1 is similar to standard CSL specifications for in-memory operations, the lock service proof is essentially identical to a spinlock proof in CSL. This makes use of the logically atomic specification for `ConditionalPut`, which says that the operation is linearizable.

6.5 Verifying the bank

Figure 6-3 shows code from the simple bank client. The bank uses GroveKV to store values for two accounts, `a1` and `a2`. These keys are protected by locks in the lock service. The `TransferOne()` operation uses these locks to atomically move a value of 1 from one account to the other. (The initial account balances sum to less than `INT_MAX`, so there is no possibility of overflow.)

The `Audit()` routine asserts that the sum of the account balances is always equal to `TOTAL`. That is, this checks that the transfers between the accounts are transactional. We prove Hoare logic specs for each of `Audit` and `TransferOne`. The distributed composition theorem then implies that a machine running `Audit` will never fail the assert.

To establish the specifications, we instantiate the lock service specification so that

```
type Bank struct {
  lck  *LockClerk
  kv *KVClerk
}

func (b *Bank) TransferOne() {
  b.lck.Lock(a1); b.lck.Lock(a2)
  old_amount := b.kv.Get(a1)
  if old_amount > 0 {
    b.kv.Put(a1, old_amount-1)
    b.kv.Put(a2, b.kv.Get(a2)+1)
  }
  b.lck.Unlock(a1); b.lck.Unlock(a2)
}

func (b *Bank) Audit() {
  b.lck.Lock(a1); b.lck.Lock(a2)
  assert(b.kv.Get(a1) + b.kv.Get(a2) == TOTAL)
  b.lck.Unlock(a1); b.lck.Unlock(a2)
}
```

Figure 6-3: Bank built using GroveKV

the lock invariants guard ownership of the account keys. We add an invariant that says that when locks for the bank's accounts are not held, the sum of their balances is maintained. The proof of the bank then proceeds as a typical non-distributed Iris proof [3], using the GroveKV and lock service specs in place of standard heap and local lock specs.

Chapter 7

Implementation

Grove is implemented as an extension to Perennial [6]. The Perennial support for crash safety was irrelevant for this work, but using Perennial we could build on the existing integration of Goose [4, 5] and Iris [11, 12] for reasoning about Go code in Coq using the Iris Proof Mode [15, 16].

We extended Perennial with a distributed Hoare triple, a network model and corresponding specification, and the distributed composition theorem. On top of this, we built verified unreliable and exactly-once RPC libraries, the key-value service GroveKV, the lock service, and the bank example application. The line counts for the Go components and their proofs are given in Figure 7-1. The Go implementation of the network layer is trusted to match the specification in Figure 3-3; it is implemented using TCP which facilitates sending arbitrarily-sized messages.

	Lines of Code	**Lines of Proof**
Network layer	120	Trusted
uRPC	149	1,065
eRPC	63	819
GroveKV	807	5,555
Lock service	20	138
Bank	74	429

Figure 7-1: Lines of code and proof for Go components.

Chapter 8

Performance evaluation

In this section, we answer the following questions: (1) Does GroveKV have reasonable performance compared to a real key-value service? (2) Does it scale up with more cores? (3) Does GroveKV scale up as servers are dynamically added?

8.1 Single shard server performance

To evaluate whether GroveKV has reasonable performance compared to real systems, we compare against the popular key-value store Redis [27]. Since Redis is single-threaded, we compare the performance of a single GroveKV shard server running on one core against the performance of a Redis server.

Our experiments comparing Redis and GroveKV run on two machines, one server and one client. Each machine has an Intel Xeon CPU E5-1410 2.8 GHz processor and 64 GB RAM. The machines are connected over a 1 Gbps network and run Ubuntu 20.04.2 LTS.

We generate client requests using YCSB [7]. For this experiment, we use uniform random key value operations with 128 bytes for each value, 8 bytes for each key, and 100,000 total key-value pairs. Each client thread issues operations sequentially in a proportion of 95% gets and 5% puts.

We increase the number of client threads, recording the latency and throughput seen by the client program, up until we saturate the system—i.e. until throughput

stops increasing. Figure 8-1 plots this latency/throughput graph for both Redis and GroveKV, with each point representing a fixed number of client threads.

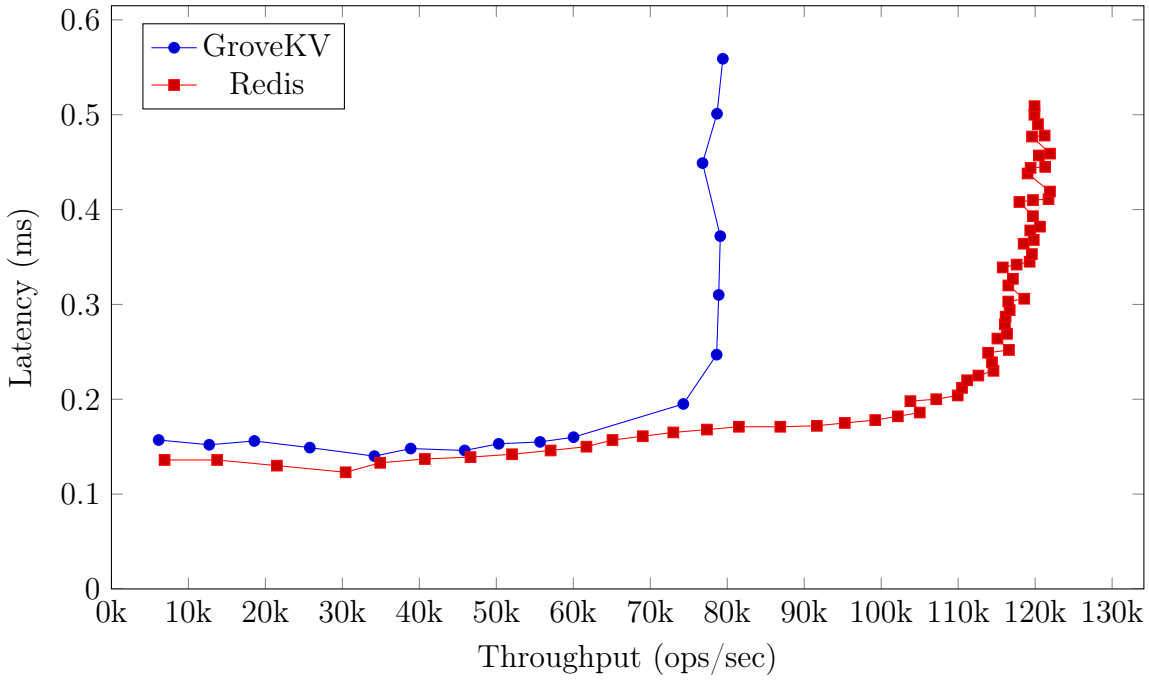

Figure 8-1: Latency/throughput comparison of GroveKV on a single core vs Redis, as we increase the client load.

The latency of GroveKV is close to the latency of Redis at low loads. Moreover, the peak throughput of GroveKV is about 2/3 that of Redis. At high load, however, GroveKV has worse latency than Redis. Through performance profiling, we observed that when saturated with requests, the GroveKV server spends the majority of its time in the kernel and the Go networking code. It is possible that better use of the Go networking APIs could decrease the amount of time spent in networking code and improve GroveKV's performance. Redis is written in C and directly uses the Linux networking API, which is likely contributing to its better performance.

8.2 Speedup from node-local concurrency

In this section, we evaluate how GroveKV's performance scales as more cores are added to a single shard server. For each number of cores, we keep increasing the

number of client threads in the benchmark program until throughput deteriorates, and measure the highest achieved throughput.

We run this experiment on a machine with 8 Intel E7-8870 2.4 GHz CPUs (each with 10 cores) and 256 GB RAM, running Arch Linux with kernel 5.10.64. The shard server runs with more and more cores on a single CPU. Our client benchmark runs on 40 cores on other CPUs.

Figure 8-2 shows the peak throughput as a function of the number of cores. Importantly, as we increase the number of cores, throughput increases almost linearly.

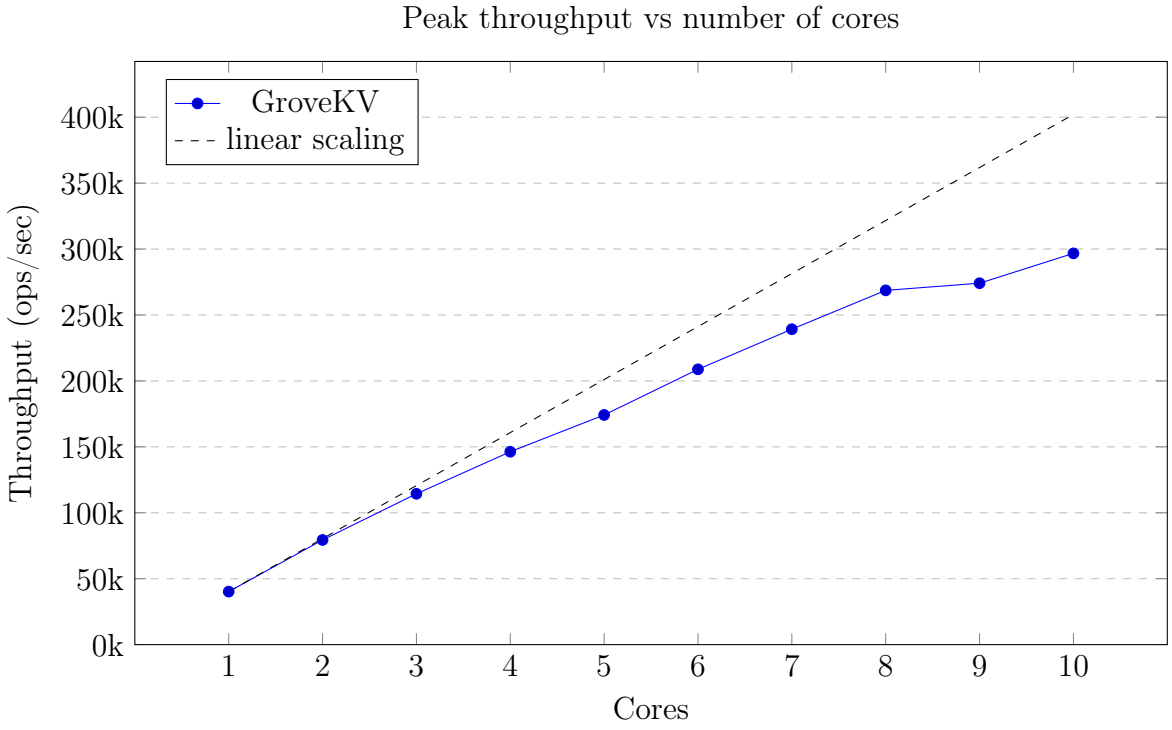

Figure 8-2: The horizontal axis shows the number of cores given to a single GroveKV shard server, and the vertical axis shows the peak throughput achieved at that configuration. The dashed line shows the hypothetical linear scaling with more cores by extrapolating the performance with a single core.

Like before, a majority of the time is spent in kernel and Go networking code. Moreover, as we increase the number of cores, an increasing amount of time is spent in the Go runtime (scheduler + garbage collection). Better use of Go and the kernel's networking interfaces could likely improve the multi-core scalability of GroveKV.

8.3 Speedup from dynamically adding servers

We additionally run an experiment in which we add servers to a running GroveKV system and measure the throughput over time. We run this experiment on the same machine as section 8.2. All servers run on the same machine. Each server uses 1 core on its own CPU; the other cores of that CPU are idle. Again, the client uses 40 cores on other CPUs.

We start with only one server in the system. After approximately 30 seconds, we add another server, invoking GroveKV's coordinator to dynamically migrate shards to balance load. We repeat this twice more until there are a total of 4 servers in the system.

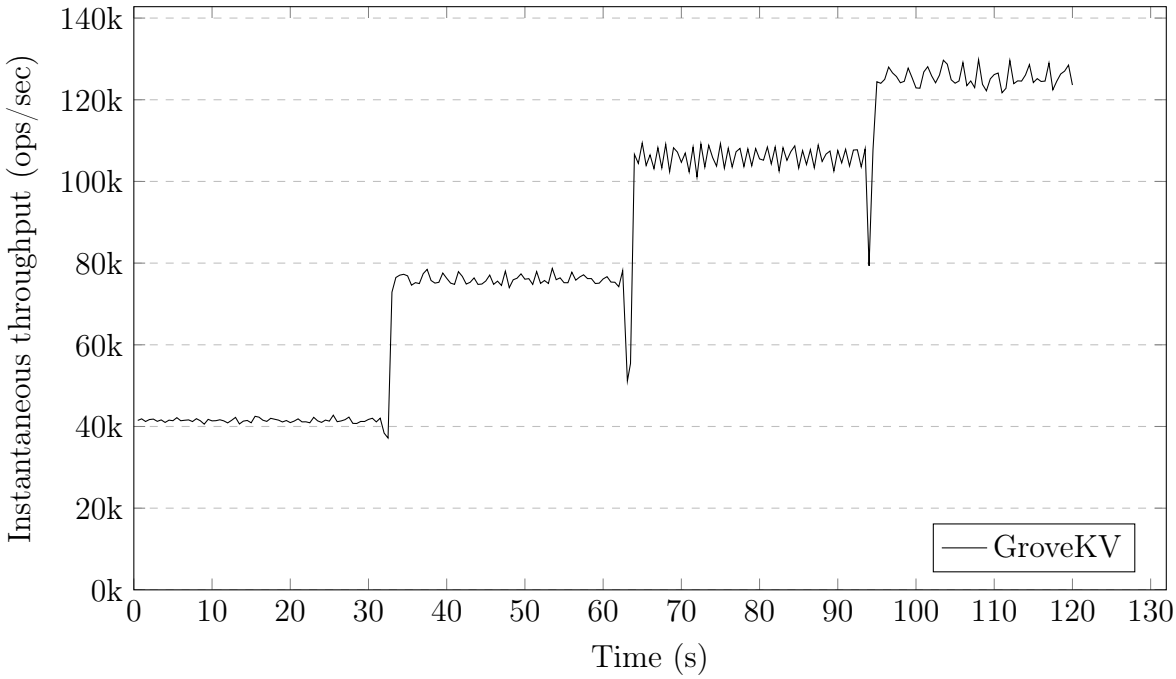

Figure 8-3: Throughput over time, servers added approximately every 30 seconds.

Figure 8-3 shows the instantaneous throughput (measured in 0.5 second time slices) of the system over time. There is a brief drop in throughput when shard migration occurs to rebalance the system, after which throughput increases based on the number of available servers. In reality, the throughput drops all the way to 0 briefly, while locks are held, but because migration takes less than 0.5 seconds, the

average throughput over the 0.5 second time slices nevers hits 0.

Chapter 9

Related work

Separation logic. Grove is not the first application of separation logic to distributed systems: the same idea is pursued by Disel [29] and Aneris [8, 17].

Disel [29] uses a notion of "protocols" to capture a logical, abstract description of the network messages exchanged between a service and its client. Reasoning about protocols is based on inductive invariants. Disel demonstrates how these protocols can be composed to verify clients that interact with multiple protocols, and to implement one service in terms of another ("delegation"). However, they use protocols as module boundaries for the purpose of abstraction; Disel does not have a notion of a client library or clerks, which is a key goal for Grove. Protocols also expose the node-local protocol-specific state (such as a set of outstanding requests). The Disel network model does not account for having to re-send a request in case of a connection failure, which means that connection failures lead to client stalls. Grove explicitly deals with re-sending requests, which required the escrow pattern.

Aneris [17] demonstrates compositional reasoning about components of a distributed system in an Iris-based concurrent separation logic. For example, Aneris is used to verify a general load balancer that is agnostic about the service provided by the backend servers. The Aneris network model is more low-level than Grove's, directly exposing the concepts of "IP addresses" and "ports". In follow-up work [8], Aneris has been used to verify a distributed causal database. This included the use of the escrow pattern to transfer resources between machines on top of the causal

consistency specs (see §4.5 of [8]). This use of escrow is similar to the use of escrow for ownership transfer in weak memory logics [31]. In contrast to our verified exactly-once RPC library, Aneris only proves causal consistency and hence does not establish that an RPC is executed at most once (and indeed, the implementation will re-execute RPCs when network packets are duplicated).

Whole-system verification. One approach to verifying distributed systems is to prove a sequence of refinements from a high-level protocol description down to executable code. IronFleet [10] used this approach by verifying protocols in TLA$^+$-like style encoded in Dafny [21], and then proving event handlers simulate transitions of the protocol. The Verdi [32] framework provides verified system transformers that convert systems verified on top of an idealized, reliable network into one that operates on an unreliable network.

However, these frameworks do not address concurrency within a node. Concurrency is difficult to handle, because the tools in existing frameworks for reasoning about or generating implementation code are restricted to sequential code. Additionally, concurrency complicates the techniques needed for refinements. For example, IronFleet uses reduction [23] to reason about event handlers as if they executed atomically as transitions of the protocol. This exploits the fact that nodes cannot observe intermediate states of event handlers on other nodes. However, node-local concurrency allows intermediate states to be exposed to other threads.

Furthermore, these refinement-based verification approaches specify correctness in terms of how the full system state evolves. In contrast, Grove's specifications for client libraries allow developers to verify application code using a local view of the part of the state that is relevant to them.

Modular verification. In order to verify large distributed systems composed of multiple components, a verification framework must support combining proofs of parts together.

The Chapar system [22] introduces a new operational semantics for specifying the

behavior of causally consistent key-value stores. Lesani et al. use this semantics as an interface for verifying the key-value store and the client applications separately. However, the framework provides only support for this form of composition, and does not provide modularity within the implementation of a key-value store itself.

WormSpace [30] proposes write-once registers (WOR) as a building block abstraction for building verified distributed systems. A verified implementation of WOR is provided using the Certified Concurrent Abstraction Layers (CCAL) approach of Gu et al. [9], which allows layers of verified components to be linked together through contextual refinement specifications. This compositionality is applied to use the WOR implementation to build verified implementations of Paxos [19], an append-only distributed log, and two-phase commit. However, the CCAL verification framework does not provide "client-facing" specifications for clerks or RPCs.

Protocol verification. TLA$^+$ [18, 20] provides a modeling language for concisely describing distributed protocols, which can then either 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. For example, ByMC [14] can automatically verify protocols that can be expressed as threshold automata. The Ivy [25] tool can represent a wider class of protocols, but still achieves semi-automatic verification by requiring inductive invariant specifications to be expressed in a decidable fragment of first-order logic [26]. I/O automata [24] support composition of modules by representing whole modules as a single I/O automaton.

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.

Protocol verification approaches also verify the system as a whole, proving that the overall system has a certain behavior. In contrast, Grove focuses on individual machines interacting with the distributed system, proving "client-facing" specifications for clerks and RPCs.

Chapter 10

Conclusion

Grove is a new framework for verifying distributed systems. The core idea of Grove is its focus on modular verification: stating and proving specifications about individual components and then using those specifications to verify components built on top. To enable this modular reasoning, Grove supports use of ownership for reasoning about coordination in a distributed system. We use duplicable ownership to specify unreliable RPCs, and verify exactly-once RPC using the escrow pattern for proving ownership transfer over unreliable networks. To demonstrate the value of these ideas, we have verified GroveKV, a performant key-value service written in Go that uses unreliable and exactly-once RPCs and supports dynamically adding new servers and rebalancing shards. Using this service we also implemented and verified a lock service and a bank application. We hope that the specification and proof techniques of this thesis will be useful to others who want to verify distributed systems in a modular fashion.

Bibliography

[1] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual conference, October 2021.

[2] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Logical foundations of secure resource management in protocol implementations. In *POST 2013*, volume 7796 of *Lecture Notes in Computer Science*, pages 105–125. Springer, 2013. doi: 10.1007/978-3-642-36830-1_6.

[3] Tej Chajed. A brief introduction to Iris. `https://plv.csail.mit.edu/blog/iris-intro.html`, 2020.

[4] 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.

[5] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent Go code in Coq with Goose. In *Proceedings of the 6th International Workshop on Coq for Programming Languages (CoqPL)*, New Orleans, LA, January 2020.

[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)*, Virtual conference, July 2021.

[7] 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 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010. doi: 10.1145/1807128.1807152.

[8] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. Distributed causal memory: modular specification and verifica-

tion in higher-order distributed separation logic. In *Proceedings of the 48th ACM Symposium on Principles of Programming Languages (POPL)*, Virtual conference, January 2021.

[9] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 646–661, Philadelphia, PA, June 2018.

[10] 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.

[11] 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.

[12] 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.

[13] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*, pages 45:1–45:32, New Orleans, LA, January 2020.

[14] 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.

[15] 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.

[16] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 77:1–30, St. Louis, MO, September 2018.

[17] 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.

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

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

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

[21] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 348–370, Dakar, Senegal, April–May 2010.

[22] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 357–370, St. Petersburg, FL, January 2016.

[23] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), December 1975.

[24] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, Cambridge, MA, November 1988.

[25] 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.

[26] 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.

[27] Redis. `https://redis.io/`. Version 6.2.5.

[28] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002.

[29] 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.

[30] Ji-Yong Shin, Jieung Kim, Wolf Honoré, Hernán Vanzetto, Srihari Radhakrishnan, Mahesh Balakrishnan, and Zhong Shao. WormSpace: A modular foundation for simple, verifiable distributed systems. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, pages 299–311, Santa Cruz, CA, November 2019.

[31] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 691–707, Edinburgh, United Kingdom, June 2014.

[32] 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.